Capstone Project

Decentralized Cluster-Based NoSQL DB System

Made by: Samer AlHamdan

Instructors:

Motasim Aldiab

Fahed Jubair

1. Introduction

This report offers a comprehensive analysis for the Decentralized Cluster-Based NoSQL Database System. during Atypon's Training in May 2023. It provides an in-depth overview of my NoSQL cluster database implementation, focusing on key aspects such as data structures, multithreading with locking mechanisms, data consistency, node hashing for load balancing, communication protocols between nodes, security considerations, code testing, alignment with Clean Code principles by Uncle Bob, adherence to "Effective Java" items by Joshua Bloch, compliance with SOLID principles, utilization of design patterns, and integration of DevOps practices.

2. The Assignment

NoSQL databases are a type of database that stores data differently from traditional relational databases. They are often more scalable and flexible, making them well-suited for big data applications.

A decentralized cluster-based NoSQL DB system is a NoSQL database that is distributed across multiple nodes. This means that the data is not stored in a single location, but rather on each node in the cluster. This makes the database more resilient to failures and more scalable than a centralized database.

In this assignment, I will build a simulation of a decentralized cluster-based NoSQL DB system in Java. The simulation will allow users to interact with the database by sending queries and receiving results. The database will be implemented as a document-based database, using JSON objects to store documents.

The simulation will need to implement the following features:

- **Bootstrapping:** The system must have a bootstrapping step to start up the cluster and initiate all nodes. The bootstrapping node will provide initial configuration information to all nodes so that they may communicate and function correctly.
- **Queries:** Users should be able to send queries to the database to create, read, update, and delete documents.

- **Data replication:** Data, schemas, and indexes should be replicated across all nodes in the cluster.
- **Read queries:** Read queries may be satisfied by any node in the cluster.
- **Write queries:** Write queries have node affinity, meaning that they must be processed by the node that has affinity for the document being written to.
- **Document-to-node affinity:** Document-to-node affinity must be load-balanced.
- **Optimistic locking:** Use optimistic locking to resolve race conditions when multiple writes to the same JSON property occur simultaneously.
- **JSON property indexing:** Implement your own JSON property indexing scheme, rather than using an existing library.

Finally, I must create a demo application that utilizes the database. This demo application may be either a command-line application or a web application.

3. Database implementation

I will be talking about my database in depth of each class and how they work together

3.1 Database

1. `createDatabase(String databaseName)`:
    - This method is used to create a new database with the given name.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabaseLock()`: This is a lock (ReadWriteLock) obtained from the `DatabaseManager` to ensure thread safety while creating a database.
        b. `FileManager.fileExists(FileManager.storagePath + "/" + databaseName)`: This checks if a directory with the provided database name already exists.
        c. `DatabaseManager.getInstance().getDatabases()`: This is a HashMap that stores information about databases in the system, where the key is the database name and the value is the `Database` object.

        d. `Database`: An instance of the `Database` class is created to represent the newly created database.

- Explanation:
    a. The method first acquires a write lock to ensure exclusive access to the database management data structures.
    b. It checks if a directory with the given database name already exists. If it does, it prints a message and returns, indicating that the database already exists.
    c. If the directory doesn't exist, it creates the database directory using `FileManager.createDirectoryIfNotFound`.
    d. An instance of the `Database` class is created, and the database name is set.
    e. The new database is added to the `DatabaseManager`'s database HashMap.
    f. Finally, the write lock is released.

2. `deleteDatabase(String databaseName)`:
    - This method is used to delete an existing database with the given name.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabaseLock()`: This is a lock (ReadWriteLock) obtained from the `DatabaseManager` to ensure thread safety while deleting a database.
        b. `FileManager.fileExists(FileManager.storagePath + "/" + databaseName)`: This checks if a directory with the provided database name exists.
        c. `DatabaseManager.getInstance().getDatabases()`: This is a HashMap that stores information about databases in the system, where the key is the database name and the value is the `Database` object.

    - Explanation:
        a. The method first acquires a write lock to ensure exclusive access to the database management data structures.
        b. It checks if a directory with the given database name exists. If it doesn't, it prints a message and returns, indicating that the database doesn't exist.
        c. If the directory exists, it uses `FileSystemUtils.deleteRecursively` to delete the database directory and its contents.

   d. If the deletion is successful, it prints a success message and removes the database entry from the `DatabaseManager`.
   e. If the deletion fails, it prints an error message.
   f. Finally, the write lock is released.

3. `getDatabases()`:
   - This method retrieves a list of all database names in the system.
   - Data Structures Used:
     a. `DatabaseManager.getInstance().getDatabaseLock()`: This is a lock (ReadWriteLock) obtained from the `DatabaseManager` to ensure thread safety while reading the database list.
     b. `DatabaseManager.getInstance().getDatabases()`: This is a HashMap that stores information about databases in the system, where the key is the database name and the value is the `Database` object.
   - Explanation:
     a. The method acquires a read lock to ensure multiple threads can read the database list concurrently.
     b. It retrieves the keys (database names) from the `DatabaseManager`'s database HashMap and converts them to a list using Java Streams.
     c. The read lock is released, and the list of database names is returned.

Overall, the code uses locks to ensure thread safety when creating, deleting, and retrieving databases. It also interacts with a `DatabaseManager` that stores information about databases in a HashMap, allowing efficient database management operations.

