Blue text on a white background

Description automatically generated

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

**Gabriel Alberto Avina Solares**

Being a thesis presented for the award of

*Masters 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**

**ACKNOWLEDGEMENTS**

**To Silvia, Karla, and Andrea**

**To my friends and supervisor**

**A mis abuelos sláinte mhaith…**

Table of Contents

[**LIST OF TABLES AND FIGURES** 1](#_Toc183731514)

[**ABBREVIATIONS** 2](#_Toc183731515)

[**INTRODUCTION** 3](#_Toc183731516)

[Background and context 3](#_Toc183731517)

[Problem statement 3](#_Toc183731518)

[Research objectives 4](#_Toc183731519)

[Research questions 4](#_Toc183731520)

[Research hypotheses 5](#_Toc183731521)

[Scope and limitations 5](#_Toc183731522)

[Significance of the Study 5](#_Toc183731523)

[1 Review of Relevant Literature 6](#_Toc183731524)

[1.1 Database management systems overview 6](#_Toc183731525)

[1.2 Relational Databases and MySQL 6](#_Toc183731526)

[1.2.1 Overview and key features 6](#_Toc183731527)

[1.2.2 MySQL 6](#_Toc183731528)

[1.3 NoSQL Databases and MongoDB 6](#_Toc183731529)

[1.3.1 Overview and key features 6](#_Toc183731530)

[1.3.2 Document-oriented databases 9](#_Toc183731531)

[1.4 Transactional Processing and Concurrency 11](#_Toc183731532)

[1.5 Comparative Studies 11](#_Toc183731533)

[1.5.1 Benchmarking Cloud Serving Systems with YCSB (Cooper *et al.*, 2010) 13](#_Toc183731534)

[1.5.2 Databases vs NoSQL Databases (Stonebraker, 2010) 15](#_Toc183731535)

[2 Methodology 16](#_Toc183731536)

[2.1 Research design 16](#_Toc183731537)

[2.2 Experimental Framework 17](#_Toc183731538)

[2.2.1 Technical infrastructure 17](#_Toc183731539)

[2.3 Analysis Methods 17](#_Toc183731540)

[2.3.1 Quantitative 17](#_Toc183731541)

[2.3.2 Qualitative 19](#_Toc183731542)

[2.4 Research validation strategy 19](#_Toc183731543)

[2.4.1 Experimental validation 19](#_Toc183731544)

[2.4.2 Data validation 19](#_Toc183731545)

[2.4.3 Process validation 19](#_Toc183731546)

[2.5 Testing Methodology 19](#_Toc183731547)

[2.5.1 Performance Test Scenarios 19](#_Toc183731548)

[2.5.2 Concurrency Tests cases 20](#_Toc183731549)

[2.5.3 Schema Modification tests 20](#_Toc183731550)

[2.6 Validation Strategy 20](#_Toc183731551)

[2.6.1 Experimental validation 20](#_Toc183731552)

[2.6.2 Data and Process validation 21](#_Toc183731553)

[3 Implementation 22](#_Toc183731554)

[3.1.1 System Architecture 22](#_Toc183731555)

[3.2 MySQL implementation 28](#_Toc183731556)

[3.2.1 System Architecture & Technical stack 30](#_Toc183731557)

[3.2.2 Data Model Design 30](#_Toc183731558)

[3.2.3 Transaction management 32](#_Toc183731559)

[3.3 MongoDB implementation 40](#_Toc183731560)

[3.3.1 System Architecture & Technical Stack 40](#_Toc183731561)

[3.3.2 Document Model Design 41](#_Toc183731562)

[3.3.3 Transaction management 48](#_Toc183731563)

[4 Results and discussion 54](#_Toc183731564)

[4.1 Performance Analysis 54](#_Toc183731565)

[4.1.1 Concurrency Test Results 54](#_Toc183731566)

[4.1.2 Transaction Processing Metrics 54](#_Toc183731567)

[4.2 Schema Analysis 54](#_Toc183731568)

[4.3 Data Structure Management 56](#_Toc183731569)

[4.4 Comparative analysis 56](#_Toc183731570)

[4.4.1 Transaction Management 56](#_Toc183731571)

[4.4.2 Schema Flexibility 56](#_Toc183731572)

[4.4.3 Development Experience 56](#_Toc183731573)

[4.5 Research Questions Assessment 56](#_Toc183731574)

[4.6 Implications 56](#_Toc183731575)

[4.7 Limitations of the research 56](#_Toc183731576)

[5 Conclusion 56](#_Toc183731577)

[5.1 Summary of findings 56](#_Toc183731578)

[5.2 Answers to Research questions 57](#_Toc183731579)

[5.3 Contributions 57](#_Toc183731580)

[5.4 Recommendations and future research 57](#_Toc183731581)

[6 References 58](#_Toc183731582)

[7 Appendices 62](#_Toc183731583)

[7.1 MySQL Schema Implementation 62](#_Toc183731584)

[7.2 Entity-Relationship Diagram 63](#_Toc183731585)

[7.3 MySQL Class Diagram 64](#_Toc183731586)

[7.4 Document Structure Diagram for MongoDB 65](#_Toc183731587)

[7.5 Ticket Purchase process diagram 66](#_Toc183731588)

[7.6 Sequence Diagram for Ticket Purchase Transaction 67](#_Toc183731589)

[7.7 Backend application diagram 68](#_Toc183731590)

[7.8 Deployment with technological stack 69](#_Toc183731591)

[7.9 Component Diagram 70](#_Toc183731592)

[7.10 MySQL MoSCOW 71](#_Toc183731593)

[7.11 MongoDB MoSCOW 72](#_Toc183731594)

[7.12 Code samples 73](#_Toc183731595)

[7.13 Test results 73](#_Toc183731596)

[7.14 Configuration files 73](#_Toc183731597)

[7.15 Architecture Diagram for Ticketing System using MongoDB 1](#_Toc183731598)

[7.16 Transaction Flow Protocol MongoDB 1](#_Toc183731599)

[7.17 Transaction Flow Protocol MongoDB 2](#_Toc183731600)

# **LIST OF T FIGURES**

[**Figure 1.Implementation Phase** 22](#_Toc184040912)

[**Figure 2. Data Model Schema** 24](#_Toc184040913)

