Decentralized Cluster-Based NoSQL DB System

A report

Done by: Aya Almallah
Instructors:
Motasim Aldiab
Fahed Jubair

Contents

1 Introduction

This report offers a comprehensive overview of the implementation of a database system. during Atypon's Training in May 2023. The report aims to detail the design, development, and various aspects of the system's implementation, including data structures, multithreading, data consistency, load balancing, testing, software engineering principles, and DevOps practices.

2 DB Implementation

Data Model and Schema

Data Model:

1. The NoSQL database follows a document-based data model.
2. Data is organized into collections, similar to tables in relational databases.
3. Each database contains multiple collections.
4. Each collection has its own schema or structure that defines allowable fields and their data types for the contained documents.
5. This schema approach provides a level of data consistency within individual collections.

JSON Objects for Storing Documents:

1. JSON (JavaScript Object Notation) objects are the primary mechanism for storing database documents.
2. Each document is essentially a JSON object adhering to the schema of its parent collection.
3. JSON's flexibility allows documents to have varying structures within the constraints of the collection's schema.

Schema Design:

1. While the database system allows flexibility in document structure, it enforces a schema for each collection.
2. During document creation within a collection, the code checks that the document conforms to the defined schema for that collection.
3. This schema definition ensures that documents within a collection have a consistent structure, making it easier to work with and query the data.

Data Storage

Storage on Local File System:

1. Data is persisted within the local file system of each database node.
2. A hierarchical structure organizes data into directories, including levels for databases, collections, and individual documents.
3. This hierarchy enhances data organization and distinguishes data entities effectively.

Replication Strategy:

1. The database system implements a deliberate replication strategy to ensure data consistency and availability.
2. Strategies like master-slave replication or multi-master replication are employed based on system requirements.
3. Replication distributes data across nodes, providing redundancy, resilience, and enabling seamless failover mechanisms.

This report section delineates the database's data model, schema design, data storage, and replication strategy, highlighting the balance between schema flexibility within collections and rigorous schema enforcement at the collection level. Additionally, it underscores the meticulous data storage structure and the replication approach chosen for robust data management and availability.

Code Structure: Packages and Classes

In this section, I will provide an overview of the packages and classes that make up the codebase of the database system. Organizing code into packages and classes enhances modularity, maintainability, and readability.

Database Package

Within the Database package, there is a collection of classes that function as the foundation of the database system. These classes play a crucial role in establishing and overseeing the structure and data of the database. The classes and their responsibilities are as follows:

Database class

- The **Database** class acts as the primary container for the database. Each instance of this class represents a separate database.

Collection class

- The **Collection** class serves as a logical container within a database. It's used to group related documents together.

DocumentIndex class

- **DocumentIndex** objects are used to facilitate quick document retrieval within a collection. They define how data is organized for faster searches.

Controllers Package

The Controllers package is a crucial component of the database system, responsible for handling incoming requests, processing data, and orchestrating the flow of information between the clients and the underlying database. This package follows the Model-View-Controller (MVC) architectural pattern, ensuring separation of concerns and promoting maintainability.

The classes and their responsibilities are as follows:

DatabaseController

- **Responsibility**: Manages operations related to databases.
- **Key Functions**:
    - Creating databases.
    - Deleting databases.
    - Retrieving a list of all databases.

CollectionController

- **Responsibility**: Handles operations specific to collections within a database.
- **Key Functions**:
    - Creating collections.
    - Deleting collections.
    - Retrieving a list of all collections in a database.
    - Retrieving documents by key, value, or document ID.

DocumentController

- **Responsibility**: Manages documents within collections.
- **Key Functions**:
    - Creating documents.
    - Updating documents.
    - Deleting documents.
    - Retrieving a list of all documents in a collection.

Services Package

The Services package plays a pivotal role in the database system, containing classes responsible for business logic, data validation, and the execution of various database operations. These services abstract complex operations, ensuring separation of concerns and modularity in the application.

The classes and their responsibilities are as follows:

DatabaseService

- **Responsibility**: Manages database-related operations.
- **Key Functions**:
    - Creating databases.
    - Deleting databases.
    - Retrieving a list of all databases.
    - Locking and unlocking database resources for concurrent access.