3.2 Collection

1. `createCollection(String databaseName, String collectionName, String schema)`:
   - This method is responsible for creating a new collection within a database.
   - Data Structures Used:
     a. `DatabaseManager.getInstance().getDatabases()`
        This HashMap stores information about the databases in the system, allowing efficient retrieval by name.
     b. `DatabaseManager.getInstance().getDatabases().get(databaseName).getCollectionLock()`
        A read-write lock is used to ensure thread safety when creating collections.
     c. `JsonNode jsonNode = objectMapper.readTree(schema)`
        JsonNode is used to represent the schema of the collection in JSON format.
   - Explanation:
     a. The method first locks the collection associated with the database to ensure exclusive access.
     b. It checks whether the database and collection already exist. If not, it creates the collection directory and initializes its metadata.
     c. The schema provided as JSON is validated, and if it's valid, the schema is written to a schema file.
     d. Affinity is assigned to the collection, and the collection is added to the database.
     e. Finally, the read-write lock is unlocked.
2. `deleteCollection(String databaseName, String collectionName)`:
   - This method is responsible for deleting a collection from a database.
   - Data Structures Used:
     a. `DatabaseManager.getInstance().getDatabases()`
        This HashMap stores information about the databases in the system, allowing efficient retrieval by name.
     b. `DatabaseManager.getInstance().getDatabases().get(databaseName).getCollectionLock()`
        A read-write lock is used to ensure thread safety when creating collections.

- Explanation:
    a. Similar to `createCollection()`, this method locks the collection associated with the database to ensure exclusive access.
    b. It checks whether the database and collection exist. If not, it returns appropriate messages.
    c. If the collection exists, it is recursively deleted along with its contents.
    d. The collection is removed from the database.
    e. The read-write lock is unlocked.

3. `List<String> getCollections(String databaseName)`:
    - This method retrieves the list of collections in a database.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabases()`
           This HashMap stores information about the databases in the system, allowing efficient retrieval by name.
        b. `DatabaseManager.getInstance().getDatabases().get(databaseName).getCollectionLock()`
           A read-write lock is used to ensure thread safety when creating collections.
    - Explanation:
        a. It locks the collection associated with the database for read access.
        b. It retrieves the list of collection names and returns them as a list.
        c. The read-write lock is unlocked.
4. `newIndex(String databaseName, String collectionName, String key)`:
    - This method creates a new index on a specified key in a collection.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabases()`
           This HashMap stores information about the databases in the system, allowing efficient retrieval by name.
        b. `DatabaseManager.getInstance().getDatabases().get(databaseName).getCollectionLock()`
           A read-write lock is used to ensure thread safety when creating collections.

- Explanation:
    a. Similar to other methods, it locks the collection associated with the database for write access.
    b. It calls `Indexing.newIndex()` to create the index.
    c. The read-write lock is unlocked.

5. `List<JsonNode> filterByKey(String databaseName, String collectionName, String key)`:
    - This method retrieves documents from a collection based on a key.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabases()`
           This HashMap stores information about the databases in the system, allowing efficient retrieval by name.
        b. `DatabaseManager.getInstance().getDatabases().get(databaseName).getCollectionLock()`
           A read-write lock is used to ensure thread safety when creating collections.
    - Explanation:
        a. It locks the collection associated with the database for read access.
        b. It retrieves documents based on the specified key using the collection's index.
        c. The documents are returned as a list.
6. `List<JsonNode> filterByValue(String databaseName, String collectionName, String key, String value)`:
    - This method retrieves documents from a collection based on a key-value pair.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabases()`
           This HashMap stores information about the databases in the system, allowing efficient retrieval by name.
        b. `DatabaseManager.getInstance().getDatabases().get(databaseName).getCollectionLock()`
           A read-write lock is used to ensure thread safety when creating collections.

- Explanation:
  a. It locks the collection associated with the database for read access.
  b. It retrieves documents based on the specified key-value pair using the collection's index.
  c. The documents are returned as a list.

Overall, the code uses a combination of HashMaps for efficient data storage and retrieval, ReadWriteLocks to ensure thread safety during database operations, and JSON representation (JsonNode) for managing collection schemas. These data structures are used to provide concurrency control and facilitate various database-related operations.

3.3 Document

1. `createDocument(String databaseName, String collectionName, String jsonDocument, Optional<String> id)`:
   - This method is used to create a new document in a collection within a database.
   - Data Structures Used:
     a. `DatabaseManager.getInstance().getDatabases()`: A HashMap that stores information about databases in the system.
     b. `Database`: An instance of the `Database` class representing the specified database.
     c. `Collection`: An instance of the `Collection` class representing the specified collection.
     d. `DocumentLock` (ReadWriteLock): Obtained from the collection to ensure thread safety while creating a document.
   - Explanation:
     a. The method first acquires a write lock on the document collection to ensure exclusive access while creating the document.
     b. It parses the provided `jsonDocument` into a JSON tree (`JsonNode`) using an `ObjectMapper`.
     c. If an `id` is provided, it uses that as the document's ID; otherwise, it generates a random ID.

    d. The ID is added to the JSON document.
    e. Various checks are performed to ensure the existence of the database, collection, and document.
    f. If the document doesn't already exist, it checks if the provided JSON document follows the schema.
    g. If the document is schema-compliant, it is written to a JSON file, and indexing is performed.
    h. If the document doesn't follow the schema, an error message is printed.
    i. Finally, the write lock is released.

2. `deleteDocument(String databaseName, String collectionName, String documentName)`:
    - This method is used to delete a document from a collection within a database.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabases()`: A HashMap that stores information about databases in the system.
        b. `Database`: An instance of the `Database` class representing the specified database.
        c. `Collection`: An instance of the `Collection` class representing the specified collection.
        d. `DocumentLock` (ReadWriteLock): Obtained from the collection to ensure thread safety while deleting a document.

    - Explanation:
        a. The method first acquires a write lock on the document collection to ensure exclusive access while deleting the document.
        b. Various checks are performed to ensure the existence of the database, collection, and document.
        c. If the document exists, indexing is removed, and the document file is deleted.
        d. If the deletion is successful, the document is removed from the collection's list of documents.
        e. If the deletion fails, an error message is printed.
        f. Finally, the write lock is released.

