ATYPON

ATYPONDB

Decentralized Cluster-Based NoSQL DB System

September– **2023**

Khaled **Al-H**afnawi

Seat **N**o. **14**

Java and DevOps Training

– Summer 23

ABSTRACT

In this report, I will provide documentation for a Decentralized Cluster-Based NoSQL DB System Project. In this documentation, I will illustrate the various components of the project: the Client, Bootstrap Node, and Worker Nodes. I will detail their functions and explain how I implemented them. Additionally, I will describe system intricacies such as my solutions to race conditions, the inclusion of an indexer, and the memory efficiency and speed of the system. I will also explain the load balancer that the Bootstrap Node uses to evenly distribute users among nodes.

Table of Contents

• Introduction

In this project, we need to build a Decentralized Cluster-Based NoSQL DB System. In this project we dive into NoSQL world specifically focusing on a decentralized cluster-based NoSQL DB system. Unlike centralized systems where there is a manager node control all requests, the decentralized paradigm eliminates this point. Consequently, maintaining data consistency and load balancing across the cluster necessitates techniques.

Our goal in this project is to design and implement an application that simulates the interaction between the users and nodes inside the decentralized NoSQL database cluster. Benefiting of Java power and flexibility.

The project encapsulates challenges such as, bootstrapping the cluster, maintaining data consistency, handling race conditions, and optimizing indexing in the database. I will illustrate each challenge and discuss its solution.

As you proceed through this documentation, you will find detailed explanations of the system's architecture, the reasoning behind design choices, and the strategies employed to overcome challenges specific to decentralized systems.

• Overview

This project delves into a unique aspect of NoSQL databases: a decentralized cluster-based system. Instead of relying on a singular managerial unit, which could lead to load balancing issues, bottlenecks, or become a single point of failure, our system employs a decentralization mechanism. Let's take a high-level sneak peek into our project's structure. Broadly, the project can be divided into three stages:

First Stage - Worker Node:

This is the stage that contains the lion's share of the challenges and details. It involves complexities associated with creating, reading, updating, and deleting documents, collections, and database details from one node and broadcasting these to all other nodes in the network. This stage also delves into the intricacies of lock mechanisms aimed at avoiding race conditions. Specific attention is given to the assignment of document affinity, ensuring modifications are permitted only from the affinity node. Additionally, the complexities of the indexer, data storage, retrieval, deletion, and the caching mechanisms to expedite data reading and retrieval are discussed in-depth.

Second Stage - Bootstrapping:

Also known as the bootstrap node stage, this phase is responsible for initiating the worker nodes and managing user requests. It checks which node a user is assigned to and manages the sign-up of new users. A load balancing mechanism is integrated to ensure even distribution of user assignments across nodes.

Third Stage - User Interface:

In this stage, all user requests, whether from a regular user or an admin, are addressed. Users can opt to either log in or sign up. For returning users, after validation, they can select their desired action - whether it's adding, deleting, creating, updating, or selecting - after connecting to their assigned node. New users go through a username validation process by the bootstrap, which then assigns them to a node and sends their details accordingly.

With this structure in mind, it's time to explore the project details. How can we design a decentralized cluster-based NoSQL database system that retains flexibility and scalability while ensuring data integrity, user-node affinity, and efficient indexing? Furthermore, how can our system effectively manage challenges like race conditions, user authentication, and node communication without central oversight?

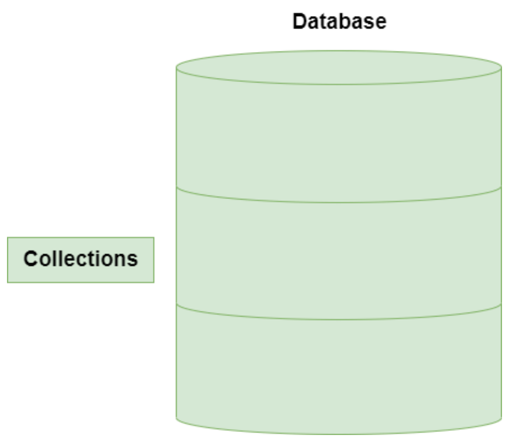
• Database Implementation.