CollectionService

- **Responsibility**: Handles operations specific to collections within a database.
- **Key Functions**:
    - Creating collections.
    - Deleting collections.
    - Creating and managing collection schemas.
    - Creating and managing collection indexes.
    - Filtering and querying data within collections.
    - Managing collection affinity values.

DocumentService

- **Responsibility**: Manages documents within collections.
- **Key Functions**:
  - Creating documents.
  - Updating documents.
  - Deleting documents.
  - Validating documents against collection schemas.
  - Handling document indexes.
  - Filtering and querying documents.

AffinityService

- **Responsibility**: Calculates and manages affinity values for collections.
- **Key Functions**:
  - Calculating collection affinity values.
  - Retrieving collection affinity values.

Validation Package

The Validation Package plays a critical role in ensuring data consistency and integrity within the database. It encompasses classes and components responsible for validating documents and schemas, ensuring that data adheres to predefined rules and standards.

The classes and their responsibilities are as follows:

DataTypes Enum

The **DataTypes** enum serves as a fundamental component of the validation system. It defines the allowable data types that can be used when specifying the structure of documents within collections.

- **Purpose**: Provides a standardized set of data types for schema validation.
- **Usage**: Enum values (e.g., STRING, LONG, DOUBLE) are referenced in schema definitions to specify the expected data types of document fields.

SchemaValidator Class

The **SchemaValidator** class validates schemas themselves. It ensures that the schema definitions provided for collections are valid and comply with the expected format.

- **Purpose**: Validates the structure and correctness of collection schemas.
- **Functionality**:
    - Checks if schema definitions follow the expected format.
    - Verifies that data types specified in the schema are valid (using the **DataTypes** enum).
    - Ensures schema consistency across collections.

DocumentValidator Class

The **DocumentValidator** class validates documents against predefined schemas. It ensures that documents conform to the expected structure defined for each collection.

- **Purpose**: Validates the conformity of documents with collection schemas.
- **Functionality**:
    - Compares the structure of incoming documents with the schema of the target collection.
    - Checks for the presence of all required fields.
    - Verifies that data types of fields match those defined in the schema.
    - Ensures data consistency within collections.

Managers Package

The Managers package in the database system contains classes responsible for managing and maintaining critical system resources and configurations. These managers play a crucial role in ensuring the reliability and scalability of the application.

The classes and their responsibilities are as follows:

StartManager

- **Responsibility**: Manages the startup process of the database system.
- **Key Functions**:
    - Initializes and populates the database structure during system startup.
    - Retrieves existing databases, collections, and their configurations.
    - Ensures a smooth startup sequence during application launch.

DatabaseManager

- **Responsibility**: Manages database instances and their resources.
- **Key Functions**:
    - Maintains a registry of databases.
    - Provides database-specific locks for concurrent access control.

ClusterManager

- **Responsibility**: Manages the cluster configuration and node information.
- **Key Functions**:
    - Keeps track of active nodes in the cluster.
    - Manages node discovery and registration.
    - Facilitates communication between nodes for distributed operations.

ShutdownManager

- **Responsibility**: Manages the graceful shutdown of the database system.
- **Key Functions**:
    - Saves indexes and affinity configurations before shutting down.
    - Ensures data consistency and persistence during system shutdown.

3 Multithreading and Locks

In the context of the database system, the effective management of concurrent client requests and data consistency remains paramount. This section provides an insight into the multithreading model, the types of locks utilized, and how they collectively ensure the reliability of the database system.

Multithreading Model

Overview: The database system adopts a multithreading model to handle concurrent client requests efficiently. Multiple threads are utilized to process client queries concurrently, enabling high throughput and responsiveness.

Concurrency Control: In a distributed environment, concurrency control is fundamental. By embracing multithreading, the system can execute multiple client requests simultaneously, optimizing resource utilization and reducing response times.

Locking Mechanisms

Lock Types: Various types of locks, including read-locks and write-locks, are employed to control access to shared resources. These locks prevent concurrent modifications that could lead to data corruption.

Database Locking: At the highest level, database-level locks are acquired to ensure exclusive access to an entire database, preventing conflicts between clients seeking to perform database-level operations.