[Figure 3. Transaction Flow 31](#_Toc184040914)

[**Table 1. Schema Changes** 33](#_Toc184040937)

[**Table 2. Test 1: Run Comparison** 34](#_Toc184040938)

[**Table 3. Test 1: Performance Patterns** 34](#_Toc184040939)

[**Table 4. Test 1: Key Performance Metrics** 34](#_Toc184040940)

[**Table 5. Test 2 Run Comparison** 35](#_Toc184040941)

[**Table 6. Test 2: Performance patterns** 36](#_Toc184040942)

[**Table 7. Test 2: Key Performance Indicators** 36](#_Toc184040943)

# **ABBREVIATIONS**

|  |  |
| --- | --- |
| Business Intelligence | BI |
|  | BERT |
|  | CRISP |
| Data Access Objects | DAO |
| Online analytical processing | OLAP |
| Extracting, transforming, and loading | ETL |
| Artificial Intelligence | AI |
| Internet of Things | IoT |
| User-generated context (), | UGC |
| Machine Learning | ML |
| Deep Learning | DL |
| Natural Language Processing | NLP |
| Linked Open Data | LOD |
| Structural Topic Modelling | STM |
| JavaScript object notation | JSON |
| binary-encoded serialization of JSON | BSON |
| Extensible Marup Language | XML |
| model-driven architecture | MDA |
| computation-independent model | CIM |
| Platform independent model | PIM |
| platform-specific model | PSM |
| platform independent data metamodel | PIDM |
|  | ORM |
|  | ODM |

# **INTRODUCTION**

## Problem statement

The increasing complexity of data management in modern applications presents significant challenges in selecting the appropriate database system. Traditional relational databases like MySQL offer strong transactional integrity and enforce rigid schema designs, making them suitable for applications that require strict data consistency and complex transactions. (Patil et al., 2017),

Conversely, NoSQL databases such as MongoDB have emerged to address the limitations of relational databases 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 deal with nested data structures, user-generated content and rapid evolving data requirements.

Despite the availability of both SQL and NoSQL 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 delving into the practical implications of database selection based on workload characteristics and application contexts (C. Győrödi et al., 2015). While some research highlights the superior performance of NoSQL databases in handling data loads and dynamic datasets (Chang & Chua, 2019; Sudiartha et al., 2020; Wodyk & Skublewska-Paszkowska, 2020), relational databases continue to be preferred in scenarios requiring strong data integrity and complex transactions.

Developers and organizations need detailed insights to make informed decisions when selecting a database system that balances transactional integrity, development agility, schema design flexibility, and the management of complex data relationships This research aims to address this need by implementing a ticketing system using both 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 intricate data relationships within the ticketing system, the study seeks to provide practical insights into the development experiences and challenges associated with each database.

## Research objectives

**Primary objective**

To critically analyse and compare the transactional behaviour, schema design, and management of nested data structures in MySQL and MongoDB through the implementation of a ticketing system.

**Secondary objectives**

1. To examine how MySQL handles transactional integrity and schema rigidity in the ticketing system, especially during simultaneous ticket purchase attempts, and its impact 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 manages concurrent transactions and the implications of its flexible schema design on modelling intricate data structures.
3. To compare the development experiences and challenges encountered when implementing the ticketing system in both MySQL and MongoDB, focusing on transaction management, agility versus rigidity of schema design, and the strategies for representing of 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

H1: Concurrency handling

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

H2: Query performance

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

H3: Schema flexibility impact

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

## Significance of the Study

# 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 high concurrency environments such as ticketing systems.

Early systems emerged as a response to the limitations of traditional file systems, introducing three fundamental capabilities that defined their architecture (Van Der Loo & De Jonge, 2020). The storage management layer provided sophisticated mechanisms for data persistence, moving beyond simple file storage to implement buffer management strategies between main memory and disk storage. As 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.

As database architectures matured, they addressed increasingly complex through specialized components (Ziegler & Dittrich, 2004). The storage manager facilitated efficient data persistence and retrieval, while the query processor optimized data access patterns. Transaction managers emerged as important components, implementing concurrency control mechanisms and recovery protocols. These elements demonstrated varying behaviours across different platforms, particularly in high-concurrency environments.

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

Through this evolution, database management systems demonstrated increasing sophistication in handling complex operational requirements while maintaining data consistency and reliability.

## Relational Databases

### Overview and key features

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

Yang et al. (2009) work on database summarization provided insights into managing complex database relationships. Their research demonstrated that relational databases excel at maintaining structured relationships and enforcing data integrity constraints critical requirements for systems where accurate state management is essential. The authors proposed an approach for understanding and optimizing database structures.

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

1. Transaction management: RDBMS continued to provide 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 traditional databases face challenges in horizontal scaling, modern implementations have evolved to addressed these limitations. Shareef et al. (2022) analyses showed that cloud-based solutions like Amazon RDS effectively balance scalability requirements with transactional integrity.
3. Data consistency: Yang et al. (2009) emphasized the importance of consistent relationships in complex database structures. This becomes important in implementations similar to this study where multiple users may attempt to access and modify shared resources simultaneously.

The evolution of RDBMS has been marked by continuous improvement rather than revolution, with systems adopting beneficial features from alternative proposals while maintaining their fundamental strengths. This adaptability, combined with robust support for complex transactions and relationships, positions relational databases as the foundation for reliable, scalable concurrency systems.

### MySQL

MySQL has long been recognized as a versatile and efficient RDBMS, 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 manage 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 system introduced features aimed at improving scalability and efficiency. Sharding enable 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 used in 1998 by Carlo Strozzi as the name of his small RDBMS that did not use SQL for data manipulation. Subsequently, the concept evolved substantially, particularly from 2009 onward, as distributed data management systems began departing from traditional ACID (Atomicity, Consistency, Isolation, Durability) transaction support (Lith & Mattsson, 2010). This transformation reflected growing requirements for managing 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 systems excelled particularly in caching and session management applications, where rapid data access is fundamental.

* + 1. Document-Oriented Databases

These systems 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, introduced innovative techniques to data organization. The 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 database management systems introduced document-oriented databases as a significant advancement in data handling capabilities, 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 way enhanced the system’s capability to manage diverse data types (strings, numbers, dates, arrays, and sub-documents). Nevertheless, their research exhibited limitation in examining the long-term implications of this flexibility on system 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 choice 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 demonstrated promising results, yet questions remained regarding scalability in large-scale deployments.