3. `updateDocument(String databaseName, String collectionName, String documentName, String jsonDocument)`:
    - This method is used to update an existing document in a collection within a database.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabases()`: A HashMap that stores information about databases in the system.
        b. `Database`: An instance of the `Database` class representing the specified database.
        c. `Collection`: An instance of the `Collection` class representing the specified collection.
        d. `DocumentLock` (ReadWriteLock): Obtained from the collection to ensure thread safety while updating a document.
    - Explanation:
        a. The method first acquires a write lock on the document collection to ensure exclusive access while updating the document.
        b. Various checks are performed to ensure the existence of the database, collection, and document.
        c. The method parses the provided `jsonDocument` and merges it with the existing document.
        d. If the updated document follows the schema, the previous index is removed, and the document is updated.
        e. Finally, the write lock is released.
4. `getAll(String databaseName, String collectionName)`:
    - This method retrieves all documents from a collection within a database.
    - Data Structures Used:
        a. `DatabaseManager.getInstance().getDatabases()`: A HashMap that stores information about databases in the system.
        b. `Database`: An instance of the `Database` class representing the specified database.
        c. `Collection`: An instance of the `Collection` class representing the specified collection.

        d. `DocumentLock` (ReadWriteLock): Obtained from the collection to ensure thread safety while reading documents.
  - Explanation:
    a. The method first acquires a read lock on the document collection to allow multiple threads to read documents concurrently.
    b. It retrieves the list of document IDs from the collection and reads each document from the file system.
    c. The documents are stored as a list of `JsonNode` objects and returned.
    d. Finally, the read lock is released.

The code uses locks to ensure thread safety while creating, deleting, updating, and retrieving documents within a collection. It also interacts with various data structures, including the `DatabaseManager`, to manage databases and collections efficiently.

3.4 Communicating with the Bootstrapper

`pingBootstrapperForInitData()`:

- This method is responsible for sending a POST request to a bootstrapper service to initialize data.
- Explanation:
  a. The method first retrieves the bootstrapper's port from the environment variable "Bootstrapper_Port" and stores it in `bootstrapperPort`.
  b. It initializes an `HttpHeaders` object `headers` and sets the content type to JSON.
  c. An `ObjectMapper` is used to create a JSON object `jsonNode` with a "port" field containing the value from the "Node_Port" environment variable.
  d. The `pingBootStrapperUrl` variable is constructed with the bootstrapper's port.
  e. An `HttpEntity` named `request` is created, which includes the JSON data and headers.
  f. Finally, a POST request is sent to the bootstrapper's URL using the `RestTemplate`, and the response (if any) is discarded.

3.5 Start Up

1. `@PostConstruct` annotation:
    - This annotation is used on a method that needs to be executed after dependency injection is done to perform any initialization.
    - Explanation:
        a. The `init` method is executed after the component is constructed and clears the existing databases.
        b. It then calls the `loadDatabases` method to load databases from the "storage" directory.
        c. It includes a `Thread.sleep` to pause execution for 2 seconds.
2. `loadDatabases(String dir)`:
    - This method loads databases from the specified directory.
    - Data Structures Used:
        a. `File[] db`: An array of `File` objects representing subdirectories in the specified directory.
    - Explanation:
        a. It checks if the specified directory exists and returns if it doesn't.
        b. It lists the subdirectories in the directory and iterates through them.
        c. For each subdirectory (which represents a database), it calls the `processDatabase` method.
3. `processDatabase(File file)`:
    - This method processes a database directory.
    - Data Structures Used:.
        a. `File[] col`: An array of `File` objects representing subdirectories (collections) in the database directory.
    - Explanation:
        a. It extracts the name of the database from the directory name.
        b. It creates a `Database` object and adds it to the `DatabaseManager`.
        c. It lists subdirectories (collections) in the database directory and calls the `processCollection` method for each collection.

4. `processCollection(File coll, String databaseName)`:
    - This method processes a collection directory within a database.
    - Data Structures Used:
        a. `String collectionName`: A string to store the name of the collection.
    - Explanation:
        a. It extracts the name of the collection from the directory name.
        b. It creates a `Collection` object and adds it to the database's collections.
        c. It calls the `loadDocuments` method to load documents for the collection.
        d. It calls the `loadIndex` method to load the index for the collection.
5. `loadIndex(String databaseName, String collectionName)`:
    - This method loads the index for a collection within a database.
    - Explanation:
        a. It reads the index data from a JSON file using the `FileManager.getDocument` method.
        b. It parses the index data and populates the index data structure in `DatabaseManager`.
6. `loadDocuments(File coll, String databaseName, String collectionName)`:
    - This method loads documents for a collection within a database.
    - Data Structures Used:
        a. `File[] doc`: An array of `File` objects representing document files.
    - Explanation:
        a. It lists document files within the collection directory.
        b. For each document file, it extracts the document name and processes it.
        c. It also handles special cases for "affinity" documents and ignores "schema" and "index" documents.
7. `run(ApplicationArguments args)`:
    - This method is called automatically after the Spring application context is fully initialized. It serves as a hook to execute custom logic at startup.

    - Explanation:
        a. In the `run` method, it invokes the `pingBootstrapperForInitData` method of the injected `bootstrappingService`. This method likely sends a request to a bootstrapper service to initialize data, but the details of this functionality are specific to the `BootstrappingService` class.

In the end this is how my code starts up before and after the Spring application context is fully initialized. In the case of the database system, this involves invoking the BootstrappingService class to get data from the bootstrapper about other nodes, and other methods are responsible for loading databases, collections, indexes, and documents from the storage directory. It uses various data structures to organize and manage the loaded data.

