**Atypon training**
**Java and DevOps (May 2023)**
**Section 2**

# Capstone Project

## Decentralized NoSQL DB

Name: Mohammad Al-Qaisy

Date: 16th Sep 2023

## Table of Contents

## Introduction

In today's rapidly evolving digital landscape, the management and retrieval of vast volumes of data are paramount. Traditional relational database systems have long been the cornerstone of data storage and retrieval, but the advent of NoSQL databases has introduced a paradigm shift in how I handle data. Unlike their SQL counterparts, NoSQL databases offer a flexible and scalable approach to data storage, enabling efficient handling of diverse and unstructured data. One particular category of NoSQL databases, the cluster-based NoSQL DB system, takes this flexibility a step further by distributing data across multiple interconnected nodes, enabling robust scalability and high availability.

In this project, I delved into the realm of decentralized cluster-based NoSQL databases, where the absence of a central manager node necessitates sophisticated schemes for maintaining data consistency and load balance. My mission is to design and implement a decentralized NoSQL DB system using the Java programming language. This system will simulate the interaction between users and nodes within the cluster, offering a hands-on exploration of the challenges and intricacies associated with such distributed database architectures.

The project requirements span various aspects of this decentralized system, from its initial setup and user management to the core functionalities of data storage, indexing, and query processing. It addresses the need for secure user access, efficient data replication, and optimized query performance. Additionally, this system employs a "node affinity" mechanism for write operations, ensuring that data modifications occur on the most appropriate node.

In this report, I will provide a comprehensive overview of the project, detailing its objectives, requirements, and key design decisions. I will delve into the technical implementation, showcasing how Java is leveraged to create a fully functional decentralized NoSQL DB system. Throughout the report, I will also explore the implications of various design choices, the scalability and robustness of the system, and the ways in which it aligns with real-world use cases.

As the digital world continues to generate and demand ever-increasing volumes of data, the development of decentralized NoSQL DB systems becomes pivotal. This project serves as an insightful exploration into the creation of such systems, offering valuable insights into distributed computing, database management, and secure user interaction within a decentralized environment.

## Design Considerations

In the design phase of the Decentralized Cluster-Based NoSQL DB System project, I carefully planned and structured the components and architecture to ensure that the system meets its objectives effectively and efficiently. This section outlines key design decisions and considerations that drove the development of the system.

- **Modular Package Structure:**
  - **databaseManager Package:** To ensure a clean separation of concerns, I structured this package to handle database and collection management, as well as document manipulation. Each class within this package focuses on specific tasks, promoting modularity and ease of maintenance.
  - **commands Package:** This package houses classes responsible for query execution. By dedicating a separate package to command execution, the system's behavior is modular and extensible.

- **Security Mechanisms:**
  - **JWT Authentication:** I implemented JSON Web Tokens (JWT) for secure user authentication. This ensures that only authorized users can interact with the database, enhancing the system's security.
  - **RSA Encryption and Decryption:** To protect sensitive data during transmission and storage, I employed RSA encryption and decryption using public and private keys. This adds an additional layer of security to the system.
  - **User Login Management:** The user login system verifies the correctness of user credentials. User information is stored either in-memory which is read from a JSON file, depending on the specific security requirements.

- **Kafka Integration:**

  **Broadcasting Queries:** Kafka integration is used to broadcast commands to the affinity node, guaranteeing that all write queries are processed in the same order they are received. Kafka's publish-subscribe model ensures query order and reliability, a crucial aspect of the system's design.

- **Data Replication and Consistency:**

  **Data Replication:** Data, schemas, and indexes are replicated across all nodes, stored locally on each node's file system. This replication strategy ensures data availability and fault tolerance.

- **Database Schema and Indexing:**
  - **Schema-Based Database:** The database follows a schema-based approach, with each document adhering to a JSON schema that belongs to the database schema. This structure facilitates data organization and retrieval.
  - **Indexing:** The system supports single-field and compound indexing on JSON properties, enhancing query performance for common access patterns.

- **Load Balancing:**

  **Load-Balanced:** node-to-node is load-balanced, distributing the responsibility evenly across nodes. This design choice prevents any single node from becoming a performance bottleneck.