In this project, we aim to implement a NoSQL database. First, let's understand what a database is. Essentially, a database is a mechanism used to store, manage, and retrieve information. It serves as a repository or collection of data. There are primarily two types of databases: the relational (SQL) and the non-relational (NoSQL). This project will focus on non-relational databases, commonly referred to as NoSQL databases.

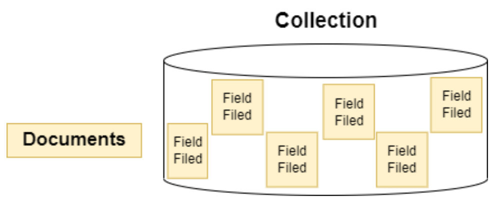
So, **what exactly is a NoSQL database**? A NoSQL database is a type of database management system (DBMS) that provides mechanisms to store and retrieve data in ways distinct from traditional relational databases. These databases are frequently utilized in applications that demand vast data volumes, low latency, and flexible data models. Notably, a non-relational database is designed to store and query data as JSON documents, positioning a document database as an enhanced key/value pair system.

Terminologies in our Database:

- **Database:** The overarching structure housing multiple collections. It establishes a namespace for collections and ensures a layer of separation from other databases.
- **Collection:** A set of documents, comparable to the **'table'** in traditional relational databases, but without a rigid schema. Collections facilitate effective data categorization and access.



- **Document:** A collection of fields and serves as the basic data unit in our NoSQL system. Typically, it's represented in JSON (JavaScript Object Notation) format, consisting of key-value pairs.
- **Field:** Individual data elements within a document, each associated with a specific key and value. Fields form the foundational units of documents and are pivotal for data representation and retrieval.



▪ Implementation.

Let's delve into how our project structure reflects our database implementation. Initially, we have a storage package that encompasses all database implementation details, ranging from low level to high level. This package is further divided into four sub-packages.

The first, models package, contains classes for documents, collections, and databases, detailing elements like fields in documents, names, IDs, affinity numbers, and other attributes we'll delve into shortly.

The second is the handlers package, which houses all the handlers and access mechanisms for models, such as reading a document, creating it, modifying a collection, and so on.

Other noteworthy packages include the indexing pack and the directory operation pack, which we will explore in subsequent sections.

Now, to dive deeper into the models pack:

Model package:

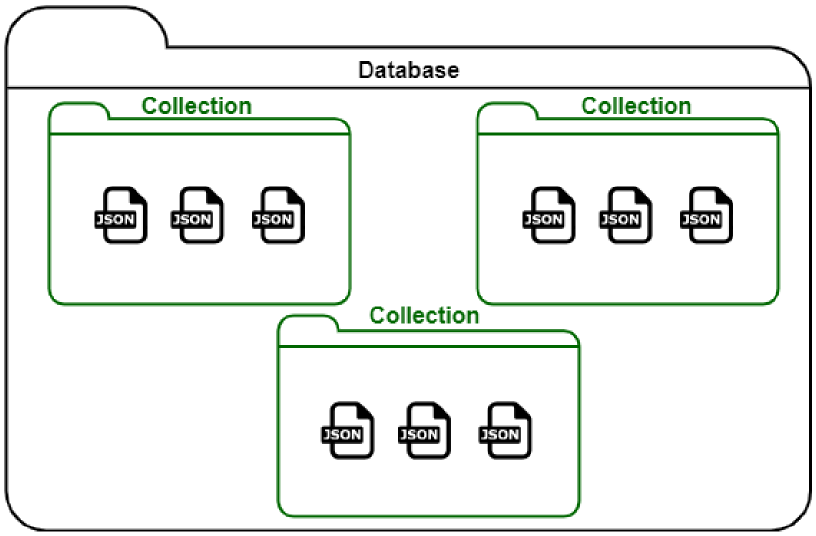
1- Document class

Starting with the fundamental data unit, the `Document` class represents our smallest level, which represent as JSON file. Breaking down the document further, at its core, it must have the document name, document ID, and an affinity number which is the latter signifies the owner node ID that exclusively has the rights to modify or delete the document.