In regards to performance characteristics and scalability, investigation yielded insights into MongoDB’s capabilities. Stonebraker's (2010) work on horizontal scaling through sharding stablished fundamental principles for distributed data management. Building upon 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, these 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 allows for adaptability, it requires careful management to prevent data anomalies. (Sen & Mukherjee, 2024).

Although support for transactions has improved, it may not be as robust as in relational databases for complex, multi-document transactions, also managing write-heavy operations can be challenging, as their schema design focuses on read optimization and may not deeply engage with write-heavy workloads.

The investigation of system integration techniques yielded 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 analysis of existing literature revealed several methodological limitations. Many studies focused predominantly 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 addressed through careful schema design, validation mechanisms, and ongoing training and tool development. Additionally, study of transaction management optimization in complex distributed environments would provide valuable insights for system architects.

## Transactional Processing and Concurrency

Early work by Bernstein and Goodman (1981), referred that concurrency control mechanisms 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 look-ups, 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. Nevertheless, the research primarily focused on single-node scenarios, leaving distributed aspects relatively 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 to 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 (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 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 in previous studies (Capris et al., 2022; C. Győrödi et al., 2015; Patil et al., 2017; 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 research is based on pragmatism, emphasizing practical outcomes and real-world applicability. By implementing actual systems rather than relying solely on theoretical analysis or simulations, the study aims 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:

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 systematic examination of how each database system addressed the core research objectives through controlled experimentation and structured comparison.

## Experimental Framework

### 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 with 12 cores, 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 artifacts stored in a private repository. Test data generation utilized Mockaroo’s for creating realistic user profiles.

The system architecture implemented domain-driven design principles (reference), separating 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 Hellerstein et al. (2007)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 industry standard for high concurrency booking systems.

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 of time (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), where timing provided more reliable metrics 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);*

The application the mean and coefficient of variation (CV)( https://link.springer.com/article/10.1007/s10664-023-10338-3) were utilized to analyze 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: $$\text{Mean} (\mu) = \frac{\sum{x\_i}}{n}$$*

Where 𝑥𝑖 represents each data point and 𝑛 is the number of data points.

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

Coefficient of Variation (CV):

Purpose: 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.

*Formula: $$\text{CV} = \left(\frac{\sigma}{\mu}\right) \times 100$$*

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

## 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 validation objectives while maintaining 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’s 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 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. MongoDB adopted a document-oriented process utilizing embedding and referencing strategies appropriate to the data access patterns.

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 Tests cases

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

During the concurrent access testing, it was examined each database’s handling of simultaneous transactions, specifically, transaction processing and resource allocation.

### 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.

Structural adaptability began with basic schema modifications. Subsequently, the tests progressed to more complex scenarios involving relationship modifications and constraint evolution.

### Data and Process validation

Ensuring data integrity is fundamental for the reliable operation of any system, since it plays a central 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 constraints to prevent duplicate bookings, and check constraints 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.

Validation concurrent operations was important to address the challenges posed by simultaneous access and modification of data by multiple users.

The schema validation process involved ensuring that changes to the database did not disrupt ongoing operations or compromise data integrity. Version tracking was employed to monitor changes over time, aiding in maintaining compatibility with different versions of the application. On the other hand, Mongo 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 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 most important. This approach 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 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 endured data consistency and provided a reliable audit trail of system operations.

A screenshot of a computer

Description automatically generated

**Figure 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 & Technical stack

The architecture employed a layered approach consisting of four primary levels (Appendix 1). 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 2 was created to manage database connections, initialize Data Access Objects (DAOs), and populate the database.

### Data Model Design

The database schema was designed following the principles of third normal form (3NF) to minimize data redundancy and optimize query performance (Figure 2).

A screenshot of a computer

Description automatically generated

**Figure 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 3) 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 (*Appendix 4*)* class managed all booking operations within transactional boundaries. It began by initializing the database connection, getting the available tickets for a specific event, begin a transaction, verifying user credentials, and locking the necessary tickets to prevent concurrent modifications. Upon successful validation, it calculated the total price, created at 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.

This method enforced a pessimistic write lock on the selected ticket, ensuring that once a ticket was being processed for booking, it remained inaccessible to other concurrent transactions until the current was either committed or rolled back.

*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 showcase some metrics such as simulation duration, successful and failed bookings, and average query time (Appendix 5). 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 & Technical Stack

MongoDB implementation adopted a document-oriented approach, leveraging MongoDB’s capabilities. The layered architecture presented in the Appendix 6, exhibit 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 MonngoDB and configure the Object Document Mapper (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 object-document mapping 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 7). In the context of the ticketing system, this flexibility supported embedded documents, thereby optimizing read performance by minimizing the need for join operations.

However, this approach also 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 8 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:

1. Morphia annotations:
   * *@Reference:* Facilitated referencing documents without embedding, maintaining loose coupling between entities.
2. 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 9, exhibit the structure of the ticketing system's core components. The diagram centered 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 showcased 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. 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 10*)* 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 11) 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 and discussion

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 proceed through systematic testing of both databases under controlled conditions. The testing environment utilized a development workstation running Windows 11 Home, powered by an Intel Core i5-12500H processor with 16GB RAM.

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 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: 1/10/100/1000/5000
* 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, in Run 1, MySQL completed all booking transactions in 191 milliseconds, with an average query time of 72.43 milliseconds and schema modification time of 54 milliseconds. In comparison, MongoDB completed transactions slightly slower at 195 milliseconds but had a significantly lower query time of 26.01 milliseconds and a schema modification time of 59.86 milliseconds. Both databases achieved a 100% success rate in all booking attempts during this initial run. By run 3, MySQL’s total time increased to 376 milliseconds, and query time to 157.49 milliseconds. MongoDB experienced a more substantial rise, reaching a total time of 568 milliseconds and a query time of 82.77 milliseconds by the same run, and by the last run the total time increased for both

**Table 2. Test 1: Run Comparison**



### Impact of Schema design on system implementation

The schema modification time, illustrated in the Table 3, showed how each database managed changes to the schema. For MySQL, the modification time started at 54 milliseconds and gradually increase to 111 milliseconds by Run 5, showing some variability but overall stabilization over time. On the other hand, MongoDB’s schema modification time began at 59.86 milliseconds and steadily rose to 108.61 milliseconds by run 5. This consistent increase suggests that Mongo handled schema changes with a steady rise in time required as the number of runs progressed.