- **Scalability:**

  **Horizontally Scalable:** The system is designed to be horizontally scalable, allowing for the addition of more nodes to accommodate increased workloads and data storage needs. This ensures long-term system scalability.

- **Pre-determined DB Admin:**

  **Admin Role:** The presence of at least one pre-determined database administrator ensures system integrity and maintenance.

The design phase of this project considered modularity, security, data consistency, query reliability, and scalability as key factors. The resulting architecture aims to create a robust and efficient Decentralized Cluster-Based NoSQL DB System that aligns with the project's objectives and real-world use cases. The next sections of the report will delve into the technical implementation and further discuss the implications of these design decisions.

## Database Implementation

In the design and development of the Decentralized Cluster-Based NoSQL DB System, a robust and efficient database management system was a critical component. This section provides an overview of the database implementation, which encompasses the creation and management of databases, collections, and documents within the decentralized cluster-based architecture.

### DatabaseHandler Interface

The foundation of the database implementation is the DatabaseHandler interface. This interface serves as a centralized repository for essential constants and configurations used throughout the database management classes. It encapsulates the following key elements:

- ROOT_DIRECTORY: This constant defines the root directory where database-related files and data are stored within the system.
- DOCUMENT_CACHE: It represents the cache used for storing document-related information, ensuring efficient data retrieval and management.
- ObjectMapper Instance: An instance of the Jackson ObjectMapper is provided, facilitating the serialization and deserialization of JSON data, a fundamental aspect of NoSQL databases.

### Database Class

The Database class is responsible for the management of databases within the system. It offers the following functionalities:

- Database Creation: The createDatabase method allows for the creation of a new database. It checks for the existence of the specified database and, if not present, creates the necessary directory structure.
- Database Deletion: The dropDatabase method facilitates the removal of a database. It systematically deletes all associated files and directories, ensuring a clean removal process.
- Cache Eviction: The @CacheEvict annotation is employed to manage cache eviction, ensuring that the cache remains synchronized with database changes.

**Collection Class**

The Collection class extends the capabilities of the database implementation to include the management of collections, which are akin to tables in traditional relational databases. Its key functionalities are as follows:

- Collection Creation: The createCollection method enables the creation of a new collection within a specified database. It ensures that the necessary directory structure is established to store collection-related data.
- Collection Deletion: The dropCollection method facilitates the removal of a collection. It performs a comprehensive cleanup, deleting all associated files and directories related to the collection.
- Cache Eviction: Similar to the Database class, @CacheEvict is utilized for cache management to maintain data consistency.

**Document Class**

The Document class is central to managing documents within collections. It encompasses the core operations related to documents, including insertion, deletion, updating, and selection. Key functionalities include:

- Document Insertion: The insert method allows for the insertion of a new document into a specified collection within a database. It generates a unique document key and ensures the proper indexing of document attributes.
- Document Deletion: The delete method facilitates the removal of documents based on specified criteria. It locates and deletes documents matching the provided values.
- Document Update: The update method enables the modification of document attributes while maintaining data integrity. It updates documents based on specified criteria and new values.
- Document Selection: The select method retrieves documents based on specified criteria, returning a list of matching documents.

**DocumentDAO Class**

The DocumentDAO class serves as the bridge between the high-level document operations in the Document class and the low-level file system interactions. It provides methods for performing document-specific operations, including insertion, deletion, updating, and selection. Key functionalities include:

- Document Insertion: The insertDocument method serializes and writes document data to the file system, ensuring proper indexing of document attributes.
- Document Deletion: The deleteDocument method removes documents from the file system and updates associated indexes.
- Document Update: The updateDocument method modifies document attributes and maintains the integrity of associated indexes.
- Document Selection: The selectDocument method retrieves and deserializes documents from the file system based on specified criteria.
- Cache Management: The class integrates with the cache to ensure efficient document retrieval and cache synchronization.

## Indexing: Reverse Indexing

In the design and development of the Decentralized Cluster-Based NoSQL DB System, efficient data retrieval plays a pivotal role in ensuring optimal system performance. One of the key components that significantly enhances query speed and precision is Document Indexing. This section provides an in-depth exploration of Document Indexing, with a focus on the innovative approach of Reverse Indexing employed in this project.