For field access within this class, we've implemented several methods. The `removeField(String key)` method allows the removal of a field, updating the indexer value simultaneously. The `addField(String attribute, String value)` method facilitates the addition of a new field, performing tasks similar to the remove method, such as updating the indexer. Meanwhile, the `replaceField(String attribute, String value)` method updates an attribute's value and ensures the indexer version remains current.

Besides field operations, two crucial methods are integral at every project stage. The `readDataFromFile(File file)` method reads an existing file into the database, creating a new document object with the ID, name, and affinity number found within the file. The `addDocument()` method, on the other hand, allows reading and writing attributes and values directly from users, subsequently adding them to both the database and the indexer.

Lastly, we've overridden the `toString` method to return the document's content whenever a user requests to view (or select query) the document.



2- Collection class

As illustrated in the previous figure, a collection is essentially a folder containing documents or JSON files associated with that collection. In our project, the `Collection` class comprises the collection name and a Map that accepts String as keys and `Document` as values. This allows us to store documents under their respective names within the same collection.

Within the Collection class, we find several operations pertaining to documents:

- `addDocument (String documentName)`: This method adds a new Document object to the HashMap with the name provided in the parameter.
- `getDocument(String documentName)`: Operating similarly to the addDocument method, this function checks for the existence of a document. If found, it returns the document; otherwise, it creates a new document with the specified name and returns it.
- `deleteDocument(String documentName):` This method removes a document from the collection.

Furthermore, we've overridden the `toString` method to enhance its functionality. Instead of the default behavior, it now iterates over the collection, retrieves all its documents, and prints their details. This modification proves particularly beneficial when a user wishes to view the entire collection, as might be the case with a "Select collection" command.

Here example for to printing the to String method for HR collection.

```
Collection {
    name: HR
    documents: {
        {"Firstname":"Huda","Lastname":"Aloosh","id":"1170b38c-0a8d-4faf-b2ed-
867aadd6e787","affinity":1}
        {"Firstname":"waleed","Lastname":"Qarni","id":"7690f70a-e4f5-415c-8bd4-
b9459c0a5065","affinity":2}
    }
}
```

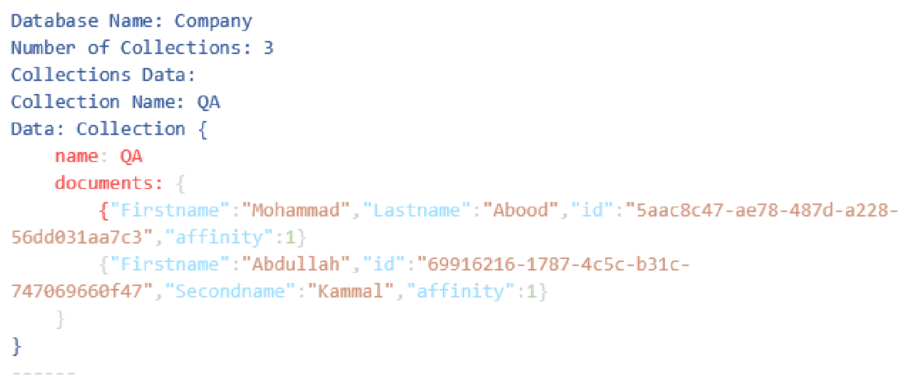
3- Database class

In the project, the database can be conceptualized as a large folder containing various collection folders, which in turn house the document files. The `Database` class, like the structure of the `Collection` class, includes a database name and a Map that accepts a String as a key and a Collection as a value. This design facilitates the storage of collections using their names within the database.

Within the `Database` class, we find several operations centered on collections, echoing the operations for documents in the `Collection` class:

- `addCollection (String collectionName)`: This method adds a new Collection object to the HashMap with the name provided in the parameter.
- `getCollection (String collectionName)`: Operating similarly to the addCollection method, this function checks for the existence of a collection. If found, it returns the collection; otherwise, it creates a new collection with the specified name and returns it.
- `deleteCollection(String collectionName):` This method extracts a specified collection from the database.