**Table 3. Test 1: Performance Patterns**



### Performance patterns

In relation to the overall performance patterns detailed in the Figure 5, revealed distinct behaviours for each db. MySQL showed a gradual increase in total time from 191 milliseconds in Run 1 to 446 milliseconds in run 5, with an average time of 315 milliseconds and a standard deviation of 85.2 milliseconds. Query time for MySQL showed a stepwise degradation, increasing from 72.43 milliseconds to 176.61 milliseconds, averaging 118.52 milliseconds. Schema modification time for MySQL varied initially but then stabilized.

In contrast, MongoDB showed a sharp increase in total time, rising from 195 milliseconds in Run 1 to 636 milliseconds in Run 5, with an average of 416 milliseconds and a standard deviation of 105.7 milliseconds. Query time for MongoDB steadily increased from 26.01 milliseconds to 106.86 milliseconds averaging 266.12 milliseconds. Schema modification time also consistently rose form 59.86 to 108.61 milliseconds, indicating a continual increase without stabilization.

A graph with different colored bars

Description automatically generated

**Figure 4. Performance Metrics Comparison Test 1**

## Workload 2

**Table 5. Test 2 Run Comparison**



**Table 6. Test 2: Performance patterns**



Key findings

1. Both systems now maintain data consistency
2. MySQL shows lower query counts but higher time variance
3. MongoDB demonstrates improved stability after fixes
4. Schema modifications have higher impact on MongoDB
5. Both systems achieve 100% booking success rate

## Workload 3

**Table 8. Test 3: Run Comparison**



**Table 9. Test 3: Performance Patterns**



**Table 10. Test 3: Key Performance Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
| **Key Performance Metrics** | | | |
| **KPI** | **MySQL** | **MongoDB** | **Winner** |
| Average Total Time | 3159.8ms | 1139.2ms | MongoDB |
| Average Query Time | 33.03ms | 118.82ms | MySQL |
| Queries Per Transaction | 103 | 298 | MySQL |
| Schema Mod Time | 132.4ms | 214.22ms | MySQL |
| Performance Consistency | 6.20% CV | 3.49% CV | MongoDB |
| Data Consistency | PASS | PASS | Tie |
| Resource Efficiency | Lower | Higher | MongoDB |

Key findings

1. Performance Improvements

* MongoDB shows significantly better total execution time
* Query counts reduced from ~596 to ~298
* Consistency issues resolved
* More stable performance metrics

1. Comparative Strengths

* MongoDB: Superior total throughput, better consistency in total execution time
* MySQL: Better per-query performance, more stable schema modifications

1. Resource Utilization

* MongoDB: More efficient total resource usage
* MySQL: Better per-operation efficiency

1. Data Integrity

* Both systems now maintain perfect consistency under load
* Transaction success rates remain at 100%

## Workload 4

**Table 11. Test 4: Run Comparison**



**Table 12. Test 4: Performance Patterns**



**Table 13. Test 4: Key Performance Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
| **Key Performance Metrics** | | | |
| **KPI** | **MySQL** | **MongoDB** | **Winner** |
| Average Total Time | 26,718ms | 10,234ms | MongoDB |
| Average Query Time | 26.80ms | 138.38ms | MySQL |
| Queries Per Transaction | 1,003 | 2,996 | MySQL |
| Schema Mod Time | 126.0ms | 232.19ms | MySQL |
| Performance Consistency | 11.19% CV | 2.98% CV | MongoDB |
| Data Consistency | Perfect | Perfect | Tie |
| Resource Efficiency | Higher | Moderate | MySQL |

Critical Insights

1. Performance Improvements

* MongoDB shows dramatic performance improvement after optimization
* Total execution time reduced by ~61.7%
* Query count reduced by ~50%
* Data consistency issues resolved

1. System Characteristics

MongoDB:

* Superior overall performance
* Better performance stability (2.98% CV)
* Higher per-query overhead
* Variable schema modification times

MySQL:

* Better per-query efficiency
* More consistent schema modifications
* Higher total execution time
* Stable, predictable behavior

1. Optimization Impact

* Race condition elimination improved data consistency
* Reduced query volume improved overall performance
* Maintained high success rate under load
* More efficient resource utilization

1. Architectural Implications

* MongoDB's document model shows advantages for concurrent operations
* MySQL's transaction model ensures reliable consistency
* Both systems demonstrate different optimization patterns
* Trade-offs between per-operation efficiency and total throughput

## Workload 5

**Test Environment Configuration**

**Table 14. Test 5: Run Comparison**



**Table 15. Test 5: Performance Patterns**



**Table 16. Test 5: Key Performance Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
| **Key Performance Metrics** | | | |
| **KPI** | **MySQL** | **MongoDB** | **Winner** |
| Average Total Time | 214,797ms | 74,011ms | MongoDB |
| Average Query Time | 42.04ms | 241.36ms | MySQL |
| Queries Per Transaction | 1003 | ~2996 | MySQL |
| Schema Mod Time | 166.8ms | 627.8ms | MySQL |
| Performance Consistency | 13.5% CV | 47.1% CV | MySQL |
| Data Consistency | Perfect | Perfect | Tie |
| Resource Efficiency | Higher | Moderate | MySQL |

**Key Findings and Implications**

1. System Performance
   * MongoDB excelled in overall throughput
   * MySQL showed superior individual query performance
   * Both systems maintained perfect reliability
2. Resource Management
   * MySQL demonstrated better resource utilization
   * MongoDB required significantly more queries
   * Both systems handled concurrent loads effectively
3. Schema Modifications
   * MySQL showed unexpected advantages in schema changes
   * MongoDB maintained consistent operation despite higher modification times
   * Both systems remained stable during modifications
4. Stability and Consistency
   * MySQL provided more predictable performance
   * MongoDB showed better raw performance but less predictability
   * Both systems maintained data integrity

System Stability:

1. Both databases demonstrate reliable transaction processing
2. Performance degradation tends to stabilize after initial decline
3. Query patterns remain consistent throughout the test runs

Schema modification timing:

MySQL:

