Blue text on a white background

Description automatically generated

**Comparative Analysis of MySQL and MongoDB in a High-Concurrency Ticketing System: A Practical Implementation Study**

**Gabriel Alberto Avina Solares**

Being a thesis presented for the award of

*Masters in Software Solutions Architecture*

Supervisor: Cormac Keogh

Technological University Dublin, Tallaght Campus

School of Enterprise Computing and Digital Transformation

December 2024

**DECLARATION**

I hereby certify that the material, which I now submit for assessment on the programmes of study leading to the award of Master of Science, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Signature of Candidate Date

**ABSTRACT**

Selecting an appropriate database system is vital for applications requiring high concurrency, as it involves balancing transactional integrity with schema flexibility. This research presents a comparative analysis of MySQL and MongoDB by simulating concurrent ticket booking scenarios, evaluating their performance under user loads ranging from 1 to 5,000 simultaneous requests. A Java-based ticketing system was implemented—utilizing Hibernate for MySQL and Morphia for MongoDB—to assess how each database handles transactional behaviour, schema modifications, and data consistency. The findings reveal that MySQL offers superior transactional integrity and consistent performance scaling, maintaining stable query response times even under substantial load. In contrast, MongoDB demonstrates greater schema flexibility and better initial performance with moderate parallel loads but exhibits increasing variability at higher concurrency levels. Schema modifications in MySQL required meticulous planning and timing, while MongoDB facilitated seamless schema evolution with minimal impact on performance. These insights enhance the understanding of database selection criteria for high-concurrency systems, indicating that MySQL is preferable for applications prioritizing strict data consistency, whereas MongoDB is advantageous for scenarios necessitating rapid schema evolution.

**ACKNOWLEDGEMENTS**

**To Silvia, Karla, and Andrea**

**To my friends and supervisor**

**A mis abuelos sláinte mhaith…**

Table of Contents

[**LIST OF FIGURES** 1](#_Toc184309914)

[**LIST OF TABLES** 2](#_Toc184309915)

[**ABBREVIATIONS** 3](#_Toc184309916)

[**INTRODUCTION** 4](#_Toc184309917)

[**Research Objectives** 5](#_Toc184309918)

[**Research Questions** 6](#_Toc184309919)

[**Research Hypotheses** 6](#_Toc184309920)

[**Significance of the Study** 6](#_Toc184309921)

[**1** **Literature Review** 7](#_Toc184309922)

[**1.1** **Database Management Systems Overview** 7](#_Toc184309923)

[**1.2** **Relational Databases** 7](#_Toc184309924)

[**1.2.1** **Overview and key features** 7](#_Toc184309925)

[**1.2.2** **MySQL** 8](#_Toc184309926)

[**1.3** **NoSQL Databases** 9](#_Toc184309927)

[**1.3.1** **Overview and key features** 9](#_Toc184309928)

[**1.3.2** **MongoDB** 11](#_Toc184309929)

[**1.4** **Transactional Processing and Concurrency** 12](#_Toc184309930)

[**1.5** **Comparative Studies** 13](#_Toc184309931)

[**2** **Methodology** 15](#_Toc184309932)

[**2.1** **Research Design** 15](#_Toc184309933)

[**2.2** **Technical Infrastructure** 16](#_Toc184309934)

[**2.3** **Analysis Methods** 17](#_Toc184309935)

[**2.3.1** **Quantitative** 17](#_Toc184309936)

[**2.3.2** **Qualitative** 19](#_Toc184309937)

[**2.4** **Research Validation Strategy** 19](#_Toc184309938)

[**2.4.1** **Architectural overview** 19](#_Toc184309939)

[**2.4.2** **Validation support** 20](#_Toc184309940)

[**2.5** **Testing Methodology** 20](#_Toc184309941)

[**2.5.1** **Performance test scenarios** 20](#_Toc184309942)

[**2.5.2** **Concurrency test cases** 21](#_Toc184309943)

[**2.5.3** **Schema modification tests** 21](#_Toc184309944)

[**2.5.4** **Data and process validation** 21](#_Toc184309945)

[**3** **Implementation** 22](#_Toc184309946)

[**3.1** **MySQL** 24](#_Toc184309947)

[**3.1.1** **System architecture and technical stack** 25](#_Toc184309948)

[**3.1.2** **Data Model Design** 25](#_Toc184309949)

[**3.1.3** **Transaction management** 27](#_Toc184309950)

[**3.2** **MongoDB** 29](#_Toc184309951)

[**3.2.1** **System architecture and technical stack** 29](#_Toc184309952)

[**3.2.2** **Document model design** 30](#_Toc184309953)

[**3.2.3** **Transaction management** 32](#_Toc184309954)

[**4** **Results** 34](#_Toc184309955)

[**4.1** **Test 1** 35](#_Toc184309956)

[**4.1.1** **Transactional behaviour** 35](#_Toc184309957)

[**4.1.2** **Impact of schema design** 35](#_Toc184309958)

[**4.1.3** **Performance patterns** 36](#_Toc184309959)

[**4.2** **Test 2** 37](#_Toc184309960)

[**4.2.1** **Transaction behaviour** 37](#_Toc184309961)

[**4.2.2** **Impact of schema design** 37](#_Toc184309962)

[**4.2.3** **Performance patterns** 38](#_Toc184309963)

[**4.3** **Test 3** 38](#_Toc184309964)

[**4.3.1** **Transaction behaviour** 38](#_Toc184309965)

[**4.3.2** **Impact on schema design** 39](#_Toc184309966)

[**4.3.3** **Performance patterns** 39](#_Toc184309967)

[**4.4** **Test 4** 40](#_Toc184309968)

[**4.4.1** **Transaction behaviour** 40](#_Toc184309969)

[**4.4.2** **Impact on schema design** 40](#_Toc184309970)

[**4.4.3** **Performance patterns** 41](#_Toc184309971)

[**4.5** **Test 5** 42](#_Toc184309972)

[**4.5.1** **Transaction behaviour** 42](#_Toc184309973)

[**4.5.2** **Impact on schema design** 43](#_Toc184309974)

[**4.5.3** **Performance patterns** 43](#_Toc184309975)

[**5** **Discussion** 46](#_Toc184309976)

[**5.1** **Restating the Research Problems and Questions** 46](#_Toc184309977)

[**5.2** **Summary of Findings** 46](#_Toc184309978)

[**5.2.1** **Transactional behaviour** 46](#_Toc184309979)

[**5.2.2** **Schema design** 47](#_Toc184309980)

[**5.2.3** **Performance** 48](#_Toc184309981)

[**5.3** **Implications** 48](#_Toc184309982)

[**5.4** **Limitations and Considerations** 49](#_Toc184309983)

[**6** **Conclusion** 50](#_Toc184309984)

[**6.1** **Answers to Research questions** 50](#_Toc184309985)

[**6.2** **Contributions** 51](#_Toc184309986)

[**6.3** **Recommendations and future research** 52](#_Toc184309987)

[**7** **References** 54](#_Toc184309988)

[**Appendices** 63](#_Toc184309989)

[**A MySQL System Architecture** 63](#_Toc184309990)

[**B Data Initializer** 64](#_Toc184309991)

[**C MySQL Class Diagram** 65](#_Toc184309992)

[**D Booking Service class** 66](#_Toc184309993)

[**E Simulation Class Metrics – MySQL** 67](#_Toc184309994)

[**F MongoDB System Architecture** 68](#_Toc184309995)

[**G Document Structure Diagram for MongoDB** 69](#_Toc184309996)

[**H MongoDB Application Schema** 71](#_Toc184309997)

[**I MongoDB Class Diagram** 72](#_Toc184309998)

[**J Booking ticket method** 73](#_Toc184309999)

[**K Booking Simulation Class** 74](#_Toc184310000)

[**L Transactional processes** 75](#_Toc184310001)

# **LIST OF FIGURES**

[**Figure 3.3.1 Implementation Phase** 24](#_Toc184309067)

[**Figure 3.3.2 Data Model Schema** 26](#_Toc184309068)

[**Figure 3.3.3. Transaction Flow** 33](#_Toc184309069)

[**Figure 4.1.1 Performance Metrics Comparison Test 1** 37](#_Toc184309070)

[**Figure 4.2. Performance Metrics Comparison Test 2** 39](#_Toc184309071)

[**Figure 4.3. Performance Metrics Comparison Test 3** 40](#_Toc184309072)

[**Figure 4.4. Performance Evolution Over Time Test 4** 42](#_Toc184309073)

[**Figure 4.5. Total Time Performance Test 4** 42](#_Toc184309074)

[**Figure 4.6. Query and schema modification performance** 43](#_Toc184309075)

[**Figure 4.7. Performance Evolution Over Time Test 5** 44](#_Toc184309076)

[**Figure 4.8.Total Time Performance Test 5** 45](#_Toc184309077)

[**Figure 4.9. Query and Schema Modification Performance** 45](#_Toc184309078)

# **LIST OF TABLES**

[**Table 4.4.1 Schema Changes** 35](#_Toc184309087)

[**Table 4.2 Test 1: Run Comparison** 36](#_Toc184309088)

[**Table 4.3 Test 2 Run Comparison** 38](#_Toc184309089)

[**Table 4.4. Test 3: Run Comparison** 40](#_Toc184309090)

[**Table 4.5. Test 4: Run Comparison** 41](#_Toc184309091)

[**Table 4.6. Test 5: Run Comparison** 44](#_Toc184309092)

# **ABBREVIATIONS**

|  |  |
| --- | --- |
| Binary-Encoded Serialization Of JSON | BSON |
| Atomicity, Consistency, Isolation, and Durability. | ACID |
| Basically available, soft state, and eventually consistent | BASE |
| Computation-Independent Model | CIM |
| Create, read, update and delete | CRUD |
| Data Access Objects | DAO |
| Deep Learning | DL |
| Extracting, Transforming, And Loading | ETL |
| Internet Of Things | IoT |
| Java Persistence API | JPA |
| JavaScript Object Notation | JSON |
| Linked Open Data | LOD |
| Model-Driven Architecture | MDA |
| Machine Learning | ML |
| Natural Language Processing | NLP |
| Not Only SQL | NoSQL |
| Object Document Mapper | ODM |
| Online Analytical Processing | OLAP |
| Object Relational Mapper | ORM |
| Platform Independent Data Metamodel | PIDM |
| Platform Independent Model | PIM |
| Platform-Specific Model | PSM |
| Relational Database Management System | RDBMS |
| Structured Query Language | SQL |
| Structural Topic Modelling | STM |
| User-Generated Context (), | UGC |
| Extensible Marup Language | XML |

# **INTRODUCTION**

The complexity of data management in modern applications presents significant challenges in selecting the appropriate tool to handle, store, and analyse business information. While traditional relational database management systems (RDBMS) offer strong transactional integrity and enforce rigid schema designs they can be limiting for certain types of data. However, they are still suitable for applications that require strict consistency and complex transactions (Patil et al., 2017).

Conversely, NoSQL databases have emerged to address the limitations of RDBMS by offering flexible schemas, horizontal scalability, and efficient handling of large volumes of unstructured or semi-structured data (Sudiartha et al., 2020). This makes them strong candidates for applications like property listing management systems, which involve nested data structures, user-generated content, and rapid evolving data requirements.

Despite the availability of both solutions, selecting the most appropriate database system for specific application requirements remains a complex task (Capris et al., 2022; Shareef et al., 2022; Yedilkhan et al., 2023). Existing comparative studies often provide broad overviews without exploring the practical implications of database selection based on workload characteristics and application contexts (C. Győrödi et al., 2015). While some researchers highlight the superior performance of NoSQL in handling data loads and dynamic datasets (Chang & Chua, 2019; Sudiartha et al., 2020; Wodyk & Skublewska-Paszkowska, 2020), RDBMS continue to be preferred in scenarios requiring strong data integrity and complex transactions (Stonebraker & Pavlo, 2024).

Developers and organizations need detailed insights to make informed decisions when selecting adequate software that balances transactional integrity, development agility, schema design flexibility, and the control of complex data relationships This research aims to address this need by implementing a ticketing system using MySQL and MongoDB to critically analyse their transactional mechanisms, schema design patterns, and strategies for modelling complex data structures.

By simulating concurrent purchase attempts and managing data relationships within the simulation, the study seeks to provide practical insights into the development experiences and challenges associated with each database.

To ensure a cohesive and comprehensive exploration of the research topic, this thesis is structured into several chapters that address each aspect of the study. It began with the literature review establishing the theoretical framework and context for comparing MySQL and MongoDB. The methodology chapter outlined the research design, including the experimental setup and data collection methods used to simulate concurrent purchase attempts. Subsequent chapters detail the implementation of the ticketing system using both databases, providing insights into the practical challenges and solutions encountered. The results chapter presents the findings from the simulations, highlighting differences in data integrity, schema design, and management of nested data structures. Finally, the discussion interprets these findings in relation to the research objectives and existing literature, while the concluding chapter summarizes key insights, underscores their significance for developers and organizations, and recommends directions for future research.

## **Research Objectives**

**Primary Objective**

To analyse and compare the transactional behaviour, schema design, and management of nested data structures in MySQL and MongoDB by simulating concurrent booking attempts in a ticketing system.

**Secondary objectives**

1. To examine how MySQL handles transactional integrity and schema rigidity in the ticketing system, especially during simultaneous ticket purchases, and its effect on managing nested data structures.
2. To explore MongoDB’s method to transactional behaviour and schema flexibility in the same ticketing scenario, analysing how it handles concurrent transactions and the impact of its flexible schema on modelling complex data structures.
3. To compare the development experiences and challenges encountered when implementing the ticketing system in both MySQL and MongoDB, focusing on transaction handling, schema design flexibility versus rigidity, and strategies for representing complex structures in each database.

## **Research Questions**

1. How does MySQL ensure transactional integrity in a high-concurrency ticketing system, and what challenges arise from its rigid schema when dealing with complex data relationships?
2. How does MongoDB handle transactions in a concurrent purchase scenario, and how does its flexible schema influence the modelling of intricate data structures?
3. What are the key differences in implementing transactional operations and data modelling between MySQL and MongoDB in the context of the ticketing system?

## **Research Hypotheses**

1. Concurrency handling

MySQL’s ACID compliance will provide better data consistency under high concurrency compared to MongoDB

1. Query performance

MongoDB will demonstrate superior performance in high-concurrency scenarios compared to MySQL due to its document-based architecture

1. Schema flexibility impact

Schema modifications will have a greater performance impact on MySQL than MongoDB during high-concurrency periods

## **Significance of the Study**

This research provides practical insights into the performance and behaviour of MySQL and MongoDB under high-concurrency conditions. By focusing on a real-world application scenario, the findings aim to assist developers and organizations in making informed decisions about database selection, balancing factors such as transactional integrity, development agility, and schema flexibility.