For the `toString` method, we've overridden its default behavior. Now, it iterates over every collection within the database and, subsequently, over every document within those collections, printing the data in a user-friendly format.



```
Database Name: Company
Number of Collections: 3
Collections Data:
Collection Name: QA
Data: Collection {
    name: QA
    documents: {
        {"Firstname":"Mohammad","Lastname":"Abood","id":"5aac8c47-ae78-487d-a228-
56dd031aa7c3","affinity":1}
        {"Firstname":"Abdullah","id":"69916216-1787-4c5c-b31c-
747069660f47","Secondname":"Kammal","affinity":1}
    }
}
------
```

Here example for to printing the toString method for Company DB.

Handlers package:

1- DocumentHandler class

The `DocumentHandler` class implements `IDocumentHandler` provides direct access to the Document class, enabling creation, reading, updating, and deletion of documents.

Within this class, we find four notable methods:

- `JsonNode createDocument(…)`: This method allows adding a document based on user input. It first validates the collection name and checks for the database's existence. Upon successful creation, it returns a JsonNode to facilitate broadcasting to other nodes.
- `void createDocument(…)`: Operating similarly to the previous method, this version accepts a JsonNode as a parameter and creates a file utilizing the information within the JsonNode.
- `readDocument(…)`: This method is essential for database loading and for accessing an existing document, whether to read or modify it.
- `deleteDocument(…)`: This method removes a document from the database and also eliminates its extensions.

2- CollectionHandler class

The `CollectionHandler` class, which implements `ICollectionHandler`, serves as a higher layer to the `Collection` class. This class encapsulates the logic for creating and deleting collections from the database, determining details such as whether a collection contains documents or is empty.

Within this class, two primary methods apply the logic:

- `createCollection(String databaseName, String collectionName)`: This method creates a new collection within a specified database. It adds the collection to the in-memory target database and sets up a corresponding directory on the filesystem using the `DirectoryHandler`.
- `deleteCollection(String databaseName, String collectionName)`: This method deletes a specified collection from a given database. It first removes all documents within that collection from the `HashIndexer`, then deletes the collection from the in-memory target database, and lastly removes the associated directory from the filesystem.

3- DatabaseHandler class

The `DatabaseHandler` class, implementing `IDatabaseHandler`, oversees operations related to databases. It offers functionalities for creating, retrieving, and deleting databases.

Key interactions with the database include retrieving a database for operations (e.g., `database.toString()`) and managing the database list. Specifically:

- `getDatabase(String name)`: This method retrieves a named database from the databaseList.
- `createDatabase(String database)`: It first checks if a database with the provided name already exists. If not, a new database is created, added to the databaseList, and a corresponding directory is set up on the filesystem using the `DirectoryHandler`.
- `deleteDatabase(String databaseName)`: This method deletes a named database. It first removes associated documents from the `HashIndexer`, then expunges the database from the databaseList, and finally removes the associated directory from the filesystem.

4- CRUDHandler class

The `CRUDHandler` class manages the basic CRUD (Create, Read, Update, Delete) operations on specific documents within a collection inside the database. These operations enable users to save, retrieve, and update information. Delving deeper, the class focuses on nuances like validation during creation, updating, and deletion, as well as details related to document locking.

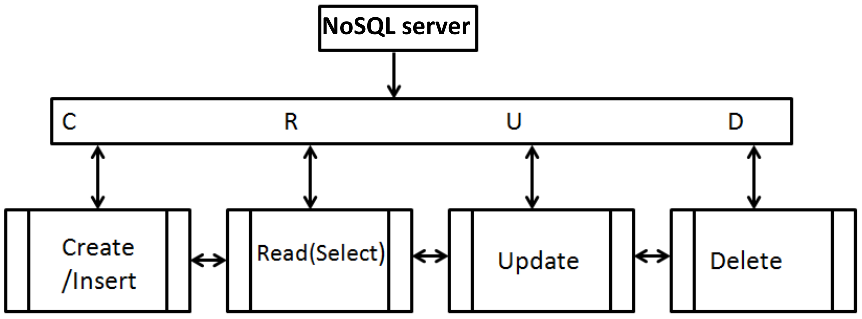
Specifically, the class verifies document affinity: if the document belongs to the current node, operations proceed with document locking to prevent race conditions, and then unlocking upon completion. If not, the operation is forwarded to the owner node for execution with the same considerations and subsequently broadcasted to all other nodes.