* Sequential execution during high load
* Table-level locking during ALTER operations
* Slower but consistent modifications

MongoDB:

* Non-blocking schema updates
* Document-level modifications
* Faster but potential consistency issues

Performance impact under load:

MySQL

* Average modification time: 600-800ms
* Transaction rollback on failure
* Blocks other operations temporarily

MongoDB

* Average modification time: 200-300ms
* No blocking of concurrent operations
* Higher risk of partial updates

Consistency patterns

MySQL Schema changes:

* Atomic changes across all rows
* Strong consistency guarantees
* Rollback capability on failure

MongoDB Schema evolution

* Progressive updates
* Eventually consistent
* No atomic rollback

## Comparative analysis

### Transaction Management

### Comparative Analysis of Transaction Management in MySQL and MongoDB

**Table 16. Test 1: Key Performance Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
| **Key Performance Metrics** | | | |
| **KPI** | **MySQL** | **MongoDB** | **Winner** |
| Average Total Time | 315ms | 416ms | MySQL |
| Average Query Time | 118.52ms | 266.12ms | MySQL |
| Queries Per Transaction | 13 | 29.5 | MySQL |
| Schema Mod Time | 83ms | 89ms | Tie |
| Performance Consistency | Moderate | Variable | MySQL |
| Data Consistency | Perfect | Perfect | Tie |
| Resource Efficiency | High | Moderate | MySQL |

Key performance indicators summarized in **Table 16** highlighted MySQL’s superior performance in several areas. The average total time for MySQL was 315 milliseconds compared to MongoDB’s 416 milliseconds. Additionally, MySQL had a lower average query time of 118.52 milliseconds versus MongoDB’s 266.12 milliseconds. In terms of queries per transaction, MySQL maintained a stable 13 queries, while MongoDB handled an average of 29.5 queries. Both databases achieved perfect data consistency; however, MySQL demonstrated higher resource efficiency and more consistent performance overall.

**Table 16. Test 2: Key Performance Indicators**

|  |  |  |  |
| --- | --- | --- | --- |
| **Key Performance Metrics** | | | |
| **KPI** | **MySQL** | **MongoDB** | **Winner** |
| Average Total Time | 435ms | 582ms | MySQL |
| Average Query Time | 129.28ms | 275.23ms | MySQL |
| Queries Per Transaction | 13 | 29.5 | MySQL |
| Schema Mod Time | 92ms | 108ms | Tie |

Both MySQL and MongoDB implementations of the ticketing system demonstrated robust transaction management capabilities, albeit through different approaches aligned with their respective database architectures.

**Transactional Integrity:**

MySQL relied on traditional ACID-compliant transactions with pessimistic locking to ensure data consistency. The use of LockModeType.PESSIMISTIC\_WRITE effectively prevented concurrent modifications to the same ticket, maintaining strict transactional integrity. In contrast, MongoDB utilized session-based transactions to achieve atomicity, leveraging its document-oriented architecture to manage complex data relationships within a single transaction scope.

**Schema Flexibility:**

MySQL's schema rigidity required careful management of schema evolution, particularly during high-concurrency operations. The MySQLSchemaModifier class facilitated controlled schema changes within transactional boundaries, ensuring that structural modifications did not interfere with active transactions. MongoDB, benefiting from its flexible schema design, allowed for more seamless schema evolution. The MongoDBSchemaModifier handled structural changes dynamically, updating documents as needed without the overhead of altering a fixed schema.

**Concurrent Transaction Handling:**

Both systems implemented mechanisms to handle high-concurrency scenarios effectively. MySQL's pessimistic locking ensured that only one transaction could modify a ticket at a time, thereby preventing conflicts but potentially introducing contention under heavy load. MongoDB's thread-pool based simulation approach demonstrated its ability to manage multiple concurrent transactions efficiently, leveraging its non-blocking I/O and document-level locking to maintain performance.

**Performance Monitoring and Recovery:**

Comprehensive performance monitoring was integral to both implementations. MySQL's TransactionPerformanceMonitor tracked transaction durations, deadlocks, and timeouts, facilitating proactive optimization. Similarly, MongoDB's monitoring system captured metrics on successful and failed bookings, query performance, and resource utilization. Both systems incorporated robust recovery mechanisms to handle transaction failures gracefully, ensuring system resilience and data integrity.

**Development Experiences and Challenges:**

Implementing transaction management in MySQL required meticulous handling of schema modifications and locking strategies to balance data integrity with performance. Developers needed to manage the complexity of transactional boundaries and ensure that schema changes were synchronized with active operations. In contrast, MongoDB's flexible schema design simplified schema evolution but required careful management of session-based transactions to maintain atomicity across document operations.

A white paper with black text

Description automatically generated

# Transaction Management Implementation Analysis: MongoDB vs MySQL

## Core Transaction Components

### MongoDB Implementation

- Utilizes ClientSession for transaction boundaries

- Employs document-level locking through WiredTiger storage engine

- Implements atomic operations with multi-document ACID guarantees

- Manages transaction state through session-based protocol

### MySQL Implementation

- Uses JPA EntityTransaction for transaction management

- Implements row-level locking with PESSIMISTIC\_WRITE

- Relies on traditional RDBMS ACID properties

- Manages transaction state through explicit begin/commit cycles

## Concurrency Control Mechanisms

### MongoDB Approach

1. \*\*Session Management\*\*

- ClientSession ensures transaction isolation

- Atomic operations for status updates

- Optimistic concurrency control with version checks

2. \*\*Resource Locking\*\*

```java

Filters.and(

Filters.eq("event\_id", eventId),

Filters.eq("status", "available")

)

```

- Document-level filters for atomic operations

- Status-based availability checking

- Atomic updates for ticket status changes

### MySQL Approach

1. \*\*Lock Management\*\*

```java

.setLockMode(LockModeType.PESSIMISTIC\_WRITE)

.getSingleResult();