The **createDatabase** and **deleteDatabase** methods within the **DatabaseService** class implement multithreading through the use of Java's **ReentrantReadWriteLock**. These write locks ensure thread-safe database creation and deletion processes, allowing multiple clients to perform these operations concurrently without the risk of data corruption.

Collection Locking: Collections within a database are protected by collection-level locks. These locks grant exclusive access to specific collections, guaranteeing data integrity within each collection.

In the **CollectionService** class, multithreading mechanisms are applied in the **createCollection**, **deleteCollection**, **filterByKey**, **filterByValue**, and **filterById** methods. These methods employ both read and write locks at both the database and collection levels, guaranteeing the safety of operations such as creating, deleting, and filtering collections and documents. This approach maintains data consistency within collections and across the database.

Document Locking: Document-level locking is implemented to manage access to individual documents within a collection. This fine-grained locking ensures that only one client can modify a document at a time.

Similar to the **CollectionService** class, the **DocumentService** class embraces multithreading by using read and write locks at the database and collection levels. This safeguards document creation, deletion, and updates, ensuring the concurrent management of documents within collections.

Concurrency Control Strategies

Read-Write Locks: The system employs read-write locks to balance concurrency and data integrity. Read locks enable multiple clients to access data simultaneously, while write locks enforce exclusive access during updates, preserving data consistency.

Lock Granularity: The appropriate lock granularity is carefully selected for different scenarios. For example, database-level locks are used when an entire database needs exclusive access, while collection-level and document-level locks are employed for more specific operations.

Conclusion

In conclusion, multithreading and locking are fundamental components of the database system. They enable the achievement of high concurrency, data consistency, and fault tolerance. The carefully designed multithreading model and locking strategies contribute significantly to the system's reliability and robustness, ensuring seamless operation in a distributed environment.

4 Data Consistency Issues in the Database

In any database system, ensuring data consistency is a paramount concern to maintain the integrity and reliability of stored information. The database system developed faces several potential data consistency challenges due to its distributed and concurrent nature. This section elucidates the key data consistency issues encountered in the system and the strategies implemented to address them.

1. Concurrent Write Operations:

- **Issue:** The system allows multiple clients to concurrently perform write operations such as creating or deleting databases, collections, and documents. This concurrency can lead to race conditions, where two or more clients attempt to modify the same data simultaneously.
- **Resolution:** To mitigate this issue, the system utilizes Java's **ReentrantReadWriteLock**. Write locks are employed at various levels, including the database, collection, and document levels. This ensures that only one client can write to a specific resource at a given time, preserving data consistency.

2. Schema Validation:

- **Issue:** When clients create documents within collections, the system verifies if the document adheres to the predefined schema for that collection. Concurrent document creation or updates may lead to schema inconsistencies.
- **Resolution:** The system leverages a schema validation mechanism to ascertain that documents conform to the collection's schema. This validation occurs within a controlled, single-threaded context, thereby maintaining schema consistency within collections.

3. Index Management:

- **Issue:** Indexes, crucial for efficient data retrieval, can suffer from inconsistencies when multiple clients concurrently update or delete documents. Incorrect index entries may affect query results.
- **Resolution:** The system employs locking mechanisms to ensure that index updates and deletions occur atomically. These locks protect the integrity of indexes during concurrent operations.

4. Affinity Management:

- **Issue:** Affinity values assigned to collections are utilized for load balancing and query optimization. Concurrent updates of affinity values may disrupt these optimizations.
- **Resolution:** Affinity values are assigned and updated within a single-threaded context, guaranteeing that modifications are consistent and do not lead to conflicting affinity assignments.

5. Data Retrieval and Filtering:

- **Issue:** Concurrent filtering and retrieval of data may yield inconsistent query results if not managed properly.
- **Resolution:** The system employs read locks during data retrieval and filtering operations to ensure that the data accessed remains consistent during queries.

In summary, the database system has implemented a comprehensive set of mechanisms, including locking, schema validation, and controlled access, to address data consistency issues stemming from concurrent operations. These strategies safeguard data integrity while enabling multiple clients to interact with the database concurrently. Despite the distributed and decentralized nature of the system, a strong focus on data consistency ensures that users can rely on accurate and reliable data storage and retrieval.