3.6 Shut Down

1. `@PreDestroy` annotation:
    - This annotation indicates that the `destruct` method should be executed when the Spring application context is being destroyed, typically during application shutdown.
2. `destruct()`:
    - This method is called automatically during application shutdown. It serves as a hook to execute custom shutdown logic.
    - Explanation:
        a. In the `destruct` method, it calls the `loadDatabases` method, passing the directory path ".\storage" as an argument.
3. `loadDatabases(String dir)`:
    - This method takes a directory path as an argument and is responsible for saving index and affinity data for databases and collections during shutdown.
    - Explanation:
        a. It lists the files and directories in the specified directory path.
        b. For each subdirectory (representing a database), it processes its collections.

c. For each collection, it:
   i. Retrieves the index and affinity data from the corresponding data structures in `DatabaseManager`.
   ii. Serializes the index and affinity data to JSON strings.
   iii. Reads the JSON strings as JSON nodes.
   iv. Writes the JSON nodes to files named "index" and "affinity" within the collection's directory using ObjectMapper.
   v. This process effectively saves the index and affinity data for each collection and database to disk for future use.

Overall, the `ShutdownManager` class is responsible for saving index and affinity data to disk when the application is being shut down. This ensures that the data is persisted and can be restored when the application is restarted.

3.7 Managers

1. **AffinityManager**:
   - `getNextAffinity()`: This method returns the next affinity value based on an atomic counter. Affinity is used to determine which node in the cluster should handle a particular request.
   - Data Structures Used:
     a. `AtomicInteger` from `java.util.concurrent.atomic`: Used for maintaining the atomic counter for affinity.
2. **ClusterManager**:
   - `getInstance()`: This is a singleton pattern implementation to get an instance of the `ClusterManager`.
   - `clusterSize()`: Returns the size of the cluster (number of nodes).
   - `nextNode()`: Returns the index of the next node in the cluster.
   - `get(int port)`: Returns the node (host:port) at the specified index.
   - `nodeIndex()`: Returns the index of the current node in the cluster based on the value of the `Node_Port` environment variable.
   - Data Structures Used:

   a. `List<String> ports`: A list of node addresses (host:port).
   b. `volatile ClusterManager instance`: Singleton instance of the `ClusterManager`.

3. **DatabaseManager**:
   - `getInstance()`: Singleton pattern implementation to get an instance of the `DatabaseManager`.
   - `getAffinityNode(String databaseName, String collectionName)`: Returns the affinity node for a specific database and collection. Affinity determines which node in the cluster is responsible for the data.
   - Data Structures Used:
     a. `HashMap<String, Database> databases`: Stores databases by name.
     b. `ReentrantReadWriteLock databaseLock`: Provides read and write locks for managing concurrent access to the databases.
4. **FileManager**:
   - `fileExists(String databasePath)`: Checks if a file or directory exists at the specified path.
   - `createDirectoryIfNotFound(String directory)`: Creates a directory if it doesn't already exist.
   - `directoryOrFileExists(String pathForFileOrDirectory)`: Checks if a directory or file exists at the specified path.
   - `createJsonFile(String directory, String jsonName)`: Creates a JSON file with the given name in the specified directory.
   - `getDocument(String databaseName, String collectionName, String documentId)`: Reads a JSON document from the file system and returns it as a `JsonNode`.
   - `readNumberFromFile(String filePath)`: Reads an integer value from a file.
5. **LoadBalancingManager**:
   - Constructor: Initializes a linked list to log requests and sets up a timer to periodically clean up old requests.
   - `logRequest()`: Logs a request by adding a timestamp to the linked list.
   - `getRequestsInLastMinute()`: Returns the number of requests logged in the last minute.

- `cleanup()`: Removes old requests from the linked list.
- `isRedirectRequired()`: Checks if a request redirection is required based on the number of requests in the last minute.
- Data Structures Used:
  a. `LinkedList<Long> requests`: Used to store timestamps of requests.
  b. `Timer timer`: Used for periodic cleanup of old requests.

6. **RequestManager**:
   - `buildUri(String node, String url, String databaseName, String collectionName, String documentName, String key, Boolean propagate, Boolean fromAffinityNode)`: Builds a URI for making an HTTP request to a specific node in the cluster.
   - `buildHttpEntity(String schema, String hName)`: Builds an HTTP entity with custom headers.

3.8 Utils

1. `Database` is a class representing a database.
   - Data Structures:
     a. `Map<String, Collection> collections`: Maps collection names to `Collection` objects.
     b. `ReentrantReadWriteLock collectionLock`: Provides read and write locks for managing concurrent access to collections.
2. `Collection` is a class representing a collection within a database.
   - Data Structures:.
     a. `List<String> documents`: Stores the names of documents in the collection.
     b. `List<String> keyIndex`: Stores key index information.
     c. `ReentrantReadWriteLock documentLock`: Provides read and write locks for managing concurrent access to documents.
     d. `Map<String, Map<String, IndexObject>> index`: Maps index keys to index objects.

3. `IndexObject` represents an index object containing a list of document indexes.
    - Data Structures:
        a. `List<String> indexes`: Stores document indexes.
4. `SchemaTypes` is an enum representing possible data types for JSON schema attributes.
    - Data Structures:
        a. Enum constants for different data types.
5. `Indexing` is a class responsible for indexing documents in a collection.
    - `newIndex(String databaseName, String collectionName, String key)`:
        a. This method is used to create a new index for a specified key in a collection of a database.
        b. Data Structures:
            i. `Database`: Represents a database.
            ii. `Collection`: Represents a collection within a database.
            iii. `Map<String, Map<String, IndexObject>> index`: The collection's index structure, which is a nested map containing keys, values, and `IndexObject` objects.
            iv. `Map<String, String> att`: A map containing attribute names and their values extracted from documents.
            v. `IndexObject`: Represents an index object containing a list of document indexes.
        c. Explanation
            i. It first retrieves the database and collection objects using `DatabaseManager` and the provided names.
            ii. It reads the JSON schema file associated with the collection to verify if the provided `key` is a valid attribute.
            iii. If the key is valid, it initializes a new index for the key in the collection.
            iv. It then iterates through the documents in the collection, extracting values for the `key` attribute and adding them to the index.