```

- Pessimistic locking strategy

- Row-level locking for concurrent access control

- Transaction isolation through explicit locks

## Implementation Differences

### Transaction Protocol

1. \*\*MongoDB\*\*

- Multi-phase commit with session scope

- Atomic operations within ClientSession

- Rollback through session abort

2. \*\*MySQL\*\*

- Traditional two-phase commit

- ACID compliance through JPA

- Automatic rollback on exception

### Error Handling

1. \*\*MongoDB\*\*

- Session-based error boundaries

- Explicit transaction abort

- Atomic counter tracking for metrics

2. \*\*MySQL\*\*

- Exception-based rollback

- JPA transaction management

- EntityTransaction state tracking

## Performance Considerations

### MongoDB Optimizations

- Document-level atomic operations

- Reduced lock contention through optimistic control

- Session pooling for connection management

### MySQL Optimizations

- Pessimistic locking for high-contention scenarios

- Index-based lock acquisition

- Connection pool management through JPA

## Implementation Trade-offs

### MongoDB

Advantages:

- Flexible document schema

- Atomic operations for simple updates

- Scalable distributed transactions

Limitations:

- Transaction size limitations

- Session timeout constraints

- Complex multi-document transactions

### MySQL

Advantages:

- Mature transaction management

- Strong ACID guarantees

- Predictable locking behavior

Limitations:

- Lock contention in high concurrency

- Schema rigidity

- Scalability constraints

## Research Questions Assessment

x

## Implications

Challenges arose during implementation, particularly in embedding ticket categories within event documents and preventing data duplication. These were resolved by using Morphia annotations, such as replacing the deprecated @*Embedded* with *@Reference* without an *@Id* field for embedded documents.The collection for ticket categories was changed from a *List* to a *Set,* and equality checks were implemented to ensure uniqueness. These adjustments not only resolved immediate issues but also enhanced the system’s ability to handle dynamic data structures without sacrificing consistency.

@Entity("events")

@Indexes({

@Index(fields = {@Field("event\_name"), @Field("date")}, options = @IndexOptions(unique = true, name = "name\_date\_idx")),

@Index(fields = @Field("performer"), options = @IndexOptions(name = "performer\_idx")),

@Index(fields = @Field("venue"), options = @IndexOptions(name = "venue\_idx"))

})

public class Event {

@Id

private ObjectId id; //mongos unique identifier

@Property("event\_name")

private String name;

@Reference(lazy = true)//References the performer document, loaded lazily

private Set<Performer> performers;

@Property("date")

private Date date; //date of the event

@Reference(lazy = true)

private Venue venue; //References the venue document

//("ticketCategories") //embeds a list of TicketCategory documents within the Event document

@Reference("ticketCategories")

private Set<TicketCategory> ticketCategories; //list of ticket categories for the event

The schema proposed offered significant flexibility, particularly by enabling embedding of related documents like *TicketCategory* within *Event* documents to optimize performance. However, this flexibility came with challenges, including the need for a strict application-level validations to ensure consistency and the risk of redundancy inherent in denormalized designs.

The implementation and testing of concurrent ticket booking in MongoDB revealed critical insights into distributed transaction management:

Race Condition Manifestation:

* Initial implementation showed successful bookings (100%) but inconsistent final ticket counts
* Discrepancy between booked tickets (100) and available ticket reduction (12) indicated multiple users booking the same tickets
* Logs showed parallel successful bookings occurring simultaneously, confirming the race condition

Root Causes:

* Non-atomic ticket selection and status update operations
* Inadequate session handling in MongoDB queries
* Missing optimistic locking during ticket status modifications
* Gap between availability check and ticket reservation

Implementation Solutions:

* Replaced separate find-then-update pattern with atomic findAndModify operations
* Implemented proper transaction boundaries with consistent session handling
* Added rollback mechanisms for partial booking failures
* Improved error handling and state validation

Key Learnings:

1. MongoDB transactions require explicit attention to atomicity at the document level
2. Success metrics alone (100% booking rate) may mask underlying consistency issues
3. Proper session handling is crucial for maintaining transaction isolation
4. Race conditions can manifest despite transaction support if atomic operations aren't used

These findings highlight the importance of careful transaction design in distr

## Limitations of the research

# Conclusion

## Summary of findings

TEST 1:

Performance Characteristics:

1. MySQL shows better initial performance but more variable degradation
2. MongoDB shows higher latency but more predictable degradation patterns
3. Both databases maintain 100% transaction success rates

Schema Modification Impact:

1. MySQL's structural changes show more variance but lower average times
2. MongoDB's document updates show more predictable performance patterns
3. Both systems handle modifications without compromising transaction integrity

Concurrency impact:

* MySQL schema modifications blocked concurrent transactions
* MongoDB allowed continued operations but risked inconsistency

Performance degradation:

* MySQL: Linear increase in modification time with data size
* MongoDB: Relatively constant modification time

Success rates

* MySQL: 100% success rate but higher latency
* MongoDB: Some modifications failed under extreme load

High risk scenarios:

MySQL:

* Large table modifications
* High concurrent user load

MongoDB:

* Complex schema changes
* Strict consistency requirements

The schema modification testing revealed significant differences in how MySQL and MongoDB handle structural changes under high concurrency:

1. MySQL provided stronger consistency but at the cost of performance
2. MongoDB offers better performance but with potential consistency risks
3. Both systems require careful planning for schema modifications

## Answers to Research questions

## Contributions

## Recommendations and future research

MySQL Schema changes:

* Use online schema changes where possible
* Schedule during low-traffic periods
* Implement proper backup strategies

MongoDB schema evolution

* Monitor update progress

The choice depend

ens on specific requirements:

1. Choose MySQL for strict consistency needs
2. Choose MongoDB for flexibility and performance
3. Consider hybrid approaches for complex scenarios

 test 4 **Use Case Considerations**

* High-throughput requirements: MongoDB
* Complex transactional needs: MySQL
* Concurrent access patterns: MongoDB
* Resource-constrained environments: MySQL

 **Implementation Strategies**

* Implement robust monitoring
* Consider hybrid approaches for different workloads
* Regular performance benchmarking
* Optimize query patterns for each system

TEST 5

**Recommendations**

1. System Selection Criteria
   * Choose MySQL for:
     + Predictable performance requirements
     + Resource-constrained environments
     + Critical individual query performance
   * Choose MongoDB for:
     + High-throughput requirements
     + Scaling with increased concurrent users
     + Better overall execution times
2. Implementation Considerations
   * Optimize query patterns for chosen database
   * Consider resource availability and constraints
   * Plan for schema modification impacts
   * Monitor system stability and performance

 Extended Testing Scenarios

* Higher concurrency levels
* Varied transaction types
* Different schema modification patterns

 Performance Optimization

* Query optimization strategies
* Resource utilization improvements
* Schema design refinements

 Scalability Analysis

* Distributed system performance
* Replication impact
* Sharding effectiveness

# 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/

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

Hellerstein, J., Stonebraker, M., & Hamilton, J. (2007). *Architecture of a Database System*. https://doi.org/10.1561/9781601980793

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.