Delving into the `CRUDHandler` class details:

4- CRUD operations, such as `createData`, enable the creation of a new document after verifying its availability and validation. Methods like `updateData` replace old data with user-provided data after verifying document affinity, while `deleteData` removes a selected key-value pair from a document. The `deleteAllData` method removes an entire document and its associated data from the database, given it belongs to the current node. This class effectively abstracts the intricacies of document locking, node affinity, and broadcasting.

5- Additionally, helper methods like `checkAffinity` and `getNodeAffinity` assist in determining the execution context for operations based on node affinity.

Further details on locking, unlocking, optimistic locking, and indexing will be discussed in subsequent sections.



• Indexing

An indexer is a component or mechanism that creates and manages indexes on data to expedite data retrieval. It allows databases to swiftly locate data without having to scan every document in a collection.

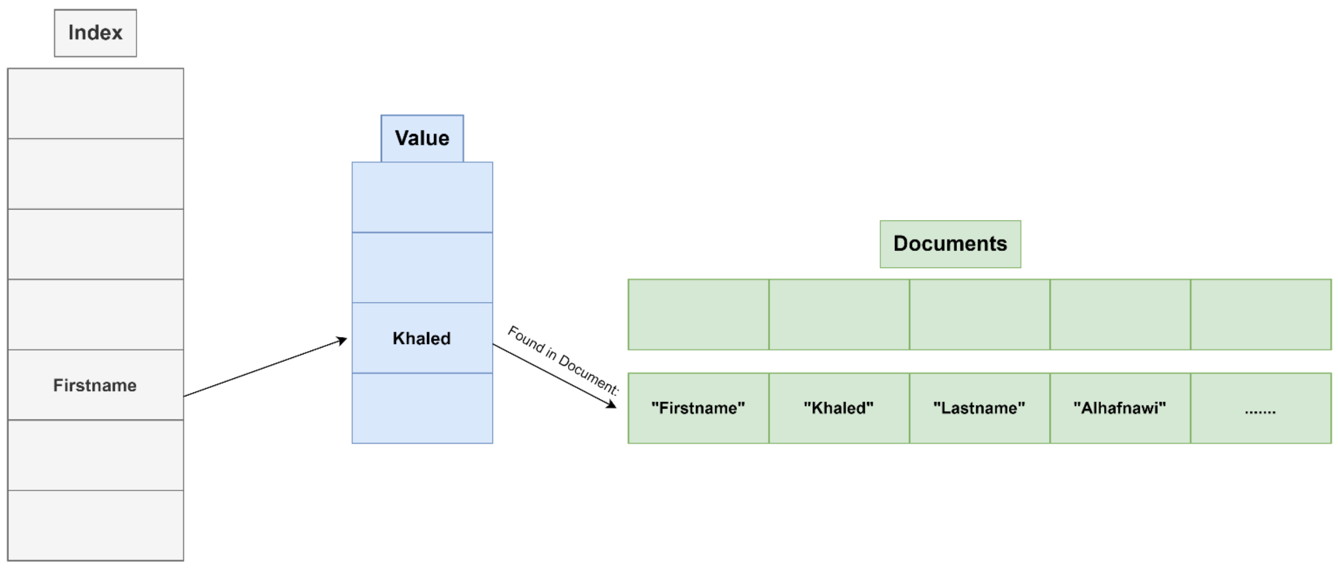
Why it's useful? The primary advantage of having an indexer in a database is **speed**. Consider the index in a book. Instead of reading the entire book to find a specific topic, you can reference the index and jump directly to the page discussing it. Similarly, in an indexed database, instead of scanning every document, the system can utilize the index to promptly locate the necessary data. This not only enhances speed but also improves **efficiency** by reducing the number of data items the system must examine, resulting in faster performance and minimized resource usage.