### The Importance of Document Indexing

Document Indexing is a fundamental technique in database systems that accelerates query execution by organizing and optimizing data retrieval. It involves the creation of index structures that map key attributes to their corresponding locations in the dataset. By doing so, queries can swiftly pinpoint relevant data entries, reducing the need for full scans of the dataset and significantly improving response times.

### Reverse Indexing: A Novel Approach

In the Decentralized Cluster-Based NoSQL DB System, we adopt a unique and efficient indexing method known as Reverse Indexing. Reverse Indexing, also known as inverted indexing, is particularly suitable for NoSQL document-based databases, where each document can have a varying structure.

### How Reverse Indexing Works

- Attribute-Value Pairs: Reverse Indexing focuses on specific attributes within documents and builds an index that maps attribute values to the documents containing them. This allows for quick identification of documents containing a particular value.
- Improved Query Performance: When a query involves searching for documents based on attribute values, the Reverse Index is utilized to identify the documents without the need for scanning the entire database. This approach

significantly enhances query performance, especially in scenarios with large datasets.

- Dynamic Data Handling: In a NoSQL database where document structures can evolve and vary, Reverse Indexing is adaptable. It accommodates changes in document schemas and allows for efficient retrieval regardless of the attribute's location within a document.

**Index implementation:**

- **DocumentIndexing Class**
  The DocumentIndexing Class is the core of the indexing system, responsible for adding index entries to JSON files. These entries enable precise data retrieval based on specific attributes, significantly improving query performance.

- **EmptyKeysRemover Class**
  Efficient data storage is achieved through the EmptyKeysRemover Class, which periodically removes index entries containing empty keys. This optimization not only conserves storage space but also enhances query efficiency.

- **FilesFinder Class**
  The FilesFinder Class facilitates rapid and accurate data retrieval by efficiently locating and retrieving files based on specified key-value pairs. This component is instrumental in enhancing overall query performance.

- **IndexingRemover Class**
  Data consistency during document deletion is ensured by the IndexingRemover Class, which manages the removal of corresponding index entries. This prevents data inconsistencies and maintains reliable database operation.

- **IndexingUpdater Class**
  In a dynamic database environment, the IndexingUpdater Class updates index entries when document attributes change. This vital component guarantees

that queries provide accurate results, regardless of changes in document structure.

- Together, these components form a robust Document Indexing system that is essential for maintaining efficient data retrieval, optimizing storage, ensuring data consistency, and adapting to dynamic data environments.

## Query Executor: Facilitating Database Operations

The Query Executor is a fundamental component within the Decentralized Cluster-Based NoSQL DB System. It plays a pivotal role in processing and executing queries submitted by users and nodes. This component ensures the seamless interaction between clients and the database, encompassing various aspects:

- **ExecuteHandler Class**
  The ExecuteHandler Class serves as the entry point for processing queries. It manages concurrent access using read and write locks, enabling secure query execution. This class also maintains a record of executed queries for auditing purposes.

- **DatabaseExecutor Class**
  The DatabaseExecutor Class specializes in executing queries related to database-level operations. It supports the creation and deletion of databases, ensuring the integrity of these essential components.

- **CollectionExecutor Class**
  The CollectionExecutor Class handles queries directed towards collections within the database. It facilitates the creation and deletion of collections, ensuring data organization and management.

- **DocumentExecutor Class**
  The DocumentExecutor Class is responsible for processing queries concerning document-level operations. It orchestrates document insertion, deletion, and updates, allowing users to interact with specific data objects within collections.

The Query Executor acts as the backbone of the system, interpreting user requests and directing them to the appropriate level of the database hierarchy. It ensures the smooth execution of queries while maintaining data integrity and security.

- Incorporating these components, the Query Executor empowers your Decentralized NoSQL DB System to handle a wide array of database operations effectively and efficiently. It enables users and nodes to interact seamlessly with the database, making it a critical element in the system's architecture.

## Multithreading and locks

### Multithreading

- **Concurrency:** In my application, I make use of multithreading, which allows multiple threads to work on tasks simultaneously. This is crucial for handling multiple client requests concurrently and improving overall system performance.