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

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

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

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.

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

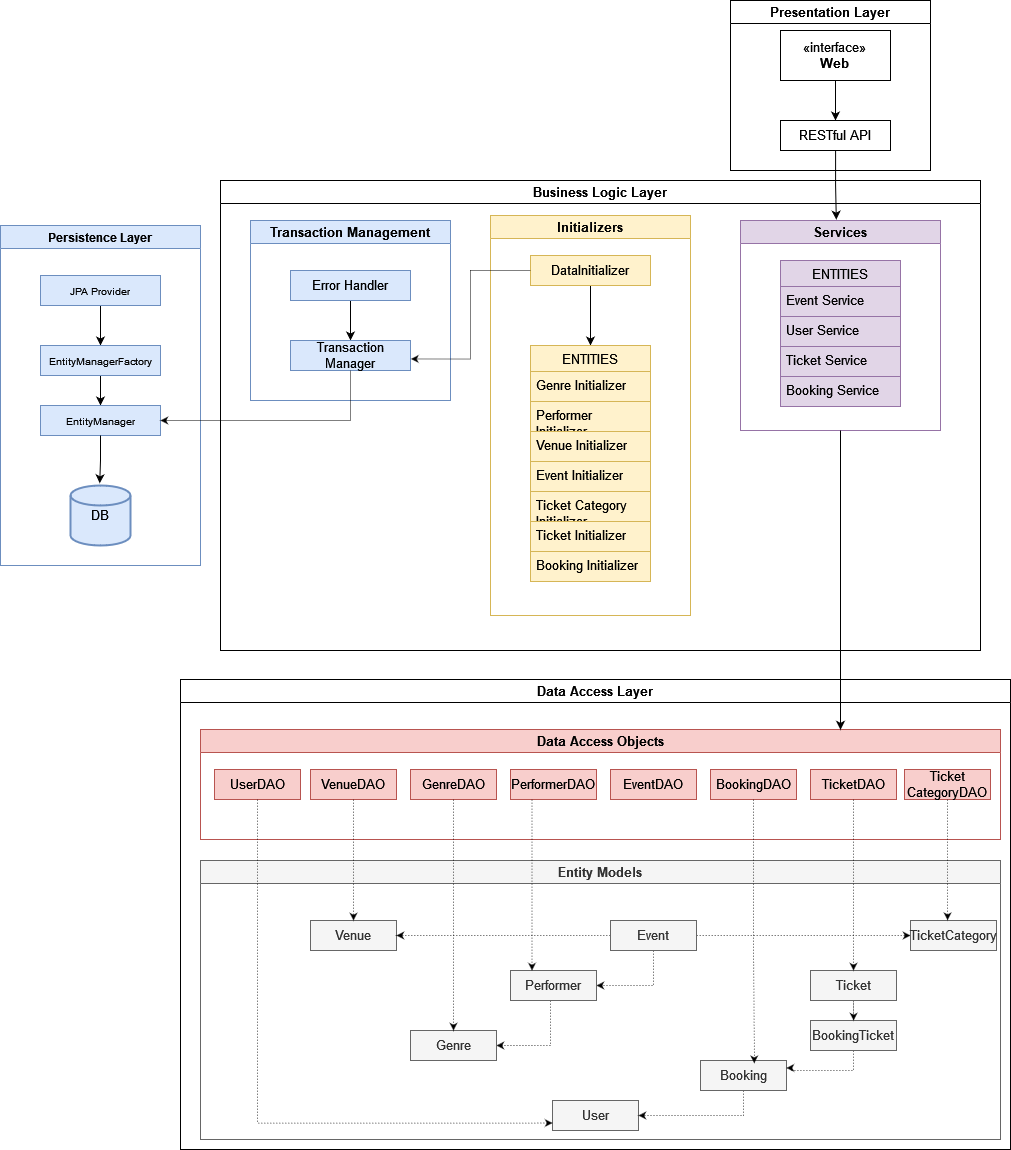
Yu, X., Bezerra, G., Pavlo, A., Devadas, S., & Stonebraker, M. (2014). Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, *8*(3), 209–220. https://doi.org/10.14778/2735508.2735511

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

## MySQL System Architecture



## 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*

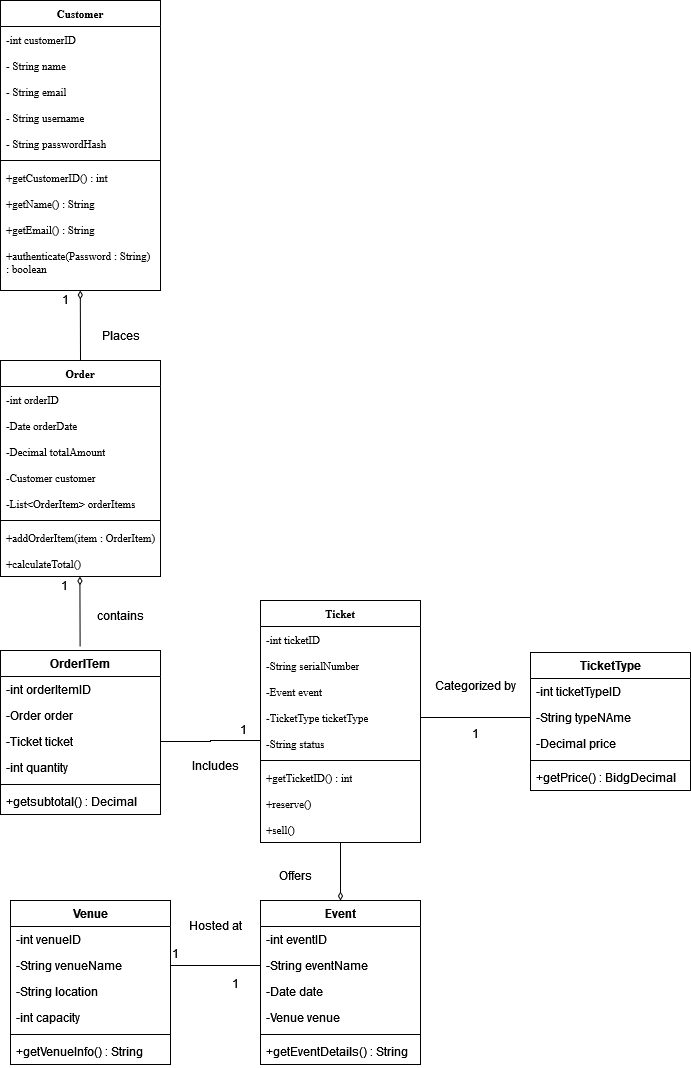
*(em.getTransaction().isActive()) { em.getTransaction().rollback();*