# **Literature Review**

## **Database Management Systems Overview**

Database management systems evolved from their initial file-system origins to become sophisticated platforms capable of handling complex data operations and concurrent access patterns (Vathy-Fogarassy & Hugyák, 2017). This evolution reflected the growing demand of modern applications, particularly in parallel environments.

Early applications surged as an option to the limitations of file systems, introducing fundamental capabilities that defined their architecture (Van Der Loo & De Jonge, 2020). The storage control layer provided sophisticated mechanisms for data persistence, moving beyond simple file storage to implement buffer management strategies between main memory and disk storage. As a response on this foundation, the query processing framework enhanced data accessibility through compilation algorithms and execution planning. The transaction processing component established mechanisms for concurrent access control and recovery management, ensuring data integrity through ACID compliance – atomicity, consistency, isolation, and durability (Ziegler & Dittrich, 2004).

The rise of NoSQL databases introduced new paradigms in data management (Tudorica & Bucur, 2011). While traditional RDBMS emphasized strict ACID applications through robust locking mechanisms, NoSQL systems characterized more flexible processes suited to distributed architectures. This architectural divergence became relevant in synchronous scenarios, where different approaches to transaction control and parallel control influenced system behaviour under intensive workloads.

Through this evolution, databases demonstrated increasing sophistication in managing complex operational requirements while maintaining data consistency and reliability.

## **Relational Databases**

### **Overview and key features**

Despite numerous attempts to replace traditional RDBMS, the relational model has consistently prevailed by absorbing beneficial features from multiple models while maintaining its core strengths (Stonebraker & Pavlo, 2024). Although modern technologies often promise revolutionary changes, its ability to evolve while preserving essential ACID properties has ensured continued dominance.

Yang et al. (2009) work on database summarization provided insights into managing database relationships. Their research demonstrated that relational databases excelled at maintaining structured connections and enforcing data integrity constraints, which are critical for architectures requiring accurate state management. The authors proposed a method for understanding and optimizing database structures.

Shareef et al. (2022) found that traditional SQL databases remained superior for applications demanding strong transactional guarantees and complex query capabilities. The synthesis of these studies highlighted several themes:

1. Transaction management: RDBMS continued to offer the most robust for ACID transactions. As demonstrated by Stonebraker & Pavlo (2024), attempts to relax these guarantees often lead to complications that outweigh any performance benefits.
2. Scalability considerations: While conventional databases face challenges with horizontal scaling, modern implementations have evolved to address these limitations. Shareef et al. (2022) analyses showed that cloud-based solutions like Amazon RDS effectively balance scalability needs with transactional integrity.
3. Data consistency: Yang et al. (2009) emphasized the importance of consistent relationships within complex structures. This is crucial in implementations like this study, where multiple users may simultaneously access and modify shared resources.

The evolution of RDBMS has been characterized by continuous improvement rather than revolutionary changes. Platforms have adopted beneficial features from alternative models while preserving their fundamental strengths. This adaptability, combined with strong support for complex transactions and relationships, establishes RDBMS as the foundation for reliable, and scalable concurrency systems.

### **MySQL**

MySQL has long been recognized as a versatile and efficient tool, widely utilized in diverse fields such as e-commerce, healthcare, and finance. Its architecture was designed to support high performance, scalability, and dependability, making it a preferred choice for both small-scale applications and enterprise-grade systems. As an open-source database, MySQL evolved through consistent innovation, incorporating features that addressed contemporary data management challenges (Šušter & Ranisavljević, 2023).

Central to MySQL’s functionality was its client-server architecture, which facilitated efficient communication between multiple clients and the server. This design enabled the database to control numerous concurrent connections, thereby ensuring reliability in high-demand environments. Within this framework, core components such as the SQL parser, query optimizer, and execution engine worked to process and deliver results seamlessly.

Moreover, the software introduced features aimed at improving scalability and efficiency. Sharding, enabled the distribution of data across multiple servers, improving performance for large datasets. Query caching store frequently executed query results in memory, reducing computational overhead for repetitive operations. Partitioning further optimized performance by dividing large tables into smaller, manageable segments, which streamlined query execution and reduced lock contention (Stanescu et al., 2016).

Over time, MySQL continued to evolve with each release, introducing enhancements that reflected advancements in database technology. MySQL 8.0 brought support for JSON data types, window functions, and improved handling of large datasets, aligning the database with modern application requirements (Besleaga, 2016; C. A. Győrödi et al., 2021).

Despite its strengths, the evolution was not without challenges. As databases expanded, maintaining performance required careful optimization through techniques such as physical programming and data tuning. These strategies involved refining the physical storage of data, optimizing indexes, and adjusting server configurations to reduce latency and improve query efficiency. Furthermore, ensuring data privacy and compliance with regulations presented additional complexities.

## **NoSQL Databases**

### **Overview and key features**

The term NoSQL (Not Only SQL) was first introduced in 1998 by Carlo Strozzi as the name of his small RDBMS that did not use SQL for data manipulation. The concept subsequently evolved, particularly from 2009 onward, as distributed data management systems began moving away from traditional ACID transaction support (Lith & Mattsson, 2010). This shift reflected the growing need to manage diverse, unstructured data in modern applications.

NoSQL databases encompass four primary architectural approaches, each optimized for specific use cases:

* + 1. Key-Value Stores

Represented the foundational NoSQL implementation, utilizing simple key-value pair storage mechanisms. Research by Hecht and Jablonski (2011), and Amghar, Cherdal and Mouline (2020) demonstrated their effectiveness in high-speed operations, though showing limitations in complex query scenarios. These architectures excel in caching and session management applications, where rapid data access is essential.

* + 1. Document-Oriented Databases

These programs advanced beyond basic key-value functionality, introducing sophisticated document-based storage using JSON, BSON, or XML formats. Kuznetsov and Poskonin (2014) highlighted their enhanced capabilities in managing unstructured data through flexible schemas and robust indexing mechanisms. This model demonstrated strength in content management and real-time analytics applications.

* + 1. Column-Family Stores

Inspired by Google’s Bigtable, column-family stores introduced innovative techniques to data organization. Research emphasized their good scalability across distributed systems and efficient storage utilization through sparse matrix implementations (Amghar et al., 2020; Kuznetsov & Poskonin, 2014). Nevertheless, ul Haque et al. (2019) identified challenges regarding partitioning strategies and performance optimization.

* + 1. Graph Databases

Emerged as specialized solutions for managing complex relational data structures. Studies by Amghar *et al.* (2022), and Thakare *et al.*(2023) documented their effectiveness in applications requiring sophisticated relationship traversal, though noting scalability challenges in distributes environments.

The selection of appropriate NoSQL architecture necessitated detailed consideration of specific use case requirements. Research indicated that document-oriented databases provided good balance for applications requiring flexible schema design while maintaining robust query capabilities.

This examination established fundamental understanding of their capabilities and limitations, providing a relevant context for subsequent detailed analysis of specific implementations such as MongoDB.

### **MongoDB**

The evolution of databases introduced a new model based on document-oriented architecture as a significant advancement in data handling capabilities. As a subset of NoSQL databases, designed to store, retrieve, and manage information, typically in formats like JSON, BSON, or XML Unlike traditional relational databases that use tables with rows and columns, document-oriented store data as documents within collections, offering a flexible schema that allows for varying data structures within the same collection (Mok, 2021).

Sen and Mukherjee (2024) demonstrated how this document-centric approach enhanced the system’s ability to manage diverse data types including strings, numbers, dates, arrays, and sub-documents. However, their research exhibited limitation in examining the long-term implications of this flexibility on application maintenance and data integrity.

Examination of schema design methodologies revealed nuanced considerations regarding data organization. Imam *et al.* (2018) presented compelling evidence supporting the strategic use of denormalization to enhance read performance. The decision between embedding data within documents and referencing other documents depends on factors like data volatility and access patterns. Embedding is often recommended when records are frequently queried together, improving read efficiency, while referencing is preferred for write-heavy workloads or when data is volatile.

Gallinucci, Golfarelli, and Rizzi (2018) addressed the complexities of schema variation through profiling techniques. Their implementation of decision trees for schema analysis showed promising results, yet questions remained regarding scalability in large-scale deployments.

Regarding performance characteristics and scalability, investigations yielded insights into MongoDB’s capabilities. Stonebraker's (2010) work on horizontal scaling through sharding stablished fundamental principles for distributed data management. Building on this foundation, Carvalho, Sá and Bernardino (2023) conducted comprehensive comparative analysis, revealing MongoDB’s superior performance in read-heavy operations while identifying limitations in scan-intensive scenarios.

Further research examined diverse scenarios, Diaz-Ordoñez, Rodríguez Baena and Yun-Casalilla (2023) highlighted the benefits of dynamic schema architecture in reducing system downtime during structural changes. However, their analysis lacked exploration of the potential risks associated with schema evolution.

Despite their advantages, document databases present challenges and limitations. According to Imam *et al.* (2018) data integrity concerns arise due to the lack of rigid schemas, potentially leading to inconsistencies. While the flexible schema offer adaptability, they require careful management to prevent data anomalies (Sen & Mukherjee, 2024).

Although transaction support has improved, it may not be as robust as in relational databases for complex, multi-document transactions. Additionally, managing write-heavy operations can be challenging, as schema designs often prioritize read optimization over write performance.

The investigation of system integration techniques provided practical insights for implementation. Seghier and Kazar (2021) presented evidence supporting polyglot persistence strategies in microservices architectures. Their research demonstrated the potential benefits of hybrid approaches, though long-term maintenance considerations warranted further investigation.

The literature analysis revealed several methodological limitations. Many studies focused on specific aspects of MongoDB implementation without considering broader system architecture implications. While challenges exist, particularly in data integrity, complex transactions, learning curve, etc, these can be mitigated through careful schema design, validation mechanisms, and ongoing training and tool development. Additionally, optimizing transaction management in complex distributed environments would offer valuable insights for system architects.

## **Transactional Processing and Concurrency**

Early work by Bernstein and Goodman (1981), referred to concurrency control mechanisms that enable users to access the database simultaneously while maintaining the illusion that each transaction executes in isolation on a dedicated system. Their research established the theoretical foundation for concurrency control, introducing essential concepts that continue to influence modern implementations.

Subsequently, Stonebraker *et al.* (2007), characterized the nature of modern OLTP workloads, manifest three distinctive characteristics: 1) they are short-lived, (2) they touch a small subset of data using index lookups, and (3) they are repetitive.

Yu *et al.* (2014), conducted an evaluation of concurrency control mechanisms in many core-environments. Their methodologically rigorous study exposed significant scalability limitations across different protocols. However, the research primarily focused on single-node scenarios, leaving distributed aspects unexplored.

Addressing this gap, Harding *et al.* (2017) extended the investigation to distributed environments through their innovative Deneva framework. Their comparative analysis of six concurrency control protocols provided empirical evidence that no single approach optimally addressed all scenarios. The study’s strength lies in its systematic evaluation methodology and reproducible framework. However, the research acknowledged limitations in workload diversity and hardware configurations tested.

More recently, Xia *et al.* (2022) introduced novel methods for transaction verification through cryptographic techniques. While their work presented solutions for ensuring transaction integrity, the performance implications of their procedure in high throughput environments remain incompletely understood.

Through these studies, several consistent patterns emerge. First, the inherent friction between concurrency and consistency continues to challenge system designers. Additionally, network latency and protocol overhead persistently impact distributed transaction performance, despite advances in hardware and software techniques.

The progression from theoretical analysis to comprehensive empirical evaluation frameworks represents significant advancement in the field. However, variations in experimental conditions and metrics sometimes complicate direct comparison between studies. Furthermore, the interaction between different consistency models and various workload patterns requires deeper examination. The effects of modern cloud infrastructure on protocol behaviour also warrant additional study.

## **Comparative Studies**

The evolution of data management systems has led to significant discussions regarding the suitability of traditional RDBMS versus NoSQL databases for handling modern data demands. This section examines studies chronologically, focusing on their contributions to understanding the performance, scalability and transactional integrity characteristics relevant to ticketing systems.

Stonebraker (2010) conducted an analysis challenging assumptions about NoSQL superiority over traditional RDBMS. The study methodically evaluated performance claims, demonstrating that perceived NoSQL advantages stemmed primarily from reduced overhead in logging, locking, and buffer management rather than fundamental architectural superiority. While the research provided considerable information into database architecture implications, its methodology focused on theoretical analysis rather than empirical testing.

Based on Stonebraker’s work, Li (2010) investigated the challenges of managing unstructured data in RDBMS environments. The research highlighted significant complexities in storing and retrieving unstructured data within traditional relational schemas, often necessitating additional architectural layers or BLOB implementations. Although, comprehensive in its analysis of RDBMS limitations, the study did not fully explore emerging solutions for handling unstructured data in modern implementations.

Cooper *et al.* (2010) introduced the Yahoo! Cloud Serving Benchmark (YCSB), establishing a standardized framework for evaluating cloud database performance. Their empirical testing revealed that MySQL demonstrated superior performance in read-intensive workloads, comparing to Cassandra. These findings were further supported by Patil *et al.* (2017) research, which documented MongoDB’s consistent insertion performance (maintaining 0.01 seconds) even as dataset sizes increased, contrasting with MySQL’s progressive performance degradation (0.0511 to 0.00698 seconds) under identical conditions.

Sudiartha et al. (2020) examined database scalability in the context of mobile applications, demonstrating NoSQL databases’ advantages in handling heterogenous data types. The research effectively illustrated MongoDB’s flexibility in adapting to evolving data structures, though its scope remined limited to mobile application scenarios. This aligned with findings from Amghar, Cherdal and Mouline (2022) who emphasized NoSQL’s inherent advantages in distributed environments. On the other hand, MySQL’s traditional vertical scaling technique demonstrated limitations in high-concurrency scenarios, though Capris *et al.* (2022) noted its continued effectiveness for specific workload patterns.

Li (2010) and Digittrix (2023) documented challenges in MySQL’s rigid schema structure, particularly when handling unstructured data. MongoDB’s flexible schema design, as analysed by Thakare *et al.* (2023), provided relevant advantages in managing diverse data types, though Stonebraker (2010) cautioned that this flexibility could compromise data integrity in certain scenarios.