While there are various ways to implement an index in a project and several types to consider, we chose to implement the Hash index in our project. This choice was motivated by several advantages hash indexes offer over other types. With a read complexity of O(1), which is impressively fast, hash indexes are ideal for scenarios with numerous exact match (equality) queries and datasets that align well with hashing.

Within our project, the details for indexing reside in the index package, comprising two classes. The `AttributeIndex` class introduces an index structure that retains the values of specific attributes and maps them to corresponding documents, promoting swift document retrieval based on attribute values. The core class, `HashIndexer`, `HashIndexer` implements `index` interface, `HashIndexer` delivers a more advanced indexing system centered on hash mapping. This class leverages the `AttributeIndex` class to store and fetch documents by their attributes. Implemented as a Singleton, the `HashIndexer` class ensures only one instance exists throughout the application. Within the `HashIndexer` class, methods such as add (to add to the index), search (to find documents matching a specific attribute), and `searchByAttribute` (to retrieve all documents possessing a particular value for a given attribute) facilitate the application of the indexing mechanism. Additionally, the overridden `toString` method provides a textual representation of the indexed data for debugging and logging objectives.

```
HashIndexer:
Attribute: Firstname
    Value: Khaled
        Found in Document:
{"Firstname":"Khaled","Lastname":"Alhafnawi","id":"c8cb52a6-fb38-4a57-a04c-
ac52bd2a4384","affinity":2}
    Value: waleed
        Found in Document:
{"Firstname":"waleed","Lastname":"Qarni","id":"7690f70a-e4f5-415c-8bd4-
b9459c0a5065","affinity":2}
    Value: Huda
        Found in Document:
{"Firstname":"Huda","Lastname":"Aloosh","id":"1170b38c-0a8d-4faf-b2ed-
867aadd6e787","affinity":1}
    Value: Fahad
        Found in Document:
{"Firstname":"Fahad","Lastname":"Abbadi","id":"8a40127a-82f5-4cac-af0b-
354297171af1","affinity":1}
        .
        .
        .
        .
```

Here an example for how data saved in the Index.



As observed in the above example and figure, the hash index stores each attribute, its corresponding value, and the details of the document file.

Further calculation details can be found in Appendix A.

• Directory Operations.

There are several operations that we must apply to directories, such as creating or deleting a collection or database and loading the database when starting the node. All these operations require interaction with directories. Within the 'directory operation' package, two primary classes handle these operations.

The `DirectoryHandler` class is responsible for creating and deleting directories. Within this class, the `createDirectory` method facilitates the creation of a directory when a database or collection is initiated, while the `deleteDirectory` method deletes the directory when a database or collection is removed. This class also includes methods to retrieve documents from a directory, open directories, and more.

The second class, `DatabaseLoader`, is used to load databases, collections, and documents from the file system based on a directory structure. This class has a primary method named `databaseLoader()`, which initializes the process of loading databases from the root directory specified by `FileConfig.BASE_DIRECTORY`. The loading process unfolds in three stages: The first stage involves loading the databases themselves. The second stage uses the `loadCollections(File database)` method to load the collections contained within the databases. The final stage loads the documents within those collections using the `loadDocuments(String databaseName, File collection)` method.

• Configuration package

his package contains the configuration information, or what we might refer to as protocol details and settings. These are shared among all nodes and are essential for nodes to communicate with each other and with other components. For instance, within this package, we can find the Queries class, settings, and other constant values, such as the number of nodes in the cluster. Let's break it down class by class:

▪ Query class.