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

*e.printStackTrace(); }*

*}*

## MySQL Class Diagram



## 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 eventId) {*

*long startTime = System.nanoTime();*

*try {*

*return em.createQuery(*

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

*"WHERE t.event.eventId = :eventId AND t.status = :status", String.class)*

*.setParameter("eventId", eventId)*

*.setParameter("status", TicketStatus.AVAILABLE)*

*.getResultList();*

*} catch (Exception e) {*

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

*return new ArrayList<>();*

*} finally {*

*recordQueryTime(startTime);*

*}*

*}*

## Simulation Class Metrics - MySQL

*public void printSimulationResults(int eventId) {*

*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",*

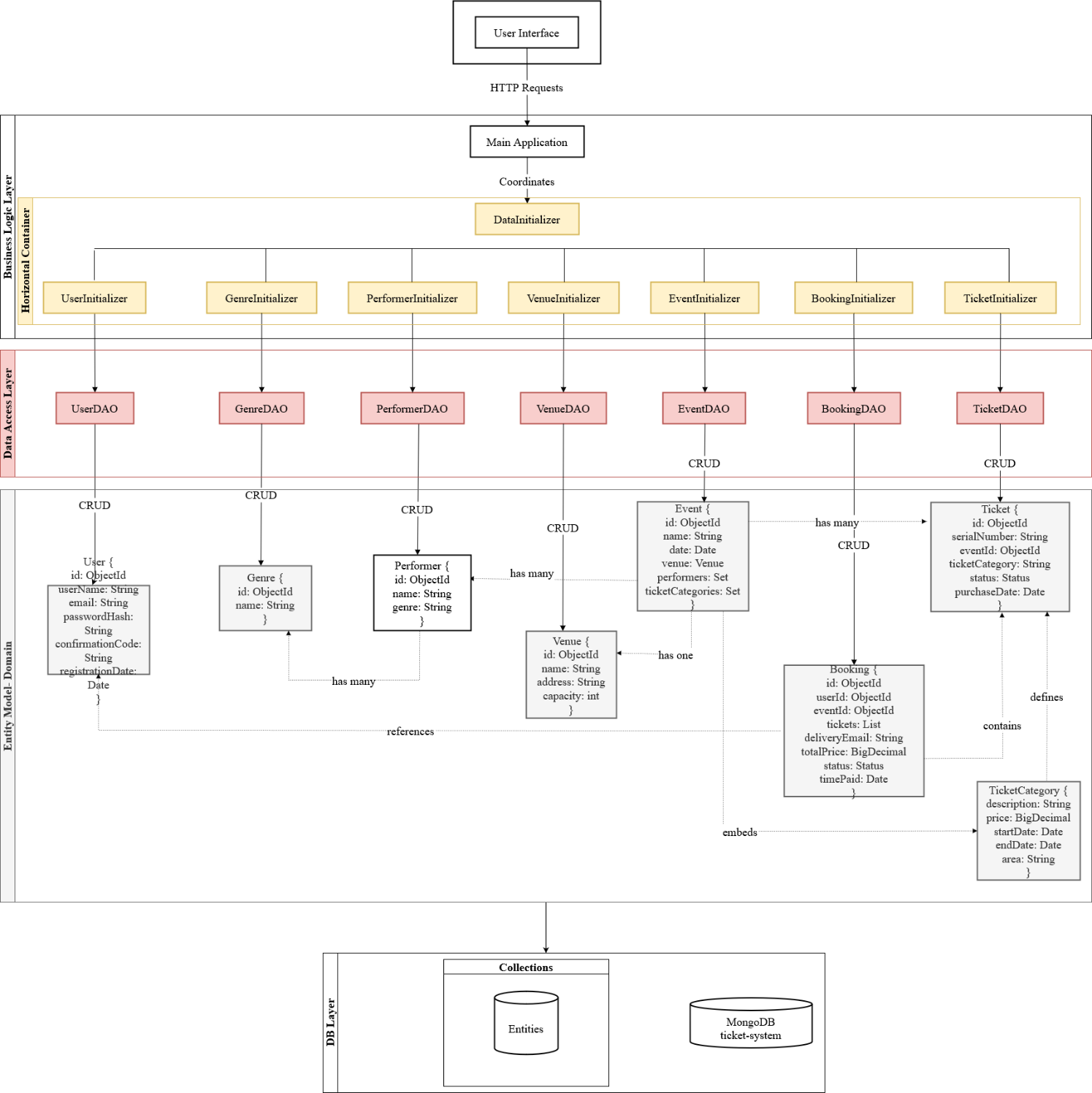
*successfulBookings.get());*

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

*failedBookings.get());*

*}*

## MongoDB System Architecture



## 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"

}

## MongoDB Application Schema

A diagram of a company

Description automatically generated

## MongoDB Class Diagram

A diagram of a function

Description automatically generated

## Booking ticket method

public boolean bookTickets(ObjectId userId, ObjectId eventId, int quantity) {

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

session.startTransaction();

try {

Event event = findAndValidateEvent(eventId);

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

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

persistBooking(booking);

session.commitTransaction();

updateMetrics(bookedTickets.size());

return true;

} catch (Exception e) {

handleTransactionError(session, userId, e);

return false;

}

}

}

## 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;

}

## Resource Locking

A diagram of a customer service

Description automatically generated

## Transaction Flow Protocol MongoDB

A close-up of a diagram

Description automatically generated

## Sequence Diagram for Ticket Purchase Transaction