Research by Candel, Sevilla Ruiz and García-Molina (2022) addressed aspects of system integration, proposing unified metamodels to facilitated migration between platforms. Reinero (2017) emphasized the importance of understanding fundamental differences in data modelling methods during system transitions, particularly relevant for organizations considering platform migration.

The examination of consistency model mechanisms revealed differences: MySQL’s adherence to ACID properties, as documented by Stonebraker (2010), ensured robust data consistency at the expense of certain performance characteristics. MongoDB’s eventual consistency model, following BASE principles (Basically available, soft state, and eventual consistency) (Thakare *et al.* 2023), prioritized availability and partition tolerance, though potentially introducing temporary data inconsistencies.

# **Methodology**

## **Research Design**

This study adopted a comparative experimental research approach (Reisner, 1988), to examine how MySQL and MongoDB handle transactional behaviour, schema rigidity, and nested structures within a ticketing system context. The experimental design facilitated a direct comparison of database performance under controlled conditions through systematic implementation and testing of identical functionality in both systems.

The selection of a comparative experimental methodology was informed by previous studies (C. Győrödi et al., 2015; Patil et al., 2017; Capris et al., 2022; Stonebraker & Pavlo, 2024), which demonstrated the effectiveness of controlled testing in revealing performance differences between SQL and NoSQL databases. This path enabled objective measurement of each database’s capabilities in managing concurrent transactions, maintaining data consistency, and handling data relationships.

This approach enabled the objective measurement of each database’s capabilities in managing concurrent transactions, maintaining data consistency, and handling data relationships. The study is based on pragmatism, emphasizing practical outcomes and real-world applicability. By implementing a simulation rather than relying solely on theoretical analysis, the proposal aimed to produce findings that are directly relevant to practitioners facing similar database selection challenges. This aligned with the recommendations of Shareef, Sharif and Rashid (2022), who identified the need for studies that provide actionable insights into database performance in specific application scenarios. The experimental framework encompassed three key areas of investigation.

The experimental framework encompassed three key areas of investigation:

1. Transactional behaviour in concurrent ticket booking scenarios
2. Impact of schema design processes on system implementation
3. Management of nested data structures in booking records

The comparative analysis was structured through:

1. Controlled test scenarios that examined basic booking operations, transaction handling methods, and schema management techniques
2. Systematic comparison of implementation differences, data handling strategies, and transaction management mechanisms
3. Quantitative measurement of performance metrics and qualitative assessment of development experiences

This methodological framework enabled a systematic examination of how each database system addressed the core research objectives through controlled experimentation and structured comparison.

## **Technical Infrastructure**

All the experiments were implemented on a development workstation Windows 11 Home. The system utilized an Intel(R) Core (TM) i5-12500H Processor 12th Generation, 16gb ram, 2500Mhz and SSD for storage operations, 12 Core(s), 16 Logical Processor(s). Network connectivity was maintained through 500mb ethernet connection to minimize latency impacts on database operations.

The software environment used MySQL Community Server 8.0. and MongoDB Community Service 8.0.3. Database management was assisted through MySQL Workbench 8.0.40 and MongoDB Compass 1.44.6 respectively. The development stack included OpenJDK 23.0.1 for core implementation, with Eclipse IDE 2023-09 serving as the primary development environment. The MongoDB shell, Mongosh 2.3.3, was used for direct interaction with the MongoDB database. Maven 3.9.5 managed project dependencies and build automation. Version control was maintained through Git 2.42.0, with project artefacts stored in a private repository. Test data generation utilized Mockaroo’s for creating realistic user profiles.

The system architecture implemented a separation of concerns across:

* Entity definitions
* Data Access Objects (DAO)
* Service layer
* Test simulation framework

## **Analysis Methods**

### **Quantitative**

The method employed to evaluate the performance characteristics and behavioural patterns of MySQL and MongoDB was collected with instrumented service classes and testing scenarios. This proposal aligned with the methodology proposed by Osman and Knottenbelt (2012) for database performance evaluation.

In terms of success rate, the formula utilized was:

*Success Rate = (Successful Bookings / Total Booking Attempts) × 100*

According to Bernstein & Newcomer (2009b) transaction success rate directly correlates with system reliability and user experience in OLTP (Online Transaction Processing) systems. For ticketing systems specifically, a success rate above 95% is considered an industry standard for synchronous applications.

Another common solution is to use an *AtomicInteger* addition operation to advance a global logical timestamp. This requires fewer instructions and thus the DBMS’s critical section is locked for a smaller period (Yu et al., 2014).

*private final AtomicInteger successfulBookings = new AtomicInteger(0);*

*private final AtomicInteger failedBookings = new AtomicInteger(0);*

Consequently, the Average query time was taken in consideration to compare both databases' performance. For ticketing systems, Zhao et al. (2020) recommend maintaining average query times below 100ms for optimal user experience.

*Average Query Time (ms) = Total Query Time / Total Queries*

This implementation followed the measurement methodology described in Cai et al. (2019), which emphasizes that timing metrics are more reliable than individual query measurements.

For the concurrency metrics, an evaluation of transaction effectiveness was performed based on Kleppmann (2017) who said conflict rates in distributed systems proved insight into the effectiveness of concurrency control mechanisms.

*Conflict Rate = (Number of Concurrency Conflicts / Total Transactions) × 100*

The schema modification success rate indicated the adaptability of the database (Möller et al., 2020).

*Modification Success Rate = (Successful Modifications / Total Modification Attempts) × 100*

The formula for the optimal size of a thread pool was:

*int optimalThreadPoolSize = numberOfCores \* targetUtilization \* (1 + waitTime / computeTime)*

Metric for schema modification

*duration = (System.nanoTime() - startTime) / 1\_000\_000;*

The application the mean and coefficient of variation (CV) (Kaltenecker et al., 2023) was utilized to analyse and compare the performance metrics of MySQL and MongoDB.

Mean:

Purpose: The mean is used to determine the average value of a dataset, providing a central value that summarizes the data.

*Formula: Mean (x̅) = Sum of Values / Number of Values*

Application: It helped in understanding the typical performance of each database across various metrics such as total time, query time, and schema modification time.

The CV measured the relative variability of the data, expressed as a percentage. It allowed for the comparison of the degree of variation between datasets, regardless of their units or scales.

*Coefficient of variation = σ/μ × 100%*

Where 𝜎 is the standard deviation and 𝜇 is the mean.

Application: It was used to assess the consistency and stability of each database's performance by comparing the variability in metrics such as total time, query time, and schema modification time.

### **Qualitative**

Following Lenberg et al. (2024) framework for qualitative software engineering research, this study employed an interpretative analysis to examine behavioural patterns and implementation challenges in database systems. The methodology emphasized reflexivity and documentation of development experiences, aligned with established qualitative research practices in software engineering (Liang et al., 2023).

For validation, peer review sessions evaluated implementation approaches, ensuring adherence to best practices and design patterns, providing feedback on architectural decisions, and optimization opportunities.

The qualitative methodology complemented quantitative metrics by supplying context for performance variations and identifying underlying causes of observed behavioural differences between both database implementations.

## **Research Validation Strategy**

### **Architectural overview**

The architecture employed a layered design with three primary components: Data Access Layer, Service Layer, and Simulation Framework. Each layer served specific verificatioobjectives while maintaining the separation of concerns (Ingeno, 2018).

1. Data Access Layer

The data access layer established the foundation, providing consistent interfaces for database operations while isolating database-specific implementations. This procedure ensured that differences in performance and behaviour could be attributed to the underlying databases rather than implementation variations.

1. Service Layer

The service layer managed business logic and transaction coordination, implementing distinct strategies appropriate to each database’s capabilities. MySQL implementation used JPA/Hibernate transaction management with pessimistic locking to ensure data consistency under concurrent access. In contrast, MongoDB implementation utilized Morphia’s object-document mapping with optimistic concurrency control, reflecting the different approaches to transaction management. These strategies enabled the evaluation of the first hypothesis regarding concurrency handling.

1. Simulation Framework

This framework managed parallel booking operations through configured thread pools, allowing systematic testing of concurrent access patterns. It also incorporated metrics collection, tracking response times, and transaction resource utilization. These measurements provided quantitative data for evaluating the second hypothesis regarding query performance.

### **Validation support**

The architecture accommodated both relational and document-base data models while maintain functional equivalence to ensure a fair comparison. Core domain entities, including Events, Tickets, Booking, and Users, were implemented to preserve essential functionalities. This technique helped to evaluate the third hypothesis regarding schema flexibility by allowing runtime modifications while measuring their impact on system performance.

MySQL followed traditional relational modelling practices, using normalized tables with foreign key constraints to maintain referential integrity (*Best Practices for Modeling Relational Data in DynamoDB - Amazon DynamoDB*, n.d.). MongoDB adopted a document-oriented process utilizing embedding and referencing strategies appropriate to the data access patterns (*Schema Validation - MongoDB Manual v8.0*, n.d.).

Initial tests established baseline performance metrics for both implementations under normal operating conditions. Subsequent tests introduced controlled stress conditions, including parallel booking attempts and runtime schema modifications. The metrics collected provided evidence for evaluating the hypotheses while the controlled environment ensured reproducibility of results.

## **Testing Methodology**

### **Performance test scenarios**

The performance testing focused primarily on quantitative measurement of database behaviour under controlled conditions. Specifically, test scenarios were structured to evaluate system performance across multiple dimensions.

In the first phase, transaction response time analysis provided baseline performance data. Specifically, measured query execution duration across various operation types. Additionally, transaction competition rates underwent continuous monitoring to assess system throughput under different load conditions.

### **Concurrency test cases**

Following the performance evaluation, the methodology examined database behaviour under simultaneous access patterns, focusing on transaction isolation and resource contention handling under conditions similar to production environments.

During testing, the study concentrated on each database’s handling of concurrent transactions, observing transaction processing and resource allocation.

In addition to performance metrics, data consistency formed a central component of concurrency testing. Under these circumstances, each system showcased unique characteristics.

### **Schema modification tests**

The final phase focused on schema modification testing to assess structural adaptability. Initially, draw from Sadalage & Fowler (2012) work on schema evolution patterns, the testing progressed through increasing levels of complexity.

Basic schema alterations, such as column additions and modifications were performed to evaluate each database’s ability to maintain data integrity during structural changes while handling concurrent transactions.

Performance metrics were collected during schema modifications to assess impact on system availability and response times (Gallinucci et al., 2018). The evaluation focused on the ability to maintain transaction processing capabilities during structural changes.

### **Data and process validation**

Ensuring data integrity is fundamental for the reliable operation of any method, since it plays a leading role in detecting and correcting errors, inconsistences, and inaccuracies within datasets. The primary types of data validation employed were format and consistency validation.

Building upon the methodology proposed by Van Der Loo & De Jonge (2020) validation was meet as a surjective function mapping datasets to Boolean values. This method implanted through explicit validation rules in both MySQL and MongoDB databases, allowing for detection of data anomalies based on specific requirements:

1. Single-Point Validations: Focused on individual data points, such as checking the status of a ticket
2. Cross-Variable Validations: Examined relationships between different fields within a record to ensure logical consistency
3. Cross-Record Validations: Assessed constraints across multiple records, for instance, booking limitations affecting several tickets
4. Temporal Validations: Monitored changes in data over time, tracking the evolution of ticket statuses

Both databases employed different procedures to enforce consistency. MySQL maintained consistency through atomic transactions using *EntityManager,* along with the application of pessimistic locking for concurrent access control. Data integrity constraints were also utilized including foreign key relationships to maintain referential integrity, unique conditions to prevent duplicate bookings, and check restrictions to validate business rules.

In contrast, MongoDB application utilized *SessionFactory* to manage multi-document transactions that ensured atomicity across multiple collections. Optimistic concurrency control was implemented (in contrast to MySQL pessimistic locking) using version fields within documents. Finally, schema validation was enforced through JSON Schema definitions, specifying structure and data types for documents.

Validating concurrent operations was important to address the challenges posed by simultaneous access and modification of data by multiple users. Key considerations included handling race conditions, preventing deadlocks, and maintained consistency levels.

The schema validation process ensured that changes to the database did not disrupt ongoing operations or compromise data integrity. Conversely, MongoDB benefited from its flexible schema design, allowing dynamic updates without adversely affecting existing documents.

# **Implementation**

The implementation phase followed a structured approach to data initialization and transaction management, as illustrated in Figure 3.1. The initialization process was divided into three main components: the data initialization process, ticket initialization, and transaction management.

The data initialization process began with the instantiation of the initialization sequence, following a hierarchical order that respected entity relationships and dependencies. This process started by initializing basic entities and culminating with the most important. This strategy ensured that all dependencies were in place before dependent entities were created.

The ticket initialization process implemented a more complex logic, beginning with iterative loops through Events and their associated Ticket Categories. For each combination, the system generated tickets based on predefined seating configurations. The process incorporated validation checks to prevent duplicate tickets and maintain data integrity. When a ticket was generated, the system verified its uniqueness before creation and persisted it to the database, updating relevant counters and metadata.

Transaction management formed the initial component of the implementation, incorporating an error-handling mechanism. Each transaction was monitored for successful completion; in case of failure, the system executed a rollback procedure and logged the error details This approach ensured data consistency and provided a reliable audit trail of system operations.

A screenshot of a computer

Description automatically generated

**Figure 3.3.1 Implementation Phase**

## **MySQL**

The database design and transaction management utilized Java Persistence API (JPA) with Hibernate as the Object-Relational Mapping (ORM) framework. This implementation aimed to evaluate the performance in handling concurrent ticket booking operations while maintaining data consistency.

### **System architecture and technical stack**

The architecture employed a layered approach consisting of four primary levels (Appendix A). At the top, the Presentation Layer provided a foundation for future web interfaces and RESTful API endpoints. Below this, the Business Logic Layer incorporated three essential components: transaction management for handling errors and coordinating transactions, initializers for systematic data population, and services for managing core business operations.

For the architecture setup the process comprised three components:

1. Setup
   * Configured as a Maven project using the *java-quick-archetype*
   * Integrate Hibernate version 5.6.15Final for JPA implementation
   * MySQL Connector Java 8.0.33 to establish database connection
2. Database Configuration

The *persistence.xml* file was configured to define the persistence unit and database connection properties:

*<persistence-unit name="ticketingsystem" transaction-type="RESOURCE\_LOCAL">*

*<provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>*

*<properties>*

*<property name="javax.persistence.jdbc.url" value="jdbc:mysql://localhost:3306/ticketsystem"/>*

*<property name="javax.persistence.jdbc.user" value="root"/>*

*<property name="javax.persistence.jdbc.password" value="password"/>*

*<property name="javax.persistence.jdbc.driver" value="com.mysql.cj.jdbc.Driver"/>*

*</properties>*

*</persistence-unit>*

1. Connection Management

*DataInitializer* class located on the Appendix B was created to manage the connections, initialize Data Access Objects (DAOs), and populate the database.