5 Node Hashing and Load Balancing

In the context of database system, efficient data distribution, load balancing, and node affinity management are fundamental aspects of maintaining optimal performance and fault tolerance. This section elaborates on how these components are integrated into the system, highlighting their roles and contributions.

Node Hashing

Node affinity is a crucial concept in the database system, influencing data distribution, availability, and performance across the cluster. It determines the preference for storing and processing data on specific nodes within the cluster.

Load Balancing

Load balancing, implemented through the concept of "Collection Affinity," plays a pivotal role in optimizing the utilization of system resources. When a new collection is created, a specific node is designated as the affinity node for that collection. This affinity node assumes responsibility for managing write operations, deletions, and other document-related activities within the collection.

The system dynamically balances the load by assigning collections to different nodes based on their affinity. This approach ensures that write queries to a specific collection are directed to its associated affinity node. Additionally, the affinity node coordinates the distribution of changes made to data within the collection to maintain data consistency across the cluster.

By aligning the concept of "Collection Affinity" with load balancing, the system optimizes resource usage, reduces latency for write queries, and maintains a high level of efficiency during operations.

In summary, the combined strategies of node hashing and load balancing, with a focus on "Collection Affinity," contribute to the database system's robustness, scalability, and overall efficiency. These mechanisms are integral to ensuring reliable data storage, retrieval, and resource utilization in the distributed database infrastructure.

6 Communication Protocols Between Nodes

In a database system, effective communication between nodes is crucial for seamless data exchange, load balancing, and maintaining data consistency across the cluster. This section delves into the communication protocols and strategies employed by the system to ensure robust inter-node communication.

the database system employs a custom communication protocol that builds upon HTTP (Hypertext Transfer Protocol) and adheres to RESTful (Representational State Transfer) principles. This custom protocol facilitates communication between nodes and enables the coordination of data replication and distribution

the communication protocols between nodes in the database system are grounded in RESTful principles and custom-designed to ensure reliable and efficient data exchange. These protocols, underpinned by HTTP and JSON, provide a strong foundation for the distributed nature of the database, allowing nodes to work together seamlessly in maintaining a consistent and responsive data store.

7 Code Testing

Testing is a critical phase in the development of the database system. Rigorous testing ensures that the code functions as intended, maintains data consistency, and can withstand real-world scenarios. I have employed a comprehensive testing strategy that includes unit testing and integration testing.

Unit Testing

Unit testing is conducted at the individual component level to verify that each function, method, or class operates correctly.

Unit Testing

Unit testing is conducted at the individual component level to verify that each function, method, or class operates correctly.

I used **Postman** for testing Postman, a widely recognized API testing tool, has been instrumental in validating the functionality of the RESTful APIs. It allows for easy creation and execution of API requests, ensuring that the endpoints respond correctly to different input scenarios.

Clean Code Principles

In the development of the database system, several Clean Code principles have been thoughtfully applied, contributing to the codebase's quality and maintainability. These principles include:

Descriptive Variable and Function Names: The codebase employs significant variable and function names that serve as self-documenting artifacts. This practice enhances code readability and helps developers understand the purpose of each component without the need for extensive comments.

Modular and Specialized Functions: The code adheres to the principle of creating compact, specialized functions. Each function has a well-defined, single responsibility, making it easier to comprehend, test, and modify. This modular approach enhances code maintainability and minimizes the risk of introducing unintended side effects.

Code Duplication Mitigation: Code duplication is actively minimized. Reusable classes and centralized management structures, such as the **DatabaseManager** class, ensure that common functionality is encapsulated and shared across the codebase. This mitigation of redundancy promotes consistency and simplifies future updates and enhancements.

Defending Code with "Effective Java" Guidelines

In the development of the database system, careful consideration has been given to adhering to several key guidelines outlined in Joshua Bloch's seminal work, "Effective Java." These guidelines are instrumental in producing high-quality, efficient, and maintainable code. Here are three "Effective Java" items that have been applied:

1. **Item 1: Consider Static Factory Methods Over Constructors**: This item advises the use of static factory methods instead of public constructors. In the codebase, classes like **DatabaseManager** and **ClusterManager** provide static factory methods (**getInstance()**) to obtain instances. This approach allows for more flexible instance creation, enables caching, and enhances readability by providing meaningful method names.