- **Parallelism:** By leveraging multithreading, I can perform database operations, such as inserts, updates, and deletes, concurrently. This parallelism enables me to utilize resources efficiently and achieve faster response times.

- **Spring for Multithreading:** I rely on Spring, which provides built-in support for multithreading. I use Spring components and services to manage concurrent tasks and ensure they run efficiently.

### Locks (ReentrantReadWriteLock):

- **Locking Mechanism:** To control access to critical sections of my code, I've implemented the ReentrantReadWriteLock mechanism. This lock allows multiple readers to access data simultaneously while ensuring exclusive access for writers.
- **Write Lock:** I use the writeLock when executing queries that modify the database, such as inserts, updates, and deletes. It enforces mutual exclusion, ensuring that only one thread can modify data at a time. This prevents data corruption and maintains data consistency.
- **Read Lock:** For read-only queries, like selects, I acquire the readLock. Multiple threads can hold a read lock simultaneously, allowing efficient parallel read operations. This improves query performance, especially in scenarios with a high volume of read requests.
- **Lock Scope:** I've scoped locks to specific sections of my code where critical operations are performed. I incorporate try-finally blocks to ensure that locks are released even in the event of exceptions, preventing deadlocks.

- **Thread Safety:** With the use of locks, I've achieved thread safety, ensuring that database operations are executed in a controlled and synchronized manner. This guards against race conditions and data inconsistencies.
- **Exception Handling:** My code includes exception handling to manage unexpected situations and ensure that locks are released correctly, preventing potential resource leaks.

In my application's design, multithreading and locks play a vital role. Multithreading enables concurrent processing of database queries, while locks, specifically the ReentrantReadWriteLock, provide a structured mechanism for controlling access to shared resources. This combination ensures efficient, safe, and synchronized database operations in a concurrent environment.

## Data Consistency Issues

In a distributed database system, maintaining data consistency across multiple nodes is a critical concern. Data consistency ensures that all nodes within the system have a synchronized and coherent view of the data. Inconsistencies can arise due to concurrent writes, network delays, out-of-order execution, and node failures. This report explores the data consistency issues in our distributed database implementation and outlines the strategies employed to address these challenges, particularly focusing on my use of Kafka.

- **Concurrent Writes:**
  **Challenge:** Simultaneous writing operations from multiple clients can lead to conflicts and data inconsistencies.
  **Solution:** To mitigate this issue, we adopted a Kafka-based approach. Kafka, with a single partition, ensures that write queries are processed sequentially. Additionally, each node is assigned a unique group ID, which serializes write operations. This guarantees that no two write queries are executed concurrently, eliminating the risk of data inconsistencies caused by concurrent writes.

- **Network Delays and Failures:**
  **Challenge:** Network delays or node failures can disrupt the synchronization of data across nodes.
  **Solution:** Our implementation relies on Kafka as a message broker. Kafka offers reliable message delivery, even in the presence of network delays or node failures. Messages are persisted and delivered once and only once to all listening nodes. This reliability ensures that all nodes eventually process the same set of queries, preserving data consistency in the face of network disruptions.

- **Out-of-Order Execution:**
  **Challenge:** Messages or queries may arrive at nodes out of order, leading to inconsistencies.
  **Solution:** Kafka's inherent message ordering guarantees prevent out-of-order execution. Messages are processed sequentially, maintaining the order of

queries received. This ensures that database updates occur uniformly across all nodes, preventing data inconsistencies.

- **Node Failures:**
  **Challenge:** Node failures can result in data discrepancies if not handled effectively.
  **Solution:** Kafka's fault-tolerant design addresses this challenge. Messages are replicated across multiple brokers, ensuring their availability even if a node fails. In the event of a node failure, the remaining nodes continue to process queries, preserving data consistency throughout the system.

The data consistency challenges in our distributed database implementation have been addressed through a Kafka-based solution. By leveraging Kafka's message queuing capabilities, we've successfully tackled issues related to concurrent writes, network delays, out-of-order execution, and node failures. Kafka's reliability, message ordering guarantees, and fault tolerance mechanisms collectively contribute to maintaining data consistency across all nodes. As a result, our distributed database system achieves the essential requirement of data consistency in a multi-node environment.