### **Data Model Design**

The schema was designed following the principles of the third normal form (3NF) to minimize data redundancy and optimize query performance (Figure 3.2).

A screenshot of a computer

Description automatically generated

**Figure 3.3.2 Data Model Schema**

Key elements of the data model included:

1. Schema definition

The database was structured with tables representing core entities, such as booking

*CREATE TABLE bookings (*

*booking\_id INT PRIMARY KEY AUTO\_INCREMENT,*

*user\_id INT,*

*delivery\_address\_email VARCHAR(100),*

*delivery\_time TIMESTAMP,*

*time\_paid TIMESTAMP,*

*time\_sent TIMESTAMP,*

*total\_price DECIMAL(10,2) NOT NULL,*

*discount DECIMAL(10,2) DEFAULT 0,*

*final\_price DECIMAL(10,2) NOT NULL,*

*booking\_status ENUM('in-progress', 'confirmed', 'canceled') DEFAULT 'in-progress'*

*FOREIGN KEY (user\_id) REFERENCES users(user\_id)*

*);*

1. Entity Relationships

Relationships between entities were defined using JPA annotations, establishing connections like many-to-one and one-to-many

*@Entity*

*@Table(name = "bookings")*

*public class Booking {*

*@Id*

*@GeneratedValue(strategy = GenerationType.IDENTITY)*

*@Column(name = "booking\_id")*

*private int bookingId;*

*@ManyToOne(fetch = FetchType.LAZY)*

*@JoinColumn(name = "user\_id", nullable = false)*

*private User user;*

*@OneToMany(mappedBy = "booking", fetch = FetchType.****LAZY****)*

*Private List<BookingTicket> bookingTickets = new ArrayList<>();*

1. Data Access Layer

The DAO pattern encapsulated database operations, providing a separation between the data layer and business logic.

*public class BookingDAO {*

*private EntityManager em;*

*public BookingDAO(EntityManager em) {*

*this.em = em;*

*}*

*public List<Booking> findAll() {*

*TypedQuery<Booking> query = em.createQuery("SELECT b FROM Booking b", Booking.class);*

*return query.getResultList();*

*}*

*}*

Once the schema was created, the next step required the representation of the core domain model. The class diagram (Appendix C) showed the relationships between entities reflected the business rules of the ticketing system: Customers could place Orders, which contained *OrderItems*, each associated with a specific *Ticket*. Tickets were categorized by *TicketType* and linked to *Events*, which were hosted at specific *Venues*. This object-oriented design aligned closely with the relational database schema implementation, facilitating seamless object-relational mapping through JPA annotations.

### **Transaction management**

MySQL’s implementation adhered to ACID principles to ensure data consistency and reliability during high-concurrency ticket bookings. Notably, the system utilized pessimistic locking mechanisms, distinguishing it from the optimistic approach employed in the MongoDB implementation.

The *BookingService* class *(*Appendix D)managed all booking operations within transactional boundaries. It began by initializing the database connection, getting the available tickets for a specific event, start a transaction, verify user credentials, and lock the necessary tickets to prevent concurrent modifications. Upon successful validation, it calculated the total price, created a booking record, and updated the ticket statuses before committing the transaction. In case of any exceptions, the transaction was rolled back to maintain data integrity.

*Public Booking createBooking(int userId, List<String> ticketSerialNumbers, String deliveryEmail) throws Exception {*

*EntityTransaction transaction = em.getTransaction();*

*try {*

*transaction.begin();*

*User user = findAndValidateUser(userId);*

*List<Ticket> lockedTickets = lockTickets(ticketSerialNumbers);*

*Booking booking = createBookingEntity(user, deliveryEmail, lockedTickets);*

*transaction.commit();*

*return booking;*

*} catch (Exception e) {*

*transaction.rollback();*

*throw e;*

*}*

The *createBooking* method served as a transaction-managed operation. Starting with a new *EntityTransaction*, it processes booking requests through a structured sequence: validating the user's existence, acquiring pessimistic locks on the requested tickets to prevent concurrent access, and creating a booking record that links the user, tickets, and delivery information.

The implementation employed locking strategies through JPA's *LockModeType.PESSIMISTIC\_WRITE*, preventing concurrent modifications to tickets during the booking process. This approach aligned with traditional relational database transaction patterns, ensuring data consistency during high-concurrency scenarios.

The *TicketDAO* served as the primary resource manager, handling ticket state transitions and maintaining referential integrity between related entities.

*private Ticket lockTicket(String serial) {*

*return em.createQuery("SELECT t FROM Ticket t WHERE t.serialNumber = :serial AND t.status* *= :status", Ticket.class)*

*.setParameter("serial", serial)*

*.setLockMode(LockModeType.PESSIMISTIC\_WRITE)*

*.getSingleResult();*

}

The transaction workflow was structured through distinct phases:

1. Transaction initiation and resource acquisition
2. Entity locking and validation
3. State updates and relationship management
4. Transaction commit or rollback
5. Resource cleanup and metric updates

Concurrency was further managed through the *BookingSimulation* class, which employed a thread pool to handle multiple booking requests simultaneously. This setup simulated real-world scenarios where numerous users might attempt to book tickets concurrently.

*public class BookingSimulation {*

*// Configuration Constants*

*private static final int NUM\_USERS = 1000;*

*private static final int MAX\_TICKETS\_PER\_USER = 1;*

*private static final int NUMBER\_OF\_CORES = Runtime.getRuntime().availableProcessors();*

*private static final int THREAD\_POOL\_SIZE = NUMBER\_OF\_CORES \* 2;*

*private static final int SIMULATION\_TIMEOUT\_MINUTES = 3;*

Error handling mechanisms were in place to catch and manage exceptions that occurred during the transaction process. If a transaction failed, it was promptly rolled back, and appropriate metrics were updated to reflect the failure.

Within the *Simulation* class display some metrics such as simulation duration, successful and failed bookings, and average query time (Appendix E). This data facilitated the evaluation of system performance under various load conditions.

1. Configuration parameters including concurrent user count and thread pool size
2. Timing metrics measuring overall simulation duration and query response times
3. Success rates tracking both completed and failed booking attempts
4. Resource utilization patterns during high-concurrency periods

These metrics collection provided insights into system performance and reliability under load conditions, supporting the evaluation of the research hypothesis.

## **MongoDB**

### **System architecture and technical stack**

MongoDB implementation adopted a document-oriented approach, leveraging MongoDB’s capabilities. The layered architecture presented in Appendix F exhibited the interactions between different system components. As well as the MySQL implementation, it consisted of three primary layers: Business Logic Layer containing initializers for data population, a Data Access Layer implementing DAO patterns for CRUD operations, and an Entity Model Domain layer defining the document structure and relationships. The system used Morphia as the Object-Document Mapper (ODM) to facilitate seamless interaction between Java objects and MongoDB collections, while maintaining a clear separation of concerns throughout the application stack.

Key implementation aspects included:

1. Setup

A *DataInitializer* class was created to establish connections with MongoDB and configure the ODM using Morphia (Kumar, 2019).

*public DataInitializer() {*

*mongoClient = MongoClients.create("mongodb://localhost:27017");*

*datastore = Morphia.createDatastore(mongoClient, "ticketsystem");*

*datastore.getMapper().mapPackage("dev.morphia.example");*

*datastore.ensureIndexes();*

*}*

1. Technical components

* MongoDB Community Edition: Served as the primary database system
* MongoDB Compass: Provided a graphical interface for database management and visualization
* MongoDB Shell (mongosh)
* Morphia ODM (v2.2.6): Enabled ODM for Java applications
* MongoDB Driver Sync (v4.5.1): Managed synchronous database operations

### **Document model design**

The schema-less architecture of Mongo allowed each document within a collection to possess a distinct structure (Appendix G). In this context this flexibility supported embedded documents, thereby optimizing read performance by minimizing the need for join operations.

However, this procedure introduced challenges related to data consistency and redundancy. Without enforced schema constraints, maintaining consistent data structures across documents required detailed application-level validations.

The *Booking* entity referenced both *User* and *Ticket* documents, maintaining normalization by avoiding the embedding of extensive ticket arrays within a booking. *@Indexed annotations* ensured that fields maintained unique values. Attempting to insert duplicate values in these fields would result in a database error, thereby preserving data uniqueness.

To prevent duplicate entries and ensure uniqueness of critical fields, unique indexes were implemented across various collections. This strategy enforced data integrity, at the database level eliminating the possibility of duplicate records that could compromise system reliability.

On the Appendix I it is visible this technique, One-to-Many relationships implemented through embedding or referencing dependent entities. For example, a *User* could have multiple *Booking* documents, established via a one-to-many relationships where each booking referenced its associated user.

Implementation techniques:

* Morphia annotations:
  + *@Reference:* Facilitated referencing documents without embedding, maintaining loose coupling between entities
* Data integrity maintenance:
  + Application-level validation ensured referential integrity through application logic due to the lack of enforced foreign key constraints in MongoDB
  + Indexing creation on referenced fields to optimize join-like operations

Example:

*@Entity("events")*

*public class Event {*

*@Id*

*private ObjectId id;*

*private String name;*

*private Date date;*

*@Reference*

*private Venue venue; // Referenced for scalability*

*@Reference*

*private List<TicketCategory> ticketCategories; //*

*// Getters and Setters*

*}*

The *Event* entity referenced *TicketCategory* and *Venue* using @*Reference* annotation, enabling the retrieval of related data without embedding entire performer or venue documents within each event.

The MongoDB implementation's class diagram illustrated in the Appendix H and I, exhibit the structure of the ticketing system's core components. The diagram centred around the *Event* entity, which maintained relationships with *TicketCategory*, *Performer*, and *Venue* entities through document references. The *EventInitializer* coordinated with *EventDAO* to manage event creation and queries, while TicketCategoryInitializer worked with *TicketCategoryDAO* to handle ticket category management. Each entity possessed its own DAO class responsible for CRUD operations. Notably, the diagram highlighted how entities used *ObjectId* as their primary identifier and incorporated specific MongoDB data types, such as String for text fields and Set collections for managing relationships between documents.

### **Transaction management**

While traditional NoSQL systems often prioritize eventual consistency following the CAP theorem (Brewer, 2012), MongoDB’s introduction of multi-document ACID transactions starting from version 4.0 (O’Grady, 2020) marked a shift toward stronger consistency guarantees.

This study adopted a hybrid method according to Stonebraker (2010) observations regarding the necessity of maintaining transactional integrity in specific domains This experiment used what Gray and Lamport (2006) referred as “distributed transaction commit protocol” (displayed in Figure 3) modified for document-oriented databases.

A diagram of a process

Description automatically generated

**Figure 3.3.3. Transaction Flow**

This implementation followed the Repository Pattern (Fowler, 2002), separating transaction logic from data access concerns, enabling a clean transaction, where the *bookTicket* method *(*Appendix J*)* class acted as the primary transaction coordinator, managing multiple DAOs and maintained transaction metrics.

The protocol implemented a multi-phase commit pattern, aligning with the “ACID transaction protocol” (Faraj, 2022), involving distinct phases:

1. Resource Verification: Initially, the system validated ticket availability and user credentials
2. Resource Allocation: Subsequently, available tickets were reserved for the transaction
3. State Update: Following allocation, ticket status transitions were managed atomically
4. Transaction Commit: Finally, changes were permanently recorded or rolled back

To simulate high-concurrency scenarios, the *BookingSimulation* class (Appendix K) implemented a thread-pool based approach, utilizing multiple threads to perform booking operations concurrently (Seppälä, 2024). This design maximized system resource utilization and tested the system's ability to handle simultaneous transactions.

The class took in several key dependencies including instances of *BookingDAO, UserDAO, EventDAO, TicketDAO,* and *Datastore,* object from the Morphia library. These dependencies allowed the class to interact with the underlying data storage and retrieve the necessary information to carry out the simulation.

The simulation process proceeded through several methodically defined stages. Initially, the system retrieved registered users from *UserDAO*. Subsequently, it created callable tasks for each simulated booking attempt. These tasks were then executed concurrently through the thread pool, followed by results collection and verification. Finally, the system performed state validation to prevent overselling scenarios. Acting as primary resource manager, the *TicketDAO* managed the booking of available tickets.

Each ticket’s state was updated individually, adhering to a consistent status update where transitions were uniformly applied across operations, maintaining consistency.

In relation to King's (2024), MongoDB does not stand an pessimistic control, thus by default the optimistic concurrency control strategy was used to handle simultaneous transactions, in which the status-based availability was determined based on the ticket status, preventing overbooking.

This approach utilized status-based availability checking to prevent overbooking scenarios. Furthermore, each transaction operated within a *ClientSession* block to ensure atomicity during booking operations.

The state transition model followed a precise pattern:

* Initially, tickets transitioned from *Available* to *Reserved* during transaction initiation
* Subsequently, *Reserved* tickets moved to *Booked* upon successful transaction completion
* Alternatively, *Reserved* tickets reverted to *Available* in cases of transaction rollback

Through practical implementation, several critical factors emerged for maintaining transactional integrity:

1. Atomic Operations: The system ensured indivisible status updates
2. State Consistency: Transitions followed predetermined patterns
3. Session Management: Transactions operated within defined boundaries
4. Concurrency Control: Optimistic approaches prevented data conflicts
5. Error Handling: Robust mechanisms managed transaction failures

Through this approach to transaction management and performance monitoring, MongoDB demonstrated effective handling of concurrent booking operations while maintaining data consistency and providing detailed performance metrics for analysis.

# **Results**

Throughout this project the author has examined how MySQL and MongoDB handle transactional behaviour and schema modifications in a high concurrency system. Following the experimental framework outlined in the methodology, the investigation proceeds through systematic testing of both databases under controlled conditions.

The schema modification query was the same for both experiments through the five tests, and as shown in table 1, both queries were adapted for MySQL and MongoDB. The results of the five experiments conducted are described in this section.