The `Query` class is an enumeration (enum) that defines a set of constants representing different query operations related to a database system. Each constant in this enum corresponds to a specific action that can be performed on the database, its collections, or its documents. Here's a breakdown of the constants and their intended operations:

- `CreateData`: Represents a query to create data in the database.
- `UpdateData`: Represents a query to update existing data in the database.
- `DeleteData`: Represents a query to delete specific data from the database.
- `DeleteAllData`: Represents a query to delete all data from a specific part of the database.
- `CreateDatabase`: Represents a query to create a new database.
- `CreateCollection`: Represents a query to create a new collection within a database.
- `DeleteDatabase`: Represents a query to delete an existing database.
- `DeleteCollection`: Represents a query to delete an existing collection from a database.
- `AddDocument`: Represents a query to add a new document to a collection.
- `ListDatabase`: Represents a query to list or view all the databases.
- `ListCollection`: Represents a query to list or view all collections within a specific database.
- `ListDocument`: Represents a query to list or view all documents within a specific collection.

The `Query` enumeration provides a standardized way to reference these database-related operations throughout the application, ensuring consistent naming and easier maintenance.

▪ BootstrapQuery class.

The `BootstrapQuery` class, is an enumeration (enum) that specifies a set of constants representing different query operations related to bootstrapping actions in the system. Each constant in this enum corresponds to a distinct operation that can be executed during the bootstrapping process. Here's a detailed breakdown of the constants:

- `ImportUsernames`: Represents a query operation to import or load usernames into the system during the bootstrapping phase. This could be useful for initializing the system with a predefined list of users or for synchronizing user data from another source.
- `ExportUserDetails`: Represents a query operation to export or extract user details from the system. This used to generate a backup, produce reports, and take the new user information from the bootstrap.

This class also like the `Query` class it promotes consistent naming conventions and simplifies system maintenance.

▪ DBConfig class.

The `DBConfig` class, serves as a configuration utility for the database aspects of the application. This class encapsulates several key configuration constants that are presumably utilized throughout the database layer of the application to ensure consistency and centralization of configuration parameters.

Here's a breakdown of its members:

- `BASE_DIRECTORY`: This constant represents the name of the root directory where the database data is stored. Its value is set to "Database".
- `FILE_EXTENSION`: Specifies the file extension for the database files. Given its value ". json", it implies that the database utilizes JSON-formatted files for data storage.
- `NODES_NUMBER`: This constant indicates the number of nodes in the database system. It's set to 4.

The class is defined as final, which means it cannot be subclassed. Furthermore, it provides a private constructor that throws an `AssertionError` exception.

▪ Postman.

The `Postman` class serves as a utility to streamline the process of sending messages over a network or communication channel. It provides a static method, `sendMessage`, which takes in a PrintWriter object and a message in the form of a string. When invoked, this method sends the given message through the PrintWriter and then flushes the stream, ensuring that the message is immediately transmitted without being held in any buffers. The use of this class simplifies message transmission and ensures consistent behavior across different parts of the application that require message sending functionalities.

▪ AffinityPortsCalculator class.

The `AffinityPortsCalculator` class, is designed to manage and allocate ports for nodes based on their affinity numbers. This class plays a crucial role in facilitating communication between nodes in a distributed system, ensuring that each node has a unique and predictable port for communication.

In the class, we find a Static Initialization Block. This block is used to populate the `affinityPorts` list. This list maintains the ports for all the nodes in the cluster. The ports are computed based on the START_PORT value and the total number of nodes, represented by `DBConfig.NODES_NUMBER`. The `getAffinityPort(int number)` method returns the relevant port from the affinityPorts list. If an invalid number is given (either too low or too high), the method throws an IllegalArgumentException.

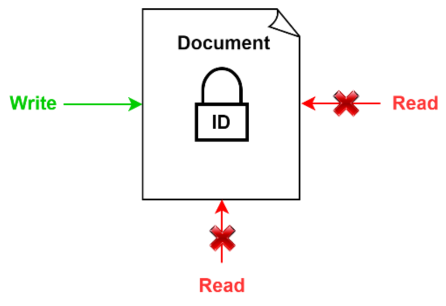
Locks

Locks are crucial for managing access in concurrent computing environments, particularly in multi-threaded or multi-process systems. They ensure that resources, like databases or file systems, are accessed by only one thread at a time, preventing potential data corruption or inconsistencies. When threads access shared resources without synchronization, unpredictable outcomes can occur. For instance, two threads updating the same database record might interfere with each other, leading to data inconsistencies. Once a lock is acquired by a thread, it restricts others from accessing the same resource until it's released. This ensures data integrity and consistency. In the project, there are two main types of locks: the per Node (Read/Write) lock and the Global (Optimistic) lock.