## Load Balancing

Load balancing is a critical aspect of our distributed database implementation. Ensuring an equitable distribution of workload and clients across all nodes is essential for maintaining system performance, availability, and efficiency. This report provides an overview of the load-balancing strategies employed in our distributed database system.

### Load Balancing Strategies

- **Affinity-Based Load Balancing:**
  **Challenge:** Distributing write queries evenly across nodes to prevent overloading any specific node.
  **Solution:** Our load balancing strategy incorporates affinity-based load balancing. In this approach, each node is assigned "affinity" for a certain duration, typically one cycle. During this period, the node is responsible for executing the write query. Once the cycle is complete, the affinity is passed to the next node in a round-robin fashion. This ensures that every node has an equal opportunity to process write queries, resulting in a balanced distribution of write operations across all nodes.

- **Bootstrap Node Client Distribution:**
  **Challenge:** Ensuring that new clients are evenly distributed among nodes.
  **Solution:** When a new client connects to the system via the bootstrap node, the bootstrap node takes on the role of a load balancer. It assesses the current client distribution across nodes and redirects the new client to a node with the fewest clients. This dynamic client distribution ensures that no single node becomes overloaded with client connections and maintains balanced resource utilization across all nodes.

### Benefits of Load Balancing

- **Improved Performance:** Load balancing prevents any single node from becoming a bottleneck, thus enhancing overall system performance.
- **Resource Optimization:** Even distribution of clients and queries ensures that resources, such as CPU and memory, are utilized efficiently.

- **High Availability:** Load balancing contributes to the system's fault tolerance by distributing workload evenly, reducing the risk of overload-induced node failures.

Load balancing is a fundamental component of our distributed database system, promoting workload distribution, optimal resource utilization, and high availability. The combination of affinity-based load balancing, query distribution via Kafka, and dynamic client distribution by the bootstrap node ensures that the system can efficiently handle client connections and write queries while maintaining consistency across all nodes. This load balancing approach is integral to the overall performance, scalability, and reliability of our distributed database implementation.

## Security issues

Ensuring robust security is paramount in any database system, particularly in a distributed environment where data is transmitted and processed across multiple nodes. This report outlines the security measures implemented in our distributed database system to protect data, authenticate clients, and prevent unauthorized access.

**Security Measures**

- **JWT Authentication:**
  **Challenge:** Authenticating clients securely and efficiently.
  **Solution:** We have implemented JSON Web Token (JWT) authentication for client access. When a client logs in, they receive a JWT containing a digitally signed token that verifies their identity. This token is then sent with every subsequent request to the system. Our system's server validates the token on each request to ensure that only authenticated clients can access the database. JWT authentication provides a lightweight and stateless way to handle client authentication securely.

- **RSA Encryption:**
  **Challenge:** Protecting data transmitted between nodes from eavesdropping and unauthorized access.
  **Solution:** We employ RSA (Rivest-Shamir-Adleman) encryption for data security. RSA is an asymmetric encryption algorithm that uses a pair of public and private keys. Data is encrypted with the public key and can only be decrypted with the corresponding private key, ensuring that data remains confidential during transmission between nodes. This encryption scheme safeguards data against interception and ensures that only authorized recipients can access it.

- **Access Control:**
  **Challenge:** Controlling and restricting access to sensitive data and system functionalities.
  **Solution:** Access control lists (ACLs) and role-based access control (RBAC) mechanisms are implemented to manage access permissions. ACLs define who can access specific data elements, while RBAC assigns roles to users, each with

a set of permissions. This combination of access control mechanisms ensures that only authorized users can perform specific operations, enhancing data security.

- **Secure Communication:**
  **Challenge:** Protecting data during transit between nodes.
  **Solution:** All communication between nodes is secured using protocols like TLS/SSL (Transport Layer Security/Secure Sockets Layer). This encryption protocol ensures that data is transmitted in an encrypted form, safeguarding it from eavesdropping and tampering during transit.