**Table 4.4.1 Schema Changes**

|  |  |
| --- | --- |
| MySQL Schema Changes | MongoDB Schema Changes |
| *ALTER TABLE bookings*  *ADD COLUMN processing\_time5 TIMESTAMP,*  *ADD COLUMN payment\_method5 VARCHAR(50),*  *ADD COLUMN booking\_source5* *VARCHAR(50)* | *{*  *"$set": {*  *"booking\_source": "ONLINE",*  *"processing\_time": <Date>,*  *"payment\_details": {}*  *}*  *}* |

Five experiments adhered to this configuration:

Test Configuration details

* Concurrent Users:
  + Test 1 -> 1 booking attempt
  + Test 2 -> 10 booking attempts
  + Test 3 -> 100 booking attempts
  + Test 4 -> 1000 booking attempts
  + Test 5 -> 5000 booking attempts
* Max Tickets Per User: 1
* Thread Pool Size: 32 (NUMBER\_OF\_CORES \* 2)
* Simulation Timeout: 1 minute
* Database Operations: Booking + Schema Modifications

## **Test 1**

### **Transactional behaviour**

As shown in Table 2, MySQL demonstrated strong initial performance, completing all booking transactions in 191 milliseconds (ms) during Run 1, with and average query time of 72.43ms. The schema modification operations required 54ms, indicating efficient handling of structural changes. MongoDB exhibited comparable initial performance with a total transaction time of 195ms, though with unusual characteristics – a lower query time of 26.01ms but slightly higher schema modification time of 59.86ms. Both databases maintained 100% success rates for all booking attempts, demonstrating reliable transaction handling.

### **Impact of schema design**

The architectural differences between a rigid schema and a flexible document model manifested in distinct performance patterns for schema modifications. As detailed in the Table 2, MySQL modification time showed an interesting pattern of initial variance followed by stabilization, starting at 54ms and increasing to 111ms by Run 5. This behaviour suggests effective optimization of schema-related operations over time. In contrast, MongoDB schema modification time showed a consistent upward trend from 59.86 milliseconds to 108.61 by Run 5, indicating different underlying mechanisms for handling schema changes in its structure.

**Table 4.2 Test 1: Run Comparison**



### **Performance patterns**

The longitudinal analysis, visualized in Figure 4, revealed how each database approach influenced performance over multiple runs. MySQL had a gradual performance degradation pattern, with total execution time increasing from 191 to 446ms across the five runs, averaging 315ms (SD=85.2ms). Query times showed a stepwise increase from 72.43ms to 176.61ms, with an average of 118.52ms.

A graph of different colored bars

Description automatically generated

**Figure 4.1.1 Performance Metrics Comparison Test 1**

MongoDB’s performance characteristics differed, showing more pronounced changes over time. The total execution time increased more steeply from 195 to 636ms across the runs, averaging 416ms (SD =105.7ms). Query times rose from an initially 26.01ms to 106.86ms, averaging 266.1ms. Indicating that while Mongo offered better initial query performance, it was more susceptible to performance degradation.

## **Test 2**

### **Transaction behaviour**

Table 3 contains the transaction response times, and metrics of the Test 2. During the initial run, MySQL processed all booking transactions withing 382ms. The system showed efficient query handling, with individual queries taking an average of 36.25ms to complete. The time required for schema modifications in this first run was 57ms.

Mongo demonstrated distinct characteristics in the first run. All transaction were completed more quickly, requiring 283ms for total execution. However. The time needed for schema modifications was notably longer at 263.7ms. Despite these differences, both databases-maintained data consistency throughout their operations.

### **Impact of schema design**

Table 3 also shows the schema modification times. MySQL's schema modification time varied, peaking at 129ms before stabilizing. MongoDB's time decreased steadily, reaching 168.85ms in Run 5.

Both databases handled nested data structures effectively. MySQL processed 13 queries per transaction, indicating consistent performance. MongoDB processed an average of 29 queries, suggesting better efficiency with complex nested structures.

**Table 4.3 Test 2 Run Comparison**



### **Performance patterns**

There were differences in the operational characteristics, as illustrated in Figure 5. MySQL demonstrated varying total execution times, with its best performance at 382ms, a mean of 518ms, and reaching a peak of 777ms in its worst case. In terms of query processing, it followed a similar pattern but at a smaller scale, ranging from 36.25 to 89ms, with a mean of 60.46ms. The schema modification, showed consistent performance starting at 57ms, averaging 83.8ms, and peaking at 129 milliseconds.

Mongo on the other hand, demonstrated its best performance at 234ms, a mean of 305ms, and a peak of 450ms in its worst performance. For processing query, it ranged from 50.54ms to 102.64ms, with a mean of 63.952ms, and schema modification time ranged from 168.85ms to 263.75ms, with a mean of 205.705ms

A graph of different colored bars

Description automatically generated

**Figure 4.2. Performance Metrics Comparison Test 2**

## **Test 3**

### **Transaction behaviour**

The third test revealed an increased in timing patterns across five runs (Table 4). During the first run, MySQL completed all booking transactions in 3,232ms, while processing individual queries in approximately 33.53ms. The database needed 151ms to handle schema modifications. In comparison, Mongo completed much faster at 1.099 milliseconds but had a longer schema modification time of 307.68 milliseconds. Both databases achieved a 100% success rate in all booking attempts during this initial run.

As the test progressed, the total processing time remained steady at 3,232ms, through the time for individual queries showed a slight increase to 34.05ms.

### **Impact on schema design**

Initially, MySQL required 151ms for schema modifications in the first run, but this timing showed a gradual decrease through the test runs, eventually reaching 123ms in the final run. The improvements suggested the system became more efficient at handling structural changes over time.

Mongo showed different timing patterns, started with longer modification times of 307.68ms in the first run. These times decreased notably throughout the testing period, reaching 182.29ms by the fifth run. This suggested a learning curve in handling schema changes.

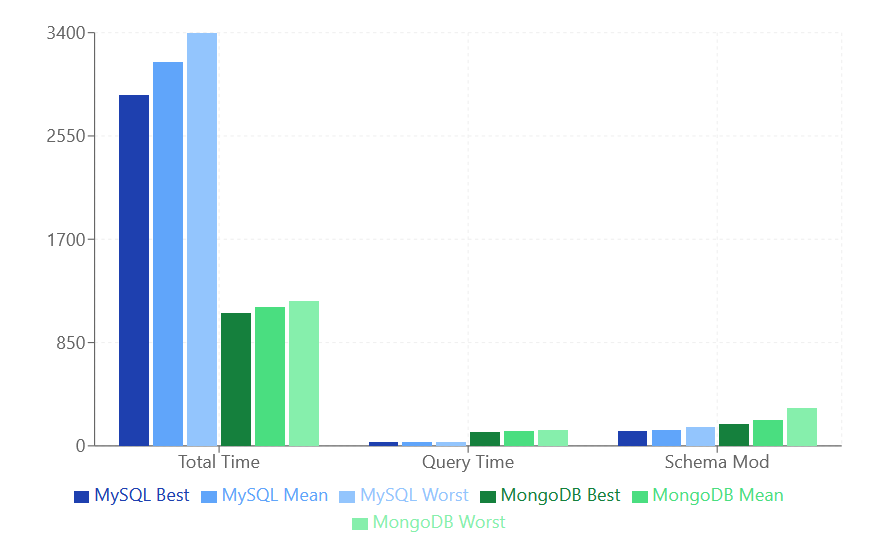
**Table 4.4. Test 3: Run Comparison**



### **Performance patterns**

Showed in the Figure 6, the patterns had interesting variations, in the case of MySQL the consistent performance across the test run, with and average total processing time of 3,159.8ms. The variation in these times remained small, showing a standard deviation of 195.9ms. Query processing times stayed stable, averaging 33.03ms with only small variations of 2.01 milliseconds.

Mongo showed strong stability as well, in total processing time, averaging 1,139.2ms across all runs. The variation in these times was 39.8ms. Query processing remains steady, averaging 118.8ms with a variation of 6.9ms. The most notable changes appeared in schema modification time, which averaged 214.22ms but showed larger variations of 51.9ms between runs.



**Figure 4.3. Performance Metrics Comparison Test 3**

## **Test 4**

### **Transaction behaviour**

During this test the system examined the attempt of 1000 bookings. During the initial run, the total processing time for MySQL reached 30,161ms, with individual queries taking 30.22ms to complete. Throughout subsequent runs, SQL showed stable performance. By the third run, the total time decreased to 25,195ms, with further improvements in the fourth and fifth, reaching 23,998ms in the final test. Query processing times followed this pattern of improvement, starting at 30.22ms and decreasing to 24.01ms by the final run. These results can be consulted in Table 5.

### **Impact on schema design**

MySQL initial schema modification time of 144ms showed continuous improvement throughout the testing period, reaching 102ms in the final run. This reduction in modification times suggested increasing efficiency in handling structural changes.

In contrast, MongoDB began at 243.45ms in Run 1 and varied through the runs, increasing to 284.09ms by Run 5, as detailed in Table 5. This suggest that Mongo had inconsistent performance in managing schema modifications over time.

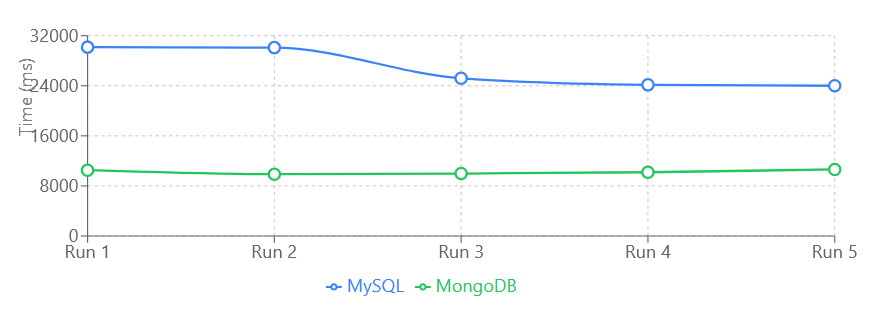
MySQL processed 1,003 queries per transaction across all test runs, indicating perfect stability with no variation. On the other hand, Mongo handled an average of approximately 2,996ms queries per transaction, with a slight variance of 0.5. The higher number of queries per transaction in Mongo suggest it was more efficient in managing complex nested structures within the system.

**Table 4.5. Test 4: Run Comparison**



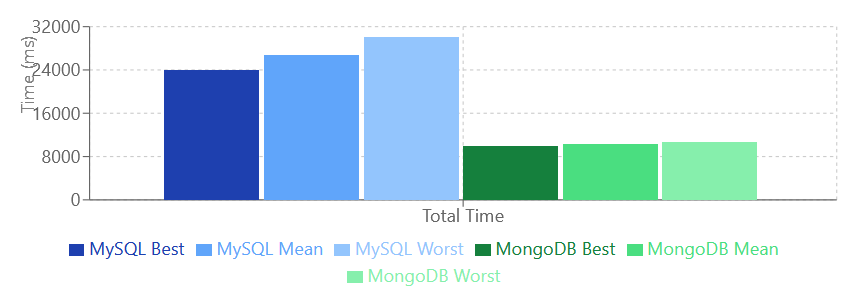
### **Performance patterns**

For this section the creation of more Figures was needed as it is a big gap between the Total time of execution and the Schema mod. Figure 7 reveals the decrease of time from the first test to the last one.



**Figure 4.4. Performance Evolution Over Time Test 4**

In Figure 8, revealed the improvement in total time, for MySQL, averaging 26,718ms. Query time averaging 26.80ms and schema modification averaging 126ms.



**Figure 4.5. Total Time Performance Test 4**

Figure 9 shows the comparison for both databases. Mongo maintained highly stable total time, averaging 10,234ms. Query time averaging 138.38ms. However, schema modification time had a higher variability, averaging 232.19ms with a standard deviation of 46.8ms, indicating more fluctuation.

A graph of different colored bars

Description automatically generated

**Figure 4.6. Query and schema modification performance**

## **Test 5**

### **Transaction behaviour**

Test 5 explored how Mongo and MySQL handled 5000 booking attempts. MySQL improved in handling transactions. Initially, in Run 1, processed in 199,285ms with an average query time of 39.82ms (Table 6). By Test 5, the total time had decreased to 251,290ms, and the query time improved to 50.22ms. This consistency indicated that MySQL became more efficient in manging transactions as the test progressed, achieving its best performance in the final run.

In contrast, Mongo exhibited a stable yet variable performance. In Run 1, Mongo completed the transaction in 75,916ms with a query time of 203.84ms. While the total time remained stable, fluctuating between 73,431 and 101,200ms across the runs, there was a noticeable increase in query times, reaching up to 266.09ms by Run 4.

### **Impact on schema design**

Such as the previous test, schema modification time showed a clear improving trend for MySQL, decreasing from 543ms in Run 1 to 63ms in Run 5. This reduction indicates that MySQL became more efficient in handling schema changes over the course of the tests.

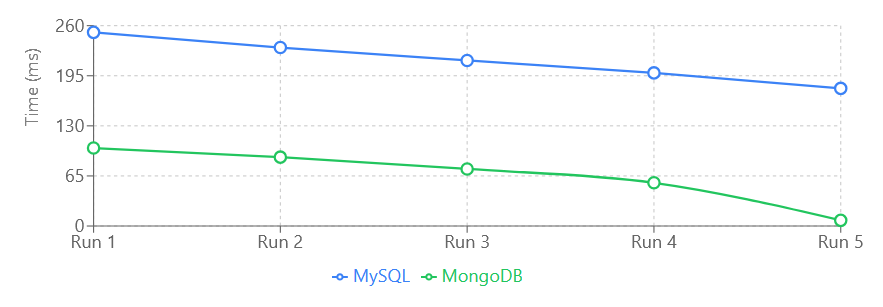
On the other hand, MongoDB’s schema modification times were more variable and increased over the test runs. Starting at 501.05ms in Run 1, schema modification times fluctuated and rose the 831.26ms by Run 5. This inconsistency suggested that Mongo faced challenges in maintaining efficient schema modifications under concurrent transaction loads, impacting its overall performance.

**Table 4.6. Test 5: Run Comparison**

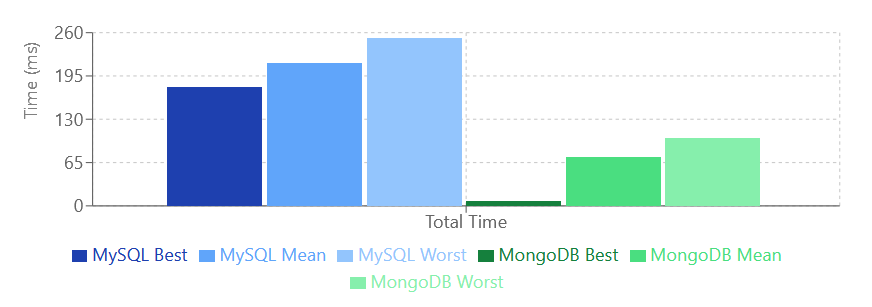


### **Performance patterns**

MySQL average for total time was 214,791ms, with a standard deviation of 28,904ms. Query time average 42.02ms with a standard deviation of 6.07ms. And the schema modification time demonstrated an improving trend, averaging 166.8ms

**Figure 4.7. Performance Evolution Over Time Test 5**

The total execution time decreased from 199,285ms in Test Run 1 to 251,290ms in Test Run 5, while query time improved from 39.82ms to 50.22ms. These consistent improvements suggest that MySQL became more efficient in handling transactions as the tests progressed. Figures 11 and 12 show these results.



