## Comparison of Schemas and Architectures

### MySQL architecture

***Strengths***

The system shows several architecture strengths that contributes to its maintainability and robustness. The adoption of a *Layered Architecture* splitting among the presentation, business, data access, and persistence layer. This design is aimed to enhance maintainability, facilitates dependency management, and improve testability.

In terms of data management, the system has an advantage, the entity relationship model which mirrors the business domain. Thus transaction management adheres to ACID principles, ensuring consistency and integrity. The use of Data Access Object (DAO) pattern promotes structured data access and maintains the distribution between entity models and data transfer objects.

The business logic is centralized within a service layer, which enforces consistent application of business rules and responsibilities. The initialization process supports system setup and ensure that the system works for operation upon deployment.

For maintenance and support, it has been incorporated error handling and logging mechanisms. Consistent naming conventions that promote uniformity across the codebase, and a defined dependency hierarchy that allow updates and modification without impacting the entire system.

***Challenges***

Complex query optimization in high-concurrency scenarios, it is hard to maintain performance, and may have a risk of data inconsistency.

A screenshot of a diagram

Description automatically generated

### MySQL class diagram

### A diagram of a computer Description automatically generated

### MongoDB

The approach has been similar, utilizing the document-oriented framework, and separating functionality across layers, including user management, event organization, ticket handling, and booking processes.

***Strengths***

Flexible data modelling, accommodating event structures and multiple ticket categories without necessitating structural changes, this is because the capability of embedded documents which also enables atomic updates and data retrieval. The absence of rigid schemas allows the system to adapt to evolving business requirements easy.

The reduce of query complexity and access due to the embedding documents, and the hierarchical data organization enable rapid access to nested information.

***Challenges***

In terms of transactional management, ACID limitations for complex booking scenarios that requires strict consistency, this affect related documents, since need meticulous handling, specially during concurrent booking attempts.

Data integrity, the lack of enforced schemas necessitates rigorous application-level validation to maintain data integrity and prevent inconsistencies

Query complexity is a potential drawback, generating reports involves detailed queries, and certain relational operations, or intricate aggregations, adding to the system’s operational complexity.

***Conclusion***

Both architectures demonstrate advantages and disadvantages for different cases. MySQL is strong in scenarios where strict data consistency and complex relationships are used, leveraging ACID compliance and structured schema. The architecture and DAO pattern proposed provided reliable transaction management.

MongoDB document-oriented approach offered superior flexibility for evolving business requirements and rapid data access through embedded documents. While this simplifies many operations, it presents challenges with ACID compliance and report queries.

A diagram of a company

Description automatically generated

### MongoDB class diagram

A diagram of a function

Description automatically generated

## Test Case Scenarios Transaction Integrity

### MySQL

Scenario-> ACID compliance will maintain data consistency across the booking process, even under stress conditions and potential failures.

Use Case:

Consideration of a popular concert where tickets are being sold. A customer, attempts to purchase different categories of tickets. The booking process involves multiple steps:

1. Create a booking record
2. Reserve specific seat
3. Payment
4. Generate ticket
5. Update inventory

If any step fails, the system must ensure no partial booking occurs.

Test Cases:

* Complete Successful Booking Flow: Verify that all steps execute smoothly without any hiccups.
* Payment Failure: Ensure the system rolls back the entire transaction if the payment step fails.
* Concurrent Booking Attempts**:** Test multiple users trying to book the same seats at the same time.

The outcome

All related tables remain consistent. No data anomalies or partial booking occur.

A screenshot of a computer

Description automatically generated

### MongoDB

Scenario -> When a highly anticipated concert releases tickets, thousands of users attempt to purchase tickets in the first minutes.

Use case:

High concurrency environment where users simultaneously attempt to book tickets. Each user tries to book 2 to 4 tickets for the same event.

Test Cases:

* Prevent Overselling: Ensure that no more tickets are sold than are available.
* Accurate Booking Records: Verify that each booking is accurately recorded without any duplicates.
* Transaction Management: Test proper handling of transactions, including locking mechanisms and conflict resolution.

The outcome

Data integrity is maintained with correct final ticket counts, accurate booking records, and no duplicate reservations.

System requirements:

* Consistency: Ensure accurate ticket counts
* Atomicity: Prevent partial transactions that could lead to data inconsistencies