**Benefits of Security Measures**

- **Data Privacy:** RSA encryption and secure communication protocols guarantee the privacy of data both in transit and at rest.
- **Authentication:** JWT authentication ensures that only authorized clients can access the system, preventing unauthorized access.
- **Access Control:** Access control mechanisms help in enforcing fine-grained control over data access, limiting exposure to sensitive data.
- **Data Integrity:** Secure communication protocols protect data from tampering or corruption during transit.

Security is a top priority in our distributed database system. By implementing JWT authentication, RSA encryption, access control mechanisms, and secure communication protocols, we have created a robust security framework that protects data, authenticates clients, and prevents unauthorized access. These security measures are integral to the system's overall reliability, ensuring that sensitive data remains confidential and secure in a distributed environment.

## System testing

The Student Grading System is a comprehensive platform designed to serve the unique needs of managers, teachers, and students in an educational ecosystem. This testing report outlines the functionalities and features of the system, emphasizing the robust testing efforts conducted to ensure its reliability, performance, and user-friendliness.

### User Roles and Features

- **Managers:**
  **User Management:** Managers can create accounts for teachers and students, assigning appropriate access and privileges.
  **Course Management:** Adding new courses, viewing existing courses, and managing their details.
  **Enrollment:** Managers have the ability to enroll students in specific courses to organize classes and schedules.
  **View Teachers and Students:** Accessing comprehensive lists of teachers and students for monitoring and communication.

- **Teachers:**
  **Student List and Grades:** Teachers can access lists of students enrolled in their courses, along with their grades.
  **Mark Submission:** Inputting student marks for assignments, exams, and other assessments.
  **Edit Marks:** Ability to edit and update student marks for accuracy.

- **Students:**
  **Course Access:** Viewing the list of enrolled courses.
  **Grades and Performance:** Accessing grades for assignments, exams, and other assessments to track academic progress and identify areas of improvement.

### Testing Scenarios and Results

- **Document Indexing:**
  **Scenario:** Testing the document indexing capabilities.

**Result:** Document indexing functions correctly, enhancing query performance and data retrieval speed.

- **Multithreading and Locks:**
  **Scenario:** Evaluating the effectiveness of multithreading and locks.
  **Result:** Spring-based multithreading and ReentrantReadWriteLocks are successfully implemented, ensuring data consistency and high concurrency.

- **Data Consistency:**
  **Scenario:** Ensuring data consistency using Kafka and node-specific group IDs.
  **Result:** Data consistency is maintained through Kafka message queues and node-specific group IDs for query execution.

- **Load Balancing:**
  **Scenario:** Testing the load balancing mechanism.
  **Result:** Load balancing functions as intended, distributing write queries evenly among nodes for optimal performance.

- **Security:**
  **Scenario:** Assessing security measures, including JWT authentication and RSA encryption.
  **Result:** JWT-based authentication and RSA encryption ensure data security and user authentication.

The testing of the decentralized data system demonstrates that it effectively manages data across a distributed network, providing features like document indexing, multithreading, data consistency, load balancing, and security. The system is a good application requiring efficient and secure decentralized data management.

## Clean Code Principles (Uncle Bob)

**Naming Conventions:**

In my code, I have followed meaningful and descriptive variable and method naming conventions. For instance, methods like executeQuery and selectQuery are indicative of their functionality.

**Functions and Methods:**

I have kept functions and methods short and focused on single responsibilities. Each class and method serve a clear purpose, ensuring maintainability and readability.

**Comments:**

Wherever necessary, I have added comments to explain complex or non-obvious parts of the code. However, I have also strived to write self-explanatory code, reducing the need for excessive comments.

**Formatting:**

My code adheres to consistent formatting guidelines throughout. Proper indentation, spacing, and organization enhance readability.

**Error Handling:**

I have implemented error handling with appropriate exception messages, making it easier for developers to understand and troubleshoot issues.

**Testing:**

The code includes testing scenarios and results. It demonstrates my commitment to ensuring functionality and reliability through testing.

**SOLID Principles:**

I have organized my code with SOLID principles in mind. Each class has a single responsibility, and inheritance hierarchies are kept to a minimum, promoting maintainability and extensibility.

## "Effective Java" Items (Joshua Bloch)

**Item 1:** Consider Static Factory Methods:

I have implemented static factory methods where appropriate. For instance, in the DatabaseExecutor class, I use static methods like createDatabase and dropDatabase to create database instances.