**Figure 4.8.Total Time Performance Test 5**

MongoDB performance was highly variable. With it initially completed transactions faster than MySQL, its query times increased significantly over the test runs, and schema modification times showed a fluctuating and upward trend. Despite maintaining lower total execution times compared to MySQL, the increased variability in query and schema modification times indicates challenges in performance under prolonged concurrent transaction loads.

A graph of different colored bars

Description automatically generated

**Figure 4.9. Query and Schema Modification Performance**

# **Discussion**

## **Restating the Research Problems and Questions**

Throughout this study it has been clear that choose the right database. This proposal focused on comparing MySQL and MongoDB, whose goal was to see how each handles transactions, schema design, and complex data structures in a ticketing system with many users trying to book tickets at the same time. The research looked to answer three key questions:

1. How does MySQL ensure transaction integrity in a high-concurrency ticketing system, and what challenges does its rigid schema present when dealing with complex relationships?
2. How does MongoDB handle transaction in a scenario with many simultaneous purchases, and how does it flexible schema affect the modelling of intricate data structure?
3. What are the main differences in implementing transactional operations and data modelling between MySQL and MongoDB in the context of the ticketing system?

Additionally, the study teste three hypotheses related to how each database handles multiple transaction at once, how fast they perform queries, and how flexible heir schemas impact performance.

## **Summary of Findings**

The five experiments conducted showed how each database handle transaction, schema modification, and perform under various levels of user concurrency. Across five tests, each increasing in the number of concurrent booking attempts MySQL and MongoDB showed their strengths and weaknesses.

### **Transactional behaviour**

Both approaches followed the transaction flow find in the Appendix 12. MySQL and MongoDB proved capable of handling all booking transactions successfully, event as the number of users surged from 1 to 5000. MySQL stood out by consistently maintaining strong transactional integrity. It managed to keep query times low in the initial tests, which aligns with Capris et al. (2023), who emphasized MySQL’s reliability in maintaining data consistency through its ACID compliance.

MongoDB initially showed satisfactory performance in the first test with just a single booking attempt. However, as the number of concurrent transactions increased, performance became more unpredictable. This observation supports the hypothesis that flexible schema can impact performance negatively under high concurrency. Győrödi et al. (2021) also noted that while Mongo excels in write-heavy workloads, its performance advantage diminishes as transaction volumes grow, a pattern clearly reflected in the later tests within this study.

### **Schema design**

The differences between both databases created distinct performance characteristics that impact their behaviour under varying workloads. MySQL’s rigid schema architecture provided a robust foundation for maintaining data integrity through unforced relationships and constraints. This structured approached enabled efficient query optimization, particularly for joins and relationship-heavy queries, as demonstrated by Capris et al.’s (2023).

MySQL table locking mechanism during schema changes represented a deliberate trade-off between consistency and availability. The system demonstrated optimization behaviour, where successive schema modification become increasingly efficient as the database engine refined its execution paths. However, this came at the cost of temporary degradation during high-concurrency periods, particularly affecting concurrent operations that required access to modified tables.

MongoDB presented a clearly different approach to schema management due to its document-oriented architecture. Its flexible model operates at the document level and enabled independent evolution of data structures without requiring system-wide modifications. Park et al.’s (2021) research on contextual concurrency control (C3) revealed that this flexibility introduced subtle complexity in validation overhead. Their findings indicated that while MongoDB is better at schema adaptation, it faces challenged in maintaining consistent performance when multiple schema variations coexist under high concurrency.

These implications extended beyond simple modification metrics, MongoDB schema enabled rapid application development and iteration, but introduced additional computational overhead for schema validation during write operations. This created a performance pattern where the database exhibited good write performance for consistent document structures but experienced degradation when handling diverse schema variations simultaneously.

### **Performance**

The contrasting performance patterns between MySQL and MongoDB showed the differences in their architecture which influenced their behaviour under transaction loads. MySQL vertical scaling and ACID compliance ensured that transaction was processed reliably, maintaining data integrity without the significant performance fluctuations. This stability was discussed in Capris et al. (2023) noting that MySQL’s structured architecture allowed it to handle complex read operations efficiently, even as transaction loads increase. The gradual optimization of schema modification times over successive runs further demonstrates MySQL’s ability to adapt and maintain performance under increasing loads.

MongoDB in contrast, backed for its horizontal scaling approach, which emphasizes distribution and partitioning across multiple nodes, allowed it to manage larger volumes of data effectively. Mongo’s flexibility exceled at managed heavy workloads (Test 5) specifically in schema flexibility, being a suitable alternative when greater agility and scalability is needed and can tolerate temporary inconsistencies.

As a summary for this section, MySQL offered a better stability and reliability due to its vertical scaling and ACID compliance, whereas MongoDB provided flexibility and scalability through horizontal scaling and schema-less design, which are advantageous for write heavy applications and those requiring dynamic data structures. However, MongoDB’s became more variable under high concurrency due to the overhead of maintaining data consistency and managing schema modifications.

## **Implications**

Choosing the right database is a long and difficult decision. MySQL is valuable due to is strong compliance, particularly valuable for applications requiring strict data consistency, such as financial transactions or inventory management. During high-concurrency testing, MySQL maintained stable query times and efficient resource management, underscoring its suitability for data integrity environments.

On the flip side, MongoDB offer advantages for rapidly evolving applications where data structures frequently change. This flexibility was evident in the initial tests, where schema modifications were handled flawless without blocking other operations. However, as concurrency increased, the performance variability highlighted the need for robust schema management practices. While Mongo is ideal for agile development and handling diverse data formats, it requires careful optimization to maintain consistency and performance.

From the business perspective, can be said that MySQL’s predictable performance under increasing loads facilitates straightforward capacity planning, enabling more accurate resource provisioning and cost estimation. On the other hand, MongoDB’s ability to reduce development time and maintenance overhead might be helpful for organizations needing rapid iteration. However, the performance fluctuations under high concurrency necessitate more sophisticated monitoring and scaling strategies, potentially increasing operational complexity.

Teams using MySQL should invest in thorough upfront schema design and anticipate future data structure requirements to prevent costly modifications during peak operations. This approach could improve stability and reduce latency.

MongoDB implementations, emphasize the importance of robust data validation mechanisms at the application level. Given the flexibility, ensuring data consistency through clear conventions and version control of schema changes is important.

## **Limitations and Considerations**

The experiments were conducted using specific versions of MySQL and MongoDB on a controlled hardware setup. This environment may not reflect the diverse hardware configurations and network conditions encountered in real-world applications. For instance, different CPU architectures or varying network latencies could significantly influence database performance.

Although the study tested up to 5,000 concurrent users, actual ticketing systems may experience much higher peak loads, especially during major event sales. Additionally, each test was limited to a short duration on one minute, which may not capture long-term performance trends or potential degradation over extended periods.

Additionally, this project used standard booking operations which may not encompass the full complexity of real-world implementations. Since modern systems often include features like dynamic pricing, waitlist management, and intricate booking rules, which introduce additional layers of data complexity and transactional requirements. These functionalities could impact database performance differently.

Consequently, while the performance metrics such as transaction times and schema modification durations were measured. Other critical aspects like latency effects and detailed behaviour of database caching mechanism were not explored. Isolating the impact of these factors is challenging with the current setup.

In terms of external validity, the findings are specific to the ticketing system implementation and may not directly apply to other domains with different data structures and transactional requirements. Applications in fields such as healthcare, social media, or finance might need different performance characteristics.

# **Conclusion**

## **Answers to Research questions**

**Q1**: How does MySQL ensure transactional integrity in a high-concurrency ticketing system, and what challenges arise from its rigid schema when dealing with complex data relationships?

By adhering to ACID principles MySQL ensured that all transactions were processed reliably, preventing data inconsistencies during simultaneous ticket purchases this was evident in the consistent success rate and stable query times observed during high-concurrency tests. The schema structure was fundamental for the maintenance of consistency and relationship integrity, as it enforced strict data validation and referential integrity across multiple tables. However, this rigidity also introduced challenges. Schema modifications required table-level locking, which could temporarily impede other operations and introduce latency during high-load periods. This trade-off between data integrity and flexibility highlights the importance of strict schema design and management deployments.

**Q2:** How does MongoDB handle transactions in a concurrent purchase scenario, and how does its flexible schema influence the modelling of intricate data structures?

Mongo displayed a different approach to transaction management, supporting its architecture to handle transaction correctly. Initially, exhibited superior query performance due to its flexible schema that helped for rapid adaptations to evolving embedded data, and modelling relationships without the overhead of maintaining complex table joins. However, as concurrency levels intensified, performance became more variable. Schema modifications at the document level were faster and non-blocking, yet this agility sometimes led to inconsistencies and longer modification times as multiple concurrent schema changes occurred.

**Q3:** What are the key differences in implementing transactional operations and data modelling between MySQL and MongoDB in the context of the ticketing system?

The main difference is that MySQL employs a rigid schema design that demands comprehensive upfront planning. This structured methodology ensures that data relationships are meticulously defined and enforced through transactional controls. As a result, delivers consistent performance and robust data integrity, however this approach can hinder development flexibility, as adapting to evolving data requirements often necessitates complex and time-consuming schema modifications.

MongoDB, on the other hand, leverages a schema-less architecture, granting developers the freedom to modify data structures dynamically without extensive alterations to the database schema. However, the flexibility of MongoDB introduces challenges in maintaining transactional consistency, unlike MySQL enforced data integrity, Mongo relies on application-level mechanisms to ensure consistency across diverse data structures. This necessitates more sophisticated validation and monitoring strategies to prevent data anomalies and ensure reliable transaction outcomes. Additionally, MongoDB’s horizontal scaling, which distributes data across multiple nodes, enhances its ability to oversee large volumes of transactions and data. Nonetheless, this distributed nature complicates the maintenance of performance consistency and data integrity, as it requires effective coordination and synchronization across nodes to prevent inconsistencies.

These implementation differences highlight a fundamental trade-off between structure and flexibility. MySQL’s disciplined approach provides stability and reliability but at the cost of reduced agility, whereas MongoDB’s adaptable framework offers scalability and ease of evolution, albeit with increased complexity in transaction management.

## **Contributions**

The findings contributed to the theoretical framework surrounding database selection criteria, emphasizing the importance of aligning database characteristics with specific application requirements. This approach provided empirical evidence supporting existing theories about trade-offs between schema flexibility and transactional consistency.

In the methodology part, the study introduced a systematic technique comparing database performance under varying concurrent loads. The experimental design, incorporating progressive increases in user concurrency offered a replicable framework for evaluating database performance in similar contexts.

The findings yielded several implications for database selection and system design:

1. Enhanced understanding of the performance characteristics of MySQL and MongoDB under high concurrency conditions
2. Detailed insights into the impact of schema design choices on system scalability
3. Practical guidelines for implementing transaction management in both database systems
4. Strategic considerations for database selection based on specific application requirements

## **Recommendations and future research**

Selecting the appropriate database should be driven by an organization’s unique operational requirements. For environments where strict data consistency and predictable performance are important, MySQL emerges as the preferable choice. For applications that demand high throughput and the ability to scale rapidly with increasing concurrent user loads, MongoDB is the way to choose. However, much of this depends on the evaluation of their workload characteristics- whether they are rad-heavy, write-heavy, or balanced- to determine which database system aligns best with their performance and scalability needs.

Another consideration is the schema management strategy, if the organization has decided to implement MySQL as the database, its implementation must be careful planning, since modifications could be needed through the time. Another observation is prioritizing schema changes during periods of low system activity to minimize disruption. Implementing online schema changes, where feasible, can help maintain system availability without compromising performance.

In contrast, MongoDB benefit from continuous monitoring and documentation of schema versions. By maintaining comprehensive records of changes and employing automated validation processes, organizations can prevent data inconsistencies and ensure that the database remains resilient against the complexities introduced by frequent schema modifications.

As a general direction for further research, examining distributed database configurations represents a promising avenue, exploring areas like:

1. **Comparative analysis of replication strategies**: Investigating how different replication methods impact system performance and data consistency can provide insights into optimizing distributed deployments
2. **Integration patterns for combining MySQL and MongoDB:** Exploring effective methods for integrating these two databases within single applications can enhance performance and flexibility
3. **Cost-benefit analysis of maintaining multiple database systems:** evaluating the operational and financial implications of managing hybrid database environments
4. **Schema design optimization for specific use cases:** Developing best practices for schema design that maximize performance and scalability based on application-specific requirements
5. **Evaluate the difference between Object Document Mapper and Object Relational Mapping:** as this study used JPA Hibernate and Morphia, future research could investigate in deep both applications within different experiments and workloads

The research underscored the importance of aligning database selection with specific workload characteristics rather than relying solely on overall performance metrics. Organizations should conduct through workload analyses to determine whether their applications are read-heavy, write-heavy, or require a balanced mix of both. This understanding will guide the selection of the most suitable database system.

Moreover, the study highlighted the necessity of ongoing database optimization and maintenance. Proper tuning-whether through indexing strategies for MySQL or document structure optimization for MongoDB- is essential to sustain performance as application demands evolve.

# **References**

Amghar, S., Cherdal, S., & Mouline, S. (2020). Data Integration and NoSQL Systems: A State of the Art. *Proceedings of the 4th International Conference on Big Data and Internet of Things*, 1–6. https://doi.org/10.1145/3372938.3372954

Amghar, S., Cherdal, S., & Mouline, S. (2022). Storing, preprocessing and analyzing tweets: Finding the suitable noSQL system. *International Journal of Computers and Applications*, *44*(6), 586–595. https://doi.org/10.1080/1206212X.2020.1846946

Bernstein, P. A., & Goodman, N. (1981). Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.*, *13*(2), 185–221. https://doi.org/10.1145/356842.356846

Bernstein, P. A., & Newcomer, E. (2009). *Principles of transaction processing* (2nd ed). Morgan Kaufmann Publishers.