- `indexDocument(String databaseName, String collectionName, String documentName)`
    a. This method is used to index a specific document in a collection.
    b. Data Structures:
        i. Same as in the `newIndex` method.
    c. Explanation
        i. It retrieves the database and collection objects using `DatabaseManager` and the provided names.
        ii. It extracts attribute values from the document specified by `documentName`.
        iii. It then iterates through the collection's existing index keys and updates the index for each key with the document's index.
- `removeIndexDocument(String databaseName, String collectionName, String documentName)`
    a. This method is used to remove the index entries for a specific document in a collection.
    b. Data Structures:
        i. Same as in the `newIndex` method.
    c. Explanation
        i. It retrieves attribute values from the document specified by `documentName`.
        ii. It iterates through the collection's existing index keys and removes the document's index from each key.
        iii. If an index becomes empty after the removal, it removes the index key from the collection's index.

6. `DocumentSchema` is a utility class for working with JSON document schemas.
    - `public static boolean verifyJsonTypes(JsonNode docSchema)`
        a. This method is used to verify if the types of attributes in a JSON document (represented by `docSchema`) conform to a predefined set of types (`SchemaTypes`).

   b. Data Structures:
      i. `SchemaTypes`: An enum containing predefined attribute types.
      ii. `Map<String, String>`: Stores attribute names and their types.
   c. Explanation
      i. It first extracts the attribute names and their types from the `docSchema` JSON object and stores them in an attribute map.
      ii. It then iterates through the attribute types and checks if each type exists in the `SchemaTypes` enum.
      iii. If any attribute type doesn't match the predefined types, it returns `false`.
      iv. Additionally, it checks if the `id` attribute exists in the `docSchema`. If not, it adds an `id` attribute with the value "schema" to the `docSchema`.

- `public static Map<String, String> getAttributeMap(JsonNode jsonDoc)`
   a. This method extracts attribute names and their types from a JSON document (`jsonDoc`) and stores them in a map.
   b. Data Structures:
      i. `Map<String, String>`: Stores attribute names and their types.
   c. Explanation
      i. It recursively traverses the JSON document to handle nested objects.
      ii. It checks if each property in the JSON document is an object; if so, it calls itself recursively to extract attributes from nested objects.
      iii. If a property is not an object, it extracts its value as a string and adds it to the attribute map.
- `public static boolean verifyJsonFileWithSchema(JsonNode jsonDoc, JsonNode jsonSchema)`
   a. This method is used to verify if a JSON document (`jsonDoc`) conforms to a given JSON schema (`jsonSchema`).
   b. Data Structures:
      i. `Map<String, String>`: Stores attribute names and their types extracted from the document and schema.
   c. Explanation

i. It extracts the attribute names and their types from both the JSON document and the schema using the `getAttributeMap` method.

ii. It then compares the attribute names from the JSON document with those from the schema to ensure they match.

iii. If any attribute name from the JSON document is missing in the schema, it returns `false`, indicating that the document does not conform to the schema.

4. Multithreading and Locks

I will be talking about the locking mechanism i used and how i implemented it in my code

4.1 ReadWriteLocks Overview

ReadWriteLocks, also known as "multiple readers, single writer" locks, are a type of synchronization mechanism used in concurrent programming to control access to shared resources. They provide a way to manage access to data in a multi-threaded or multi-process environment where multiple threads may need to read the data simultaneously, but write operations must be performed exclusively.

Here's an overview of ReadWriteLocks, what they do, and their purpose:

1. Read Lock (Shared Lock):

- The ReadWriteLock has two types of locks: read locks and write locks. A read lock is also called a shared lock.
- Multiple threads can acquire read locks simultaneously without blocking each other.
- Read locks are used when threads only need to read the shared data and not modify it. This allows for efficient parallelism since multiple readers can access the data concurrently without interfering with each other.
- Read locks do not block other read locks, meaning that while one thread holds a read lock, other threads can also acquire read locks.

2. Write Lock (Exclusive Lock):

- The write lock is also called an exclusive lock. It is used when a thread wants to modify the shared data.
- Only one thread at a time can acquire a write lock, and it must wait until no other threads hold any read or write locks.
- Write locks ensure exclusive access to the shared resource, preventing multiple threads from modifying it simultaneously, which could lead to data corruption.

3. Purpose of ReadWriteLocks:

- **Improve Concurrency:** ReadWriteLocks are designed to improve concurrency in situations where multiple threads need to access shared data. By allowing multiple threads to read concurrently, they can increase the overall throughput of your application.
- **Optimize Read-Mostly Scenarios:** ReadWriteLocks are particularly useful in scenarios where data is mostly read, and write operations are less frequent. In such cases, read operations can occur simultaneously, reducing contention and improving performance.
- **Prevent Write Conflicts:** Write locks ensure that write operations are atomic and exclusive. This prevents data corruption and maintains data integrity when multiple threads want to modify the same resource.
- **Reduce Blocking:** By allowing multiple readers to access data simultaneously, ReadWriteLocks reduce contention and blocking, which can occur when using traditional locks. This can lead to better responsiveness and resource utilization.

4. When to Use ReadWriteLocks:

- ReadWriteLocks are most beneficial in scenarios where the data access patterns are predominantly read-heavy. If your application requires frequent writes with few reads, traditional locks might be more appropriate.
- Use ReadWriteLocks when you want to optimize concurrent access to shared resources without compromising data consistency.
- Keep in mind that while ReadWriteLocks can improve performance in read-mostly scenarios, they introduce complexity and potential issues like deadlock if not used correctly. Therefore, it's essential to carefully design your application's synchronization strategy.