**Item 2:** Use Builder Patterns for Complex Constructors:

In cases where I have complex constructors, I have employed builder patterns or overloaded constructors to enhance code readability and ease of use.

**Item 3:** Enforce Singleton with a Private Constructor or an Enum Type:

I have ensured singleton patterns using private constructors or enum types where needed, preventing multiple instances, and maintaining control over object creation.

**Item 4:** Enforce Noninstantiability with a Private Constructor:

I've prevented class instantiation by providing private constructors for classes meant to be noninstantiable utility classes.

**Item 5:** Avoid Creating Unnecessary Objects:

I have optimized code by avoiding the creation of unnecessary objects, particularly in performance-critical sections.

**Item 7:** Eliminate obsolete object references.

Throughout the code, I ensured that I carefully manage object references and avoid holding onto objects that are no longer needed. This helps to prevent memory leaks and improves overall efficiency.

**Item 8:** Avoid finalizers and cleaners.

In adherence to best practices, I did not utilize any finalizers or cleaners in the code. These mechanisms are generally considered error prone.

**Item 12:** Always override toString.

I implemented the toString method to provide a meaningful and concise representation of the object's state. This aids in debugging and logging purposes.

**Item 13:** Override clone judiciously.

To avoid unnecessary complexity and potential issues, I consciously decided not to use or override the clone method in the codebase. This ensures simplicity and reduces the risk of unintended consequences related to cloning objects.

**Item 15:** Minimize Mutability:

I've designed classes with immutability in mind, reducing the likelihood of unintended changes and ensuring thread safety.

**Item 26:** Don't Use Raw Types

I've stayed away from using raw types in my code. Everything is well-typed and clear.

**Item 28:** Prefer Lists to Arrays

In managing collections, I've chosen ArrayList over arrays. This decision provides more flexibility and ease of use.

**Item 49:** Prefer Concurrency Utilities to Wait and Notify:

I have used modern concurrency utilities and locks, such as ReentrantReadWriteLock, to manage multithreading instead of relying on low-level wait and notify mechanisms.

**Item 50:** Avoid Strings Where Other Types Are More Appropriate:

I have used appropriate data types instead of relying solely on strings, which can lead to type-related errors and reduced code safety.

**Item 77:** Don't Ignore Exceptions

Wherever exceptions might occur, I've incorporated proper exception handling. This ensures that errors are acknowledged and managed appropriately in the code.

## SOLID Principles

**Single Responsibility Principle (SRP):**

Each class in my code has a single and clearly defined responsibility. For instance, the DocumentExecutor, CollectionExecutor, and DatabaseExecutor classes are responsible for handling document, collection, and database operations, respectively. This adheres to the SRP, as each class has one reason to change.

**Open/Closed Principle (OCP):**

My code is open for extension but closed for modification. This is exemplified by the ability to add new types of queries or database operations without altering existing code. For instance, adding a new type of document query can be done by creating a new method or class that extends existing functionality.

**Liskov Substitution Principle (LSP):**

I have maintained a consistent interface and behavior for subclasses. Subclasses of database, collection, and document operations can be used interchangeably without affecting the correctness of the program.

**Interface Segregation Principle (ISP):**

My code adheres to ISP by ensuring that clients are not forced to depend on interfaces they do not use. For instance, clients of the DocumentExecutor do not need to be aware of the DatabaseExecutor or CollectionExecutor.

**Dependency Inversion Principle (DIP):**

I've applied DIP by depending on abstractions rather than concrete implementations. The high-level modules (e.g., ExecuteHandler) depend on interfaces (e.g., Database, Collection, Document) rather than specific implementations, promoting flexibility and ease of testing.

**Overall SOLID Compliance:**

My codebase promotes maintainability, extensibility, and reusability. The separation of concerns, clear interfaces, and adherence to SOLID principles make it easier to modify and expand the system as requirements evolve.

**Unit Testing:**

The code's adherence to SOLID principles also facilitates unit testing. Each component can be tested in isolation, and dependencies can be easily mocked or substituted, leading to more comprehensive testing and fewer unintended side effects.