Besleaga, C. (2016, October 6). *MySQL :: MySQL 8.0 Labs: JSON aggregation functions*. MySQL. https://dev.mysql.com/blog-archive/mysql-8-0-labs-json-aggregation-functions/

*Best practices for modeling relational data in DynamoDB - Amazon DynamoDB*. (n.d.). Retrieved December 4, 2024, from https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-relational-modeling.html

Brewer, E. (2012). CAP twelve years later: How the “rules” have changed. *Computer*, *45*(2), 23–29. Computer. https://doi.org/10.1109/MC.2012.37

Cai, S., Gallina, B., Nyström, D., & Seceleanu, C. (2019). Data aggregation processes: A survey, a taxonomy, and design guidelines. *Computing*, *101*(10), 1397–1429. https://doi.org/10.1007/s00607-018-0679-5

Candel, C. J. F., Sevilla Ruiz, D., & García-Molina, J. J. (2022). A unified metamodel for NoSQL and relational databases. *Information Systems*, *104*, 101898. https://doi.org/10.1016/j.is.2021.101898

Capris, T., Melo, P., Garcia, N. M., Pires, I. M., & Zdravevski, E. (2022). Comparison of SQL and NoSQL databases with different workloads: MongoDB vs MySQL evaluation. *2022 International Conference on Data Analytics for Business and Industry (ICDABI)*, 214–218. https://doi.org/10.1109/ICDABI56818.2022.10041513

Carvalho, I., Sá, F., & Bernardino, J. (2023). Performance Evaluation of NoSQL Document Databases: Couchbase, CouchDB, and MongoDB. *Algorithms*, *16*(2), Article 2. https://doi.org/10.3390/a16020078

Chang, M.-L. E., & Chua, H. N. (2019). SQL and NoSQL Database Comparison. In K. Arai, S. Kapoor, & R. Bhatia (Eds.), *Advances in Information and Communication Networks* (Vol. 886, pp. 294–310). Springer International Publishing. https://doi.org/10.1007/978-3-030-03402-3\_20

Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking cloud serving systems with YCSB. *Proceedings of the 1st ACM Symposium on Cloud Computing*, 143–154. https://doi.org/10.1145/1807128.1807152

Diaz-Ordoñez, M., Rodríguez Baena, D. S., & Yun-Casalilla, B. (2023). A new approach for the construction of historical databases—NoSQL Document-oriented databases: The example of AtlantoCracies. *Digital Scholarship in the Humanities*, *38*(3), 1014–1032. https://doi.org/10.1093/llc/fqad033

Digittrix, N. (2023, September 18). The Battle of Databases: MySQL vs. MongoDB. *Medium*. https://medium.com/@nikita.digittrix/the-battle-of-databases-mysql-vs-mongodb-ca57164b8386

Faraj, H. A. (2022). Moving RDBMS to NoSQL Paradigms. *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*, 684–689. https://doi.org/10.1109/CSCI58124.2022.00125

Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Wesley. https://dl.ebooksworld.ir/motoman/Patterns%20of%20Enterprise%20Application%20Architecture.pdf

Gallinucci, E., Golfarelli, M., & Rizzi, S. (2018). Schema profiling of document-oriented databases. *Information Systems*, *75*, 13–25. https://doi.org/10.1016/j.is.2018.02.007

Gray, J., & Lamport, L. (2006). Consensus on transaction commit. *ACM Trans. Database Syst.*, *31*(1), 133–160. https://doi.org/10.1145/1132863.1132867

Győrödi, C. A., Dumşe-Burescu, D. V., Győrödi, R. Ş., 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

Győrödi, C., Győrödi, R., Pecherle, G., & Olah, A. (2015). A comparative study: MongoDB vs. MySQL. *2015 13th International Conference on Engineering of Modern Electric Systems (EMES)*, 1–6. https://doi.org/10.1109/EMES.2015.7158433

Harding, R., Van Aken, D., Pavlo, A., & Stonebraker, M. (2017). An evaluation of distributed concurrency control. *Proc. VLDB Endow.*, *10*(5), 553–564. https://doi.org/10.14778/3055540.3055548

Hecht, R., & Jablonski, S. (2011). NoSQL evaluation: A use case oriented survey. *2011 International Conference on Cloud and Service Computing*, 336–341. https://doi.org/10.1109/CSC.2011.6138544

Imam, A. A., Basri, S., Ahmad, R., Watada, J., Gonzlez-Aparicio, M. T., & Almomani, M. A. (2018). Data Modeling Guidelines for NoSQL Document-Store Databases. *International Journal of Advanced Computer Science and Applications (Ijacsa)*, *9*(10), Article 10. https://doi.org/10.14569/IJACSA.2018.091066

Ingeno, J. (2018). *Software Architect’s Handbook: Become a Successful Software Architect by Implementing Effective Architecture Concepts*. Packt Publishing.

Kaltenecker, C., Mühlbauer, S., Grebhahn, A., Siegmund, N., & Apel, S. (2023). Performance evolution of configurable software systems: An empirical study. *Empirical Software Engineering*, *28*(6), 152. https://doi.org/10.1007/s10664-023-10338-3

King, E. (2024, March 1). *Hello Interview | System Design in a Hurry*. Hello Interview. https://www.hellointerview.com/learn/system-design/answer-keys/ticketmaster

Kleppmann, M. (2017). *Designing Data-Intensive Applications*. https://martin.kleppmann.com/2017/03/27/designing-data-intensive-applications.html

Kumar, C. (2019, July 19). *Introduction to Morphia—Java ODM for MongoDB | Baeldung*. Baeldung. https://www.baeldung.com/mongodb-morphia

Kuznetsov, S. D., & Poskonin, A. V. (2014). NoSQL data management systems. *Programming and Computer Software*, *40*(6), 323–332. https://doi.org/10.1134/S0361768814060152

Lenberg, P., Feldt, R., Gren, L., Wallgren Tengberg, L. G., Tidefors, I., & Graziotin, D. (2024). Qualitative software engineering research: Reflections and guidelines. *Journal of Software: Evolution and Process*, *36*(6), e2607. https://doi.org/10.1002/smr.2607

Li, C. (2010). Transforming relational database into HBase: A case study. *2010 IEEE International Conference on Software Engineering and Service Sciences*, 683–687. https://doi.org/10.1109/ICSESS.2010.5552465

Liang, J. T., Arab, M., Ko, M., Ko, A. J., & LaToza, T. D. (2023). A Qualitative Study on the Implementation Design Decisions of Developers. *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 435–447. https://doi.org/10.1109/ICSE48619.2023.00047

Lith, A., & Mattsson, J. (2010). *Investigating storage solutions for large data—A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data* [CHALMERS UNIVERSITY OF TECHNOLOGY]. https://www.semanticscholar.org/paper/Investigating-storage-solutions-for-large-data-A-of-Lith-Mattsson/b8f6b9d79c75e66c3b2f5034fe8172fd24cc0d13

Mok, W. Y. (2021). A Logical Database Design Methodology for MongoDB NoSQL Databases. *2021 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, 1451–1455. https://doi.org/10.1109/IEEM50564.2021.9673004

Möller, M. L., Scherzinger, S., Klettke, M., & Störl, U. (2020). Why It Is Time for Yet Another Schema Evolution Benchmark. In N. Herbaut & M. La Rosa (Eds.), *Advanced Information Systems Engineering* (pp. 113–125). Springer International Publishing. https://doi.org/10.1007/978-3-030-58135-0\_10

O’Grady, S. (2020, May). *MongoDB ACID Transactions Whitepaper*. MongoDB. https://www.mongodb.com/resources/products/capabilities/mongodb-multi-document-acid-transactions

Osman, R., & Knottenbelt, W. J. (2012). Database system performance evaluation models: A survey. *Performance Evaluation*, *69*(10), 471–493. https://doi.org/10.1016/j.peva.2012.05.006

Patil, M. M., Hanni, A., Tejeshwar, C. H., & Patil, P. (2017). A qualitative analysis of the performance of MongoDB vs MySQL database based on insertion and retriewal operations using a web/android application to explore load balancing—Sharding in MongoDB and its advantages. *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC)*, 325–330. https://doi.org/10.1109/I-SMAC.2017.8058365

Reinero, B. (2017, April 27). *Transitioning from Relational Databases to MongoDB - Data Models | MongoDB Blog*. MongoDB. https://www.mongodb.com/blog/post/transitioning-from-relational-databases-to-mongodb

Reisner, P. (1988). Query Languages1. In M. Helander (Ed.), *Handbook of Human-Computer Interaction* (pp. 257–280). North-Holland. https://doi.org/10.1016/B978-0-444-70536-5.50017-8

Sadalage, P. J., & Fowler, M. (2012). *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence* (1st ed.). Addison-Wesley Professional.

*Schema Validation—MongoDB Manual v8.0*. (n.d.). Retrieved December 4, 2024, from https://www.mongodb.com/docs/current/core/schema-validation/

Seghier, N. B., & Kazar, O. (2021). Performance Benchmarking and Comparison of NoSQL Databases: Redis vs MongoDB vs Cassandra Using YCSB Tool. *2021 International Conference on Recent Advances in Mathematics and Informatics (ICRAMI)*, 1–6. https://doi.org/10.1109/ICRAMI52622.2021.9585956

Sen, P. S., & Mukherjee, N. (2024). An ontology-based approach to designing a NoSQL database for semi-structured and unstructured health data. *Cluster Computing*, *27*(1), 959–976. https://doi.org/10.1007/s10586-023-03995-y

Seppälä, I. (2024). *Java Design Patterns* [Java]. https://github.com/iluwatar/java-design-patterns (Original work published 2014)

Shareef, T. H., Sharif, K. H., & Rashid, B. N. (2022). A Survey of Comparison Different Cloud Database Performance: SQL and NoSQL. *Passer Journal of Basic and Applied Sciences*, *4*(1), 45–57. https://doi.org/10.24271/psr.2022.301247.1104

Stanescu, L., Brezovan, M., & Burdescu, D. D. (2016). Automatic Mapping of MySQL Databases to NoSQL MongoDB. *Annals of Computer Science and Information Systems*, *8*, 837–840. https://doi.org/10.15439/2016F45

Stonebraker, M. (2010). SQL databases v. NoSQL databases. *Commun. ACM*, *53*(4), 10–11. https://doi.org/10.1145/1721654.1721659

Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., & Helland, P. (2007). The end of an architectural era: (It’s time for a complete rewrite). *Proceedings of the 33rd International Conference on Very Large Data Bases*, 1150–1160.

Stonebraker, M., & Pavlo, A. (2024). What Goes Around Comes Around... And Around... *SIGMOD*, *53*, 21–37. https://doi.org/10.1145/3685980.3685984

Sudiartha, I. K. G., Indrayana, I. N. E., Suasnawa, I. W., Asri, S. A., & Sunu, P. W. (2020). Data Structure Comparison Between MySql Relational Database and Firebase Database NoSql on Mobile Based Tourist Tracking Application. *Journal of Physics: Conference Series*, *1569*(3), 032092. https://doi.org/10.1088/1742-6596/1569/3/032092

Šušter, I., & Ranisavljević, T. (2023). OPTIMIZATION OF MYSQL DATABASE. *Journal of Process Management and New Technologies*, *11*(1–2). https://doi.org/10.5937/jpmnt11-44471

Thakare, A., Tembhurne, O. W., Thakare, A. R., & Reddy, S. N. (2023). NoSQL Databases: Modern Data Systems for Big Data Analytics - Features, Categorization and Comparison. *International Journal of Electrical and Computer Engineering Systems*, *14*(2), Article 2. https://doi.org/10.32985/ijeces.14.2.10

Tudorica, B. G., & Bucur, C. (2011). A comparison between several NoSQL databases with comments and notes. *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, 1–5. https://doi.org/10.1109/RoEduNet.2011.5993686

Van Der Loo, M. P. J., & De Jonge, E. (2020). Data Validation. In R. S. Kenett, N. T. Longford, W. W. Piegorsch, & F. Ruggeri (Eds.), *Wiley StatsRef: Statistics Reference Online* (1st ed., pp. 1–7). Wiley. https://doi.org/10.1002/9781118445112.stat08255

Vathy-Fogarassy, Á., & Hugyák, T. (2017). Uniform data access platform for SQL and NoSQL database systems. *Information Systems*, *69*, 93–105. https://doi.org/10.1016/j.is.2017.04.002

Wodyk, R., & Skublewska-Paszkowska, M. (2020). Performance comparison of relational databases SQL Server, MySQL and PostgreSQL using a web application and the Laravel framework. *Journal of Computer Sciences Institute*, *17*, 358–364. https://doi.org/10.35784/jcsi.2279

Xia, Y., Yu, X., Butrovich, M., Pavlo, A., & Devadas, S. (2022). Litmus: Towards a Practical Database Management System with Verifiable ACID Properties and Transaction Correctness. *Proceedings of the 2022 International Conference on Management of Data*, 1478–1492. https://doi.org/10.1145/3514221.3517851

Yang, X., Procopiuc, C. M., & Srivastava, D. (2009). Summarizing relational databases. *Proc. VLDB Endow.*, *2*(1), 634–645. https://doi.org/10.14778/1687627.1687699

Yedilkhan, D., Mukasheva, A., Bissengaliyeva, D., & Suynullayev, Y. (2023). Performance Analysis of Scaling NoSQL vs SQL: A Comparative Study of MongoDB, Cassandra, and PostgreSQL. *2023 IEEE International Conference on Smart Information Systems and Technologies (SIST)*, 479–483. https://doi.org/10.1109/SIST58284.2023.10223568