4.2 ReadWriteLocks Implementation

The locking mechanisms are implemented in the `DatabaseService`, `CollectionService`, and `DocumentService`, they are crucial for ensuring data consistency and preventing race conditions in a multi-threaded or distributed environment. Let's discuss these locking mechanisms in more detail for each service:

1. DatabaseService Locking Mechanism:

- The `DatabaseService` uses ReadWriteLock from the Java Concurrency package (`java.util.concurrent.locks`) to manage concurrent access to database-related operations.
- Two types of locks are used:
  - `writeLock`: This lock is acquired for operations that modify the state of the database, such as creating or deleting a database. It ensures that only one thread can perform these operations at a time, preventing conflicts and maintaining data consistency.
  - `readLock`: This lock is acquired for read-only operations, such as retrieving the list of databases. Multiple threads can hold the read lock simultaneously, allowing concurrent read operations without blocking.
- The `writeLock` is acquired using `writeLock().lock()` and released using `writeLock().unlock()`. Similarly, the `readLock` is acquired using `readLock().lock()` and released using `readLock().unlock()`.
- The use of these locks ensures that database creation and deletion operations are atomic and that reading the list of databases can be performed concurrently without interference.

2. CollectionService Locking Mechanism:

- The `CollectionService` employs a similar locking mechanism using ReadWriteLock for managing concurrent access to collection-related operations.
- Just like in the `DatabaseService`, there are `writeLock` and `readLock` locks.
- `writeLock` is used when creating or deleting a collection, creating an index, and updating documents. These operations modify the state of a collection and require exclusive access.
- `readLock` is used when retrieving the list of collections and filtering documents by key or value. These operations are read-only and can be performed concurrently.

- The locks are acquired and released using the same methods as described in the `DatabaseService` section.
- These locks ensure that operations like collection creation, deletion, and indexing are performed atomically and prevent conflicts when querying data.

3. DocumentService Locking Mechanism:

- The `DocumentService` also implements a locking mechanism, but it uses a different approach. It utilizes ReentrantReadWriteLock from the Java Concurrency package.
- Just like in the other services, there are `writeLock` and `readLock` locks.
- `writeLock` is used when creating, deleting, or updating a document within a collection. These operations modify the state of the collection and require exclusive access.
- `readLock` is used when retrieving all documents within a collection. This operation is read-only and can be performed concurrently.
- The locks are acquired and released using the same methods as described in the previous sections.
- These locks ensure that document-related operations like creation, deletion, and updating are performed atomically within a collection and prevent conflicts when retrieving data.

In summary, the locking mechanisms in these services are crucial for maintaining data consistency and preventing race conditions in a multi-threaded or distributed environment. They provide a way to control access to critical sections of code, ensuring that only one thread can perform certain operations at a time while allowing concurrent access for read-only operations. This approach helps maintain the integrity of my decentralized, cluster-based NoSQL database system and ensures that data modifications are synchronized and atomic.

5. Data Consistency issue in the Database

In my code there are potential data consistency issues that can occur. Here are some considerations:

1. **Race Conditions**: Even with ReadWriteLocks, I need to be cautious about race conditions. Race conditions occur when multiple threads access and modify shared data simultaneously, leading to unpredictable outcomes. In my code, it's essential to ensure that critical sections of code (e.g., data modification) are properly synchronized with write locks.
2. **Transaction Management**: If my application needs to perform multiple operations that should be treated as a single transaction (e.g., updating data across multiple collections), I'll need to carefully manage transactions to maintain data consistency. If one part of the transaction fails, I may need to roll back changes made earlier in the transaction.
3. **Read-Modify-Write Sequences**: If I have scenarios where I read data, perform some computation, and then write the modified data back, I should be aware that between the read and write operations, other threads might have modified the data. This can result in lost updates or incorrect computations.
4. **Deadlocks**: While ReadWriteLocks help prevent some types of data inconsistency, they can also introduce the potential for deadlocks. Deadlocks occur when multiple threads are waiting for each other to release locks, resulting in a standstill. Proper lock acquisition order and deadlock detection/resolution strategies are essential.

6. Load balancing and Affinity

6.1 Load Balancing

Load balancing is a crucial component in distributed systems where multiple servers or nodes work together to handle incoming requests. The goal of load balancing is to distribute the workload evenly among these nodes to prevent overload on any single node, ensuring optimal resource utilization and improved system performance.

In my code, I have implemented a form of load balancing to distribute databases and collections across my decentralized cluster-based NoSQL database system. I have done this by making a round robin approach when one node gets too many requests it sends that request to the next node and so on and also by organizing databases and collections into directories and files within a storage path. Here are the key aspects of load balancing in my code:

1. **Database and Collection Creation**: When I create a new database or collection, my code checks if it already exists. If the database or collection does not exist, it proceeds with the creation process.
2. **Load Balancing Decision**: Instead of explicitly distributing databases or collections to specific nodes or servers, my code follows a round-robin fashion for load balancing. This means that when a new database or collection is created, it doesn't choose a specific server or node to place it on.
3. **Implicit Distribution**: By not specifying a server or node for placement, my code implicitly relies on the file system's behavior to determine where to create the database or collection directory. In a round-robin fashion, the file system decides the next available location to create the directory.
4. **Request Routing**: When clients or requests come in to access a particular database or collection, my code doesn't involve a central load balancer to decide where to route the request. Instead, the client interacts directly with the database or collection based on its name. The file system's behavior (e.g., directory creation) essentially handles the load balancing.
5. **Balancing Overload**: If one node or server is under heavy load due to an influx of requests, the round-robin approach ensures that subsequent database or collection creations are distributed to other available nodes in a cyclical manner. This helps distribute the workload across the nodes in my cluster.