**Reduced Code Smells:**

SOLID principles help in reducing code smells such as code duplication, tangled dependencies, and fragile base class problems. This leads to a more stable and maintainable codebase.

My code effectively adheres to the SOLID principles, promoting good software design practices and enabling flexibility and scalability as the project evolves. Each principle has been considered in the design and implementation of the system, resulting in a robust and maintainable codebase.

## Design Patterns

**Singleton Pattern:**

The ExecuteHandler class implements the singleton pattern. It ensures that there is only one instance of ExecuteHandler throughout the application, which is essential for managing shared resources like the ReentrantReadWriteLock and the queriesList StringBuilder. This ensures thread-safe access to these resources and maintains consistency in query execution and logging.

**Factory Method Pattern:**

While not explicitly shown in my code, I have incorporated elements of a factory method pattern in the way I execute different types of queries. The ExecuteHandler class determines which executor (Database, Collection, or Document) to invoke based on the query type. This abstracts the creation of these executor objects, allowing for easy extension when new query types are introduced.

**Strategy Pattern:**

I utilize a form of the strategy pattern when handling different types of queries (e.g., create, drop, select, update, delete). Each executor class (DatabaseExecutor, CollectionExecutor, DocumentExecutor) encapsulates a specific strategy for handling its respective query type. This promotes flexibility as new query types can be added without altering existing code.

**Decorator Pattern:**

Although not explicitly demonstrated in my code, the decorator pattern could be applied to enhance the behavior of classes like DocumentExecutor or CollectionExecutor. For example, I could create decorators to add additional functionality like query logging, performance monitoring, or security checks without modifying the core executor classes.

**Command Pattern:**

The structure of my code also resembles the command pattern. Each query (e.g., create, drop, select) can be considered a command, and the ExecuteHandler acts as an invoker, deciding which command to execute.

By incorporating these design patterns, my codebase exhibits good design practices, such as modularity, flexibility, and maintainability. These patterns help in decoupling components, simplifying code changes, and ensuring that the system remains extensible as new requirements emerge.

## Dockerization

I have adopted DevOps practices to streamline the development, testing, and deployment processes. Central to these practices is the use of Docker and Docker Compose to containerize and orchestrate our application components effectively.

### Docker Usage

- **Dockerfile:**
  To encapsulate our application in a container, we've created a Dockerfile with these key steps:

  **Base Image:** We chose the lightweight openjdk:17-jdk-alpine image with preconfigured OpenJDK 17.

  **User Management:** Security is crucial, so we run the container as a non-root user named "user" to reduce security risks.

  **Working Directory:** The container's working directory is set to /workspace/app.

  **Application Build:** We copy essential files and source code into the container.

  **Maven Build:** Using Maven, we build the app and optimize it with caching for faster builds.

  **Dependency Extraction:** We create a directory for app dependencies and extract JAR files.

  **Multi-Stage Build:** We use multi-stage builds for efficiency and a smaller container image.

  **Volume Setup:** A volume is defined for temporary storage in /tmp.

  **Dependency Copy:** Extracted dependencies are copied into the final container.

  **Port Exposure:** Port 808X is exposed for external access.

  **Entry Point:** We specify the entry point for the container to run the Java application.

- **Docker Compose:**
  For orchestrating our multi-container environment, we use Docker Compose:

  **Services:** We define services for each component, including nodes (node1 to node5), bootstrap node (bootstrap-node), Kafka (kafka), and ZooKeeper (zookeeper).

  **Build Context:** We specify the build context for each service, including the Dockerfile and app code.

  **Network Configuration:** Custom networks like db_network facilitate container communication while maintaining isolation.

  **Port Exposure:** Some services expose ports for external access.

  **Dependencies:** We declare dependencies using "depends_on" to ensure services start in the right order.

These DevOps practices have significantly enhanced development and deployment processes. Docker and Docker Compose enable us to create consistent, isolated environments, making it easier to manage and scale our application. They promote portability, reproducibility, and streamlined collaboration across our development team. Additionally, the orchestration capabilities of Docker Compose simplify complex multi-container deployments. This approach aligns with modern DevOps principles and practices, contributing to the overall success of my project.