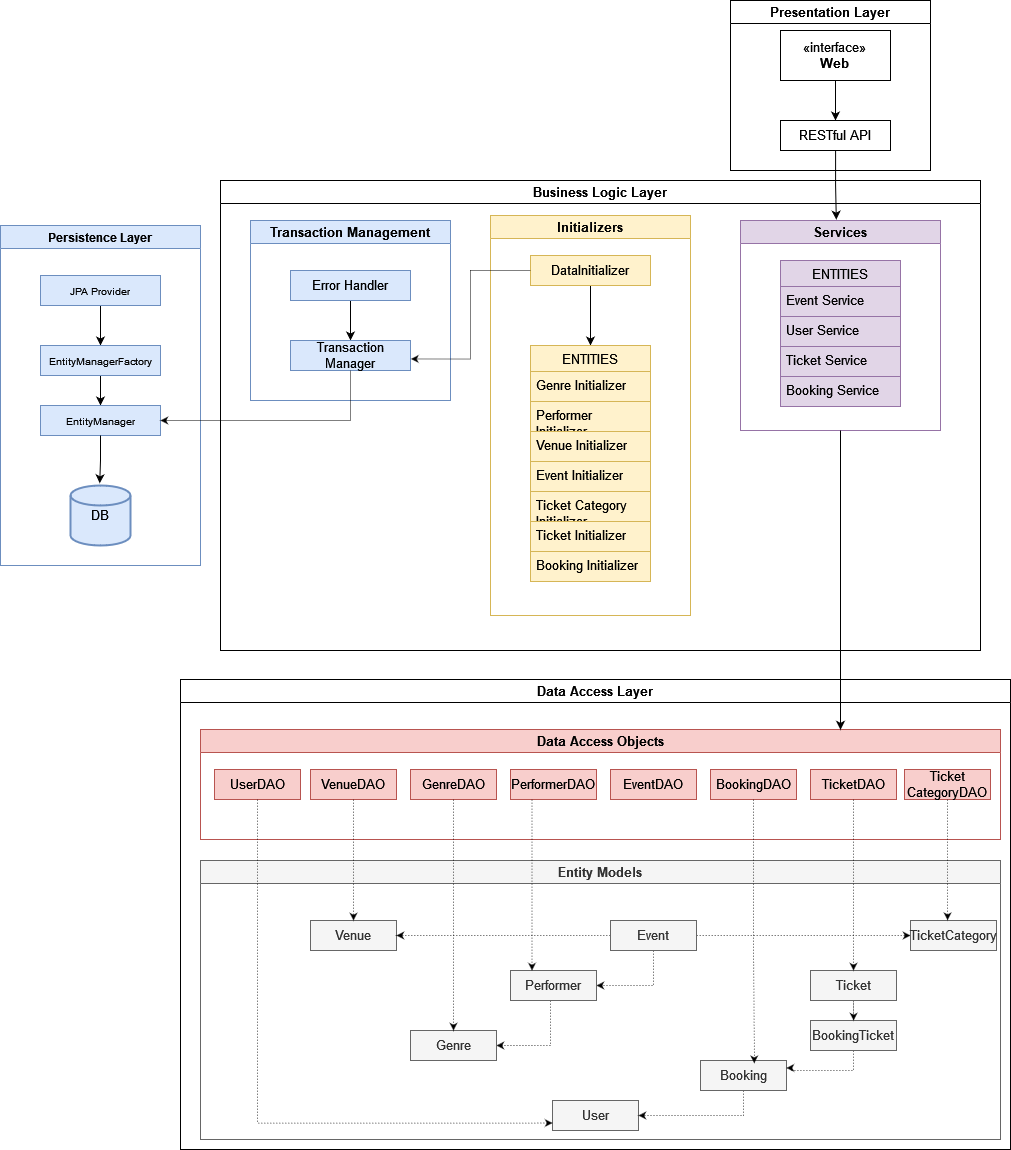
Yu, X., Bezerra, G., Pavlo, A., Devadas, S., & Stonebraker, M. (2014). Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, *8*(3), 209–220. https://doi.org/10.14778/2735508.2735511

Zhao, X., Pi, D., & Chen, J. (2020). Novel trajectory privacy-preserving method based on prefix tree using differential privacy. *Knowledge-Based Systems*, *198*, 105940. https://doi.org/10.1016/j.knosys.2020.105940

Ziegler, P., & Dittrich, K. R. (2004). Three Decades of Data Intecration—All Problems Solved? In R. Jacquart (Ed.), *Building the Information Society* (pp. 3–12). Springer US. https://doi.org/10.1007/978-1-4020-8157-6\_1

# **Appendices**

## **A MySQL System Architecture**



## **B Data Initializer**

*public DataInitializer() {*

*// Create an EntityManagerFactory based on the persistence unit*

*emf = Persistence.createEntityManagerFactory(PERSISTENCE\_UNIT\_NAME);*

*// Create an EntityManager to manage entities*

*em = emf.createEntityManager();*

*// Initialize DAOs with the EntityManager*

*genreDAO = new GenreDAO(em);*

*performerDAO = new PerformerDAO(em);*

*venueDAO = new VenueDAO(em);*

*more entities…*

*// Initialize initializers*

*initializers = new ArrayList<**>();*

*// Initialize initializers*

*List<String> genreNames = new ArrayList<**>();*

*initializers.add(new GenreInitializer(genreDAO, genreNames));*

*initializers.add(new PerformerInitializer(performerDAO, genreDAO));*

*rest of entities…*

*}*

*// populates the database*

*public void populateData() {*

*try {*

*for (Initializer initializer : initializers) { em.getTransaction().begin();*

*initializer.initialize(); em.getTransaction().commit()**; }*

*validateData();*

*} catch (Exception e) { //Rollback transaction in case of any errors*

*if*

*(em.getTransaction().isActive()) { em.getTransaction().rollback();*

*System.out.println(“Transaction rolled back due to an error.”)**; } // Print stack trace for debugging*

*e.printStackTrace(); }*

*}*

## **C MySQL Class Diagram**

A black background with white text

Description automatically generated

## **D Booking Service class**

*public class BookingService {*

*// Essential fields*

*private final EntityManager em;*

*// Metrics tracking*

*private final AtomicInteger successfulBookings = new AtomicInteger(0);*

*private final AtomicInteger failedBookings = new AtomicInteger(0);*

*private final AtomicInteger totalTicketsBooked = new AtomicInteger(0);*

*private long totalQueryTime = 0;*

*private int totalQueries = 0;*

*public BookingService(EntityManager em) {*

*this.em = em;*

*verifyDatabaseConnection();*

*}*

*private void verifyDatabaseConnection() {*

*try {*

*em.createNativeQuery(“SELECT 1”).getSingleResult();*

*System.out.println(“Database connection verified in BookingService.”);*

*} catch (Exception e) {*

*throw new RuntimeException(“Failed to verify database connection**”, e);*

*}*

*}*

*public List<String> getAvailableTicketSerials(int ventide) {*

*long startTime = System.nanoTime();*

*try {*

*return em.createQuery(*

*“SELECT t.serialNumber FROM Ticket t* *“ +*

*“WHERE t.event.eventId = :ventide AND t.status* *= :status”, String.class)*

*.setParameter(“ventide**”, ventide)*

*.setParameter(“status”, TicketStatus.AVAILABLE)*

*.getResultList();*

*} catch (Exception e) {*

*System.err.println(“Error getting available tickets:* *“ + e.getMessage());*

*return new ArrayList<**>();*

*} finally {*

*recordQueryTime(startTime);*

*}*

*}*

## **E Simulation Class Metrics – MySQL**

*public void printSimulationResults(int ventide) {*

*System.out.println(“\n=== Simulation Results ===”);*

*// Configuration metrics*

*System.out.printf(“Concurrent Users: %d%n”, NUM\_USERS);*

*System.out.printf(“Thread Pool Size: %d%n”, THREAD\_POOL\_SIZE);*

*// Performance metrics*

*long duration = (simulationEndTime – simulationStartTime) / 1\_000\_000;*

*System.out.printf(“Total Simulation Time: %d ms%n”, duration);*

*System.out.printf(“Average Query Time: %.2f ms%n”,*

*bookingService.getAverageQueryTime());*

*// Success metrics*

*System.out.printf(“Successful Bookings: %d%n**”,*

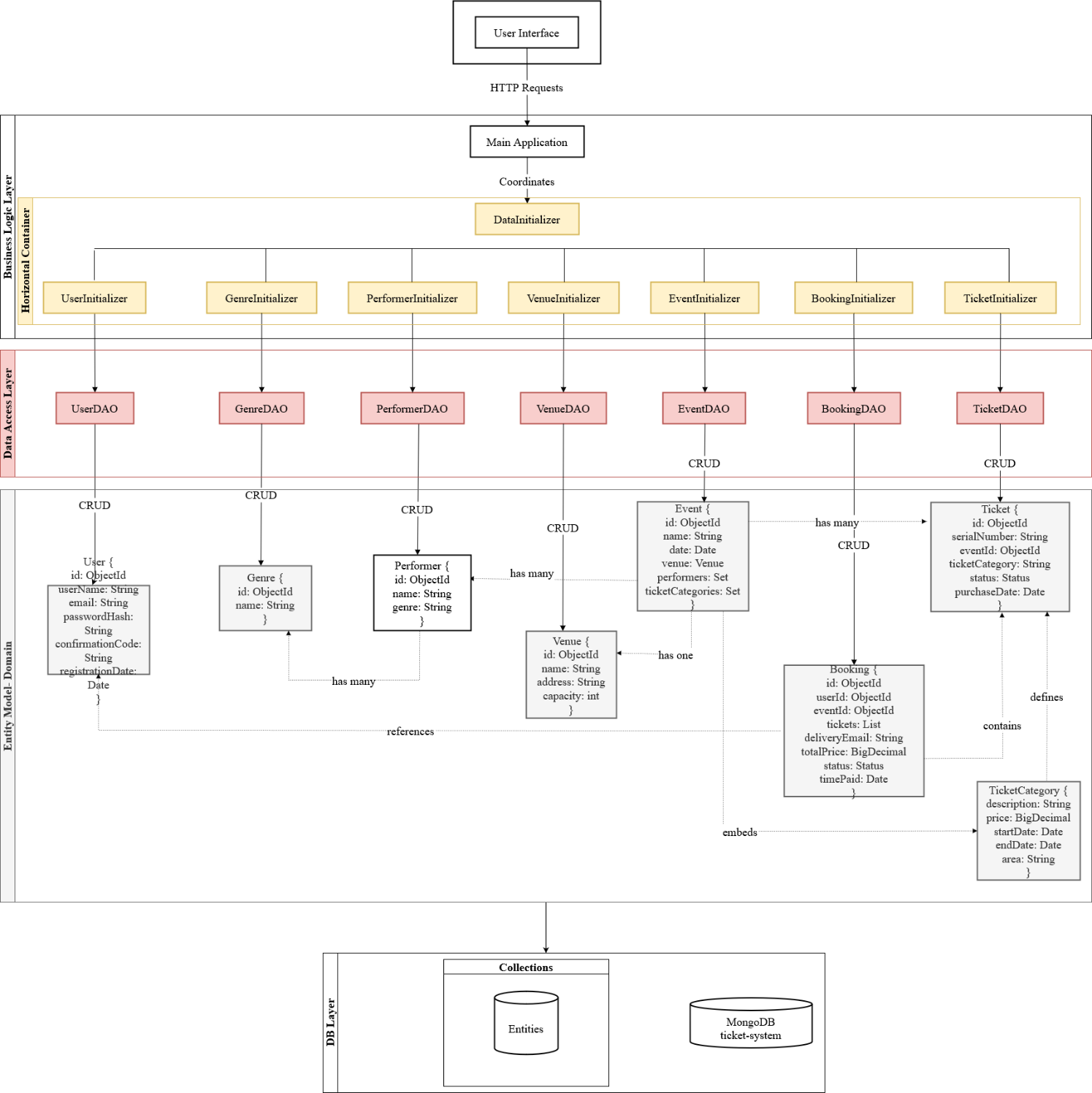
*successfulBookings.get());*

*System.out.printf(“Failed Bookings: %d%n”,*

*failedBookings.get());*

*}*

## **F MongoDB System Architecture**



## **G Document Structure Diagram for MongoDB**

{

“collMod”: “bookings”,

“validator”: {

“$jsonSchema”: {

“bsonType”: “object”,

“required”: [“user\_id”, “delivery\_email”, “delivery\_time”, “time\_paid”, “time\_sent”, “total\_price”, “discount”, “final\_price”, “status”, “tickets”],

“additionalProperties”: false,

“properties”: {

“\_id”: {},

“user\_id”: {

“bsonType”: “objectId”,

“description”: “User ID is required and must reference the users collection”

},

“delivery\_email”: {

“bsonType”: “string”,

“pattern”: “^.+@.+\\..+$”,

“description”: “Delivery email is required and must follow email format”

},

“delivery\_time”: {

“bsonType”: “date”,

“description”: “Delivery time is required”

},

“time\_paid”: {

“bsonType”: “date”,

“description”: “Time paid is required”

},

“time\_sent”: {

“bsonType”: “date”,

“description”: “Time sent is required”

},

“total\_price”: {

“bsonType”: “decimal”,

“minimum”: 0,

“description”: “Total price is required and must be a non-negative decimal”

},

“discount”: {

“bsonType”: “decimal”,

“minimum”: 0,

“description”: “Discount is required and must be a non-negative decimal”

},

“final\_price”: {

“bsonType”: “decimal”,

“minimum”: 0,

“description”: “Final price is required and must be a non-negative decimal”

},

“status”: {

“enum”: [“confirmed”, “in-progress”, “canceled”],

“description”: “Booking status is required and must be one of the specified values”

},

“tickets”: {

“bsonType”: “array”,

“description”: “Array of ticket references”,

“items”: {

“bsonType”: “objectId”,

“description”: “Each ticket must reference the tickets collection”

},

“uniqueItems”: true

}

}

}

},

“validationLevel”: “strict”,

“validationAction”: “error”

}

## **H MongoDB Application Schema**

A diagram of a company

Description automatically generated

## **I MongoDB Class Diagram**

A diagram of a function

Description automatically generated

## **J Booking ticket method**

public ventid bookTickets(ObjectId userId, ObjectId ventide, int quantity) {

try (ClientSession session = datastore.startSession()) {

session.startTransaction();

try {

Event event = findAndValidateEvent(ventide);

List<Ticket> bookedTickets = bookAvailableTickets(session, ventide, quantity);

Booking booking = createBooking(userId, ventide, calculateTotalPrice(event, bookedTickets), bookedTickets);

persistBooking(booking);

session.commitTransaction();

updateMetrics(bookedTickets.size());

return true;

} catch (Exception e) {

handleTransactionError(session, userId, e);

return false;

}

}

}

## **K Booking Simulation Class**

public class BookingSimulation {

// Configuration Constants

private static final int NUM\_USERS = 1000;

private static final int MAX\_TICKETS\_PER\_USER = 1;

private static final int NUMBER\_OF\_CORES = Runtime.getRuntime().availableProcessors();

private static final int THREAD\_POOL\_SIZE = NUMBER\_OF\_CORES \* 2;

private static final int SIMULATION\_TIMEOUT\_MINUTES = 3;

// Simulation components

private final BookingService bookingService;

private final UserDAO userDAO;

private final EventDAO eventDAO;

private final TicketDAO ticketDAO;

private final ExecutorService executorService;

// Metrics

private long simulationStartTime;

private long simulationEndTime;

private final AtomicInteger successfulBookings = new AtomicInteger(0);

private final AtomicInteger failedBookings = new AtomicInteger(0);

private int initialTicketCount;

private Event event;

}

## **L Transactional processes**

A white paper with black text

Description automatically generated