6. **Simplicity**: Round-robin load balancing is relatively simple to implement, as it doesn't require complex algorithms or central load balancers. It leverages the existing file system's behavior for distribution.

While my load balancing approach is relatively simple and directory-based, it ensures that databases and collections are organized and distributed within my storage path, preventing potential bottlenecks and overloading of specific directories.

6.2 Affinity System

An affinity system, in the context of distributed systems, typically refers to a mechanism that ensures that related data or tasks stay together on the same node or cluster to improve data locality and reduce latency. This can be especially important for systems with distributed databases or caches.

In my code, I have also implemented an affinity system to associate collections with affinity values. The affinity value for each collection is set when I create a new collection using the `createCollection` method. Here's how my affinity system works:

1. **Affinity Assignment**: When a new collection is created, it is assigned an affinity value using `AffinityManager.getNextAffinity()`. This affinity value is associated with the collection and stored in my database management system.
2. **Affinity-Based Access**: Later, when I perform operations on collections such as indexing or filtering, the affinity value is used to determine which node or server should handle the operation. Collections with the same affinity value are likely to be stored on the same server, ensuring data locality and reducing communication overhead.
3. **Balancing with Affinity**: By assigning affinity values to collections, I am ensuring that related collections are co-located, which can improve performance when querying related data. This is a form of affinity-based load balancing within my system.

Overall, my affinity system allows me to group related collections together on the same server, improving data access times and reducing the need for cross-network communication when querying related data.

My load balancing mechanism organizes databases and collections into directories within my storage path, while my affinity system associates collections with affinity values to optimize data locality and reduce communication overhead in my decentralized NoSQL database system. These mechanisms help distribute and manage data efficiently in my system.

7. Communication between nodes

Communication Mechanism:

I am using RestTemplate to facilitate communication between nodes. Here's how it works in more detail:

1. **RESTful API Endpoints:** Each node in my cluster exposes a set of RESTful API endpoints. These endpoints define the operations that can be performed on the node, such as creating databases, collections, or documents, as well as querying data.
2. **HTTP Requests:** When one node needs to communicate with another (e.g., to replicate data or perform distributed queries), it creates HTTP requests using RestTemplate. These requests are sent to the RESTful API endpoints on the target node.
3. **Request Handling:** The target node's server receives the incoming HTTP request. It routes the request to the appropriate handler method based on the endpoint and HTTP method (e.g., GET, POST, PUT, DELETE).
4. **Business Logic:** Inside the handler method, the target node performs the necessary business logic to fulfill the request. This may include reading or writing data to its local storage, executing queries, or coordinating with other nodes in the cluster.
5. **Response:** After processing the request, the target node sends back an HTTP response to the originating node. The response contains the result of the operation, which could be data, status information, or error messages.
6. **Error Handling:** Proper error handling is crucial in distributed systems. If the target node encounters an issue while processing the request, it responds with an appropriate HTTP error status code (e.g., 4xx or 5xx) and, optionally, an error message in the response body.

Affinity:

In my system, I have implemented an affinity system to associate data with specific nodes. This is important for maintaining data consistency and efficient data access. When communicating between nodes, I should consider this affinity information to ensure that requests related to specific data or collections are routed to the appropriate nodes.

In summary, my communication between nodes relies on RestTemplate and RESTful APIs, which is a reasonable choice for many distributed systems. However, I will need to consider scaling, data consistency, fault tolerance, security, and monitoring as my system evolves. Additionally, I may explore more advanced communication technologies as my system's complexity increases.

8. Security issues

I can provide some general insights into potential security issues that can occur in distributed systems, especially when using RESTful APIs and communication between nodes. Here are some common security concerns to be aware of:

1. **Authentication and Authorization:** Lack of proper authentication and authorization mechanisms can lead to unauthorized access to my APIs or data.
2. **Data Encryption:** Data transferred between nodes may be exposed to eavesdropping if not encrypted. I can use HTTPS (TLS/SSL) to encrypt data in transit. Ensure that sensitive data, such as authentication tokens or database credentials, is securely transmitted and stored.
3. **Input Validation:** Insufficient input validation can lead to security vulnerabilities like SQL injection, cross-site scripting (XSS), or other injection attacks. I can implement strict input validation and sanitize user inputs to prevent these common attacks. I can use parameterized queries or prepared statements to prevent SQL injection.
4. **API Rate Limiting and Throttling:** Without rate limiting and throttling, my APIs may be vulnerable to abuse or denial-of-service attacks. I can implement rate limiting to restrict the number of requests from a single IP or user within a specific time frame. Throttle excessive requests to prevent system overload.

5. **Error Handling:** Exposing sensitive information in error responses (e.g., stack traces or database error details) can aid attackers. I can implement proper error handling and ensure that error responses do not disclose sensitive information. Log errors securely for debugging purposes.
6. **Cross-Origin Resource Sharing (CORS):** If CORS is not properly configured, my APIs may be vulnerable to cross-site request forgery (CSRF) and other cross-origin attacks. I can implement CORS policies to control which domains are allowed to make requests to my APIs.
7. **Session Management:** Poor session management can lead to session fixation or session hijacking. I can implement secure session management practices, such as using random session identifiers, enforcing session expiration, and using secure cookies.
8. **Third-Party Libraries and Dependencies:** Outdated or vulnerable third-party libraries can introduce security risks into my system. I should regularly update and patch third-party dependencies. Or I can use tools to scan for known vulnerabilities in my dependencies.

To identify and address these security concerns in my specific code, I can consider performing security assessments, such as code reviews and penetration testing. Additionally, staying updated on security best practices and vulnerabilities relevant to the technologies and frameworks I am using.