2. **Item 9: Prefer Try-with-Resources to Try-Finally**: "Effective Java" encourages the use of try-with-resources to manage resources that require closing, like files or network connections. In the codebase, the **try-catch** blocks in methods like **getIndex()** and **getCollectionsAffinity()** utilize try-with-resources to ensure proper resource cleanup, promoting resource efficiency and robustness.

3. **Item 15: Minimize Mutability**: This item advises limiting the mutability of classes and objects to enhance code reliability. The codebase demonstrates this principle by using immutable classes for various data structures. For instance, **Database** and **Collection** classes are designed with immutable properties, ensuring that once created, their state remains unaltered. This minimizes unexpected side effects and simplifies code reasoning.

SOLID Principles

The SOLID principles are adhered to meticulously, serving as a beacon for software modularity and sustainability. These principles dictate the code architecture, guaranteeing a pliable and extensible system:

- **Single Responsibility Principle (SRP):** Each class and function within the codebase assumes a solitary, well-defined responsibility.
- **Open-Closed Principle (OCP):** The codebase is open for extension but sealed against modification, enabling the addition of new features sans alterations to existing code.
- **Liskov Substitution Principle (LSP):** Subtypes are interchangeable with their base types, ensuring the code operates reliably with derived classes.
- **Interface Segregation Principle (ISP):** Interfaces are customized to fulfill distinct client requirements, mitigating superfluous dependencies.
- **Dependency Inversion Principle (DIP):** Abstractions are preferred over concrete implementations, nurturing flexibility and decoupling.

Design Patterns in the Codebase

The codebase of the database system extensively employs design patterns to address recurring design challenges. These patterns contribute to code reusability, flexibility, and maintainability, ultimately enhancing the system's overall architecture. Notable design patterns observed include:

1. **Singleton Pattern**: The Singleton pattern ensures that a class maintains a single instance throughout the system's lifecycle. It serves as a global access point to this instance. The codebase applies the Singleton pattern in key classes like **DatabaseManager** and **ClusterManager**. This ensures that there is only one instance of these classes, allowing for centralized control and resource management.
2. **Factory Method Pattern**: The Factory Method pattern abstracts the process of object creation, allowing subclasses to define the type of objects that will be instantiated. Within the codebase, static factory methods like **getInstance()** are used in classes such as **DatabaseManager** and **ClusterManager**. These methods facilitate object creation while encapsulating the underlying class details, promoting flexibility.
3. **Locking and Multithreading Patterns**: To guarantee thread safety and efficient resource utilization, the codebase integrates various locking patterns. Reentrant locks, including read and write locks, are employed to coordinate access to shared resources like databases, collections, and documents. These locking patterns prevent issues such as race conditions, data corruption, and thread contention.

4. **Builder Pattern**: The Builder pattern separates the construction of complex objects from their representation. In the codebase, this pattern is employed when creating JSON documents and database schema files. It offers a structured approach to document creation, abstracting the intricacies of the underlying structure.
5. **Observer Pattern**: The Observer pattern facilitates event-driven behavior by enabling objects (observers) to subscribe to and receive updates from other objects (subjects) when their state changes. Within the codebase, various components interact and respond to alterations in database, collection, and document structures. This enhances modularity and adaptability.
6. **Composite Pattern**: The Composite pattern treats individual objects and compositions of objects uniformly. In the codebase, this pattern manages the hierarchical structure of databases, collections, and documents. Regardless of the hierarchy level, it ensures consistent interactions and operations.
7. **Strategy Pattern**: The Strategy pattern allows the selection of algorithms at runtime. In the codebase, strategies are applied for filtering and querying data based on specific criteria. The **Filters** class offers different filtering strategies, enhancing document retrieval efficiency.

The integration of these design patterns underscores the codebase's commitment to functional, robust, and maintainable software. Design patterns not only address recurring design challenges but also promote best practices in software development. They contribute to the creation of a database system that is scalable, adaptable, and well-structured.