9. Code cleanliness, Design Patterns and DevOps

9.1 Clean Code Principles

1. **Descriptive Variable and Method Names**: My code exemplifies the clean code principle of using descriptive and meaningful variable and method names. This practice enhances the understandability of the code and makes it self-documenting.
2. **Consistent Formatting and Indentation**: Consistency in code formatting and indentation is maintained throughout my codebase. This consistency not only enhances code readability but also simplifies maintenance.
3. **Modularity and Organization**: My code adheres to the clean code principle of modularity and organization. It is logically divided into well-defined classes and methods, promoting a separation of concerns and ease of maintenance.
4. **Encapsulation and Information Hiding**: Proper encapsulation techniques have been applied, and access modifiers are used judiciously. This adherence to information hiding principles protects the integrity of my code.
5. **Error Handling**: My code incorporates error handling mechanisms, which are crucial for improving fault tolerance and providing informative feedback in case of issues.
6. **Effective Use of Libraries**: Leveraging external libraries like Jackson and Spring showcases my code's efficiency and ability to harness existing resources, reducing the need for boilerplate code.
7. **Dependency Injection**: The principle of dependency injection is utilized effectively, resulting in loosely coupled components. This practice not only enhances maintainability but also facilitates unit testing.
8. **Consistent Naming Conventions**: My code follows consistent naming conventions for variables, methods, and classes. This consistency fosters clarity and aids in comprehending the codebase.

9.2 Design Patterns

1. **Singleton Pattern**: My codebase makes use of the Singleton design pattern, which guarantees the existence of a single instance of manager classes like DatabaseManager and ClusterManager. This pattern promotes resource efficiency and control.

2. **Factory Method Pattern**: Although not explicitly labeled as such, I have methods like `newIndex` and `indexDocument` in the `Indexing` class that create and manage instances of `IndexObject`. This can be seen as a simplified form of the Factory Method pattern where I create objects without specifying the exact class of the object that will be created.
3. **Composite Pattern**: In the `DocumentSchema` class, I recursively traverse a JSON structure to build a map of attributes. This recursive processing of a complex structure is reminiscent of the Composite pattern, where objects can be composed into tree structures to represent part-whole hierarchies.
4. **Strategy Pattern**: My `LoadBalancingManager` class uses a strategy-like approach to log and clean up requests. It employs a scheduling strategy using a TimerTask. While not a strict Strategy pattern implementation, it shares some similarities with this pattern.
5. **Decorator Pattern**: In some parts of my code, I extend the behavior of classes, such as when adding indexes to collections in the `Indexing` class. This extension of functionality is similar to the Decorator pattern, where I can add new behaviors to objects dynamically.

9.3 SOLID Principles

1. **Single Responsibility Principle (SRP)**: The SRP is evident in my codebase, with each class having a distinct responsibility. For example, classes are responsible for managing databases or clusters, aligning perfectly with the SRP.
2. **Open/Closed Principle (OCP)**: My codebase exhibits adherence to the Open/Closed Principle. It is designed to be extensible without necessitating modifications to existing code. This facilitates the addition of new database operations or features seamlessly.
3. **Liskov Substitution Principle (LSP)**: Although not explicitly demonstrated, adherence to the SRP and OCP often indirectly supports the Liskov Substitution Principle (LSP), ensuring that derived classes can be substituted for their base classes without issues.
4. **Interface Segregation Principle (ISP)**: While my codebase doesn't explicitly use interfaces, the Spring framework's component-based architecture adheres to the concept of dependency injection, promoting loose coupling and adhering to ISP principles.

5. **Dependency Inversion Principle (DIP)**: My codebase embraces the Dependency Inversion Principle by effectively employing dependency injection. This practice decouples high-level modules from low-level modules, enhancing flexibility and testability.

9.4 DevOps Practices

1. **Continuous Integration (CI)**: My codebase is well-suited for Continuous Integration practices, where code changes can be automatically integrated and tested. CI pipelines can be established to streamline the development process.
2. **Continuous Deployment (CD)**: With CI in place, my codebase can also support Continuous Deployment practices. This entails automated deployments to various environments, reducing manual intervention and speeding up the release cycle.
3. **Infrastructure as Code (IaC)**: Although specific IaC tools aren't explicitly mentioned, my codebase appears to interact with various services and infrastructure components. Managing infrastructure as code is a pivotal DevOps practice.
4. **Configuration Management**: My code effectively manages configuration settings through environment variables. This is a best practice for configuration management in DevOps, ensuring consistency across environments.
5. **Monitoring and Logging**: Proper error handling and logging mechanisms are incorporated into my codebase. These are critical for DevOps practices, as they facilitate the detection and diagnosis of issues in production environments.
6. **Scalability and Resilience**: My codebase demonstrates an understanding of scalability through its cluster-based architecture. Additionally, resilience is addressed through robust error handling and retry mechanisms.
7. **Containerization**: My codebase references container-related elements, suggesting the potential use of containerization technologies like Docker. Containerization is integral to achieving portability and consistency in DevOps.
8. **Orchestration**: The mention of orchestrating containers and the use of RESTful APIs for communication aligns with container orchestration practices. This enables efficient management of containerized applications.
9. **Automation**: Automation is a cornerstone of DevOps, and my codebase includes automated processes for tasks like initialization and indexing. This automation reduces manual intervention, leading to increased efficiency.

10. **Feedback Loops**: DevOps relies on feedback loops for continuous improvement. My codebase facilitates these loops through CI/CD pipelines and monitoring mechanisms, allowing rapid identification and resolution of issues.

These principles, patterns, and practices collectively contribute to clean, maintainable, and DevOps-friendly code. However, the effective implementation of these practices also depends on their integration into my overall development and deployment processes. DevOps, in particular, encompasses culture, practices, and tools, working in harmony to deliver software more effectively.