Read/Write Lock.

In our project we find the Lock class which provides a Singleton interface to manage read-write locks identified by string IDs. The Singleton pattern ensures that there's only one centralized map of locks, allowing consistent access and management of locks across different parts of the application using the string ID.

The `getLock` method provides access to a `ReadWriteLock` for a given string identifier. If a lock for the provided ID doesn't exist yet, it's created and stored in the map. If it exists, the existing lock is returned.



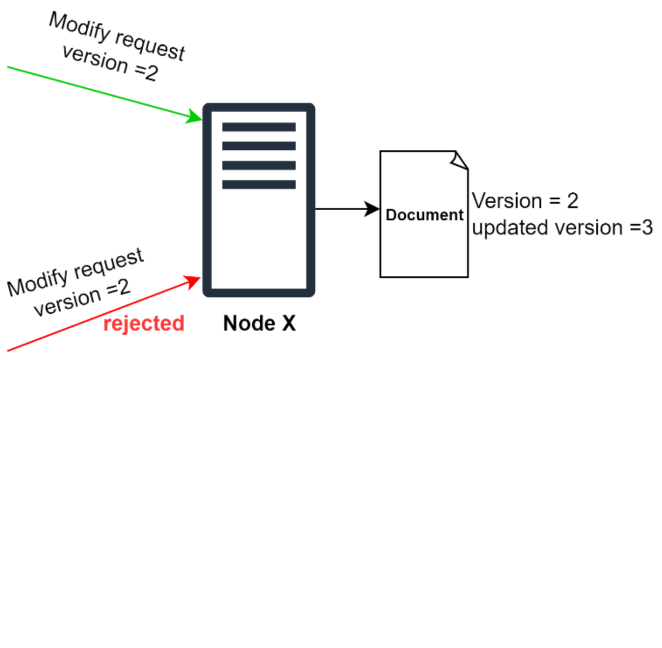
The `deleteLock` method allows for removing a lock from the map based on its string identifier. This can be useful in scenarios where a specific lock is no longer needed, and the application wants to free up resources.

▪ Optimistic Locking.

An optimistic lock, also known as a global lock, is a concurrency control mechanism for databases. Unlike traditional locks that block other transactions until a lock is released, an optimistic lock allows transactions to proceed without interference.

- `getDocument()`: Returns the current Document instance held by the OptimisticLock.
- `setDocument(Document document, int expectedVersion)`: Sets or updates the current Document only if the provided expectedVersion matches the current version. If there's a mismatch, it signifies that another transaction might have modified the Document concurrently, and an error message is printed. If the update is successful, the version number is incremented.
- `getVersion()`: Returns the current version of the Document.

The essence of the `OptimisticLock` class is to prevent dirty writes in scenarios where multiple transactions might be accessing and updating the same Document concurrently. Before updating the Document, a transaction must ensure that the Document hasn't been updated by another transaction since it was last accessed. If there's a version mismatch, the update is rejected.



In the figure, we visualize a scenario involving optimistic locking. The upper modification request arrives first, applies the modification, and updates the version number. When the second request arrives with the old version number, it is rejected.

• Communication protocols

In this project, communication is divided into three parts: 1) Node-to-Node communication, 2) Node-to-Bootstrap communication, and 3) Node-to-User communication. Each of these components plays a crucial role in the database life cycle. What's particularly important is binding the socket to the ports and informing each side about the port number it should use to reach the other side. Let's now delve into the port numbers.

▪ Ports.

We assign node ports in such a way that there's ample scope for scalability in the number of nodes in the future, or for implementing the RAID concept to ensure high availability.

	User Listener	**Node Listener**	**Bootstrap Listener**	**Broadcast Listener**
Node 1	**1500**	**1600**	**1700**	**1800**
Node2	**2500**	**2600**	**2700**	**2800**
Node3	**3500**	**3600**	**3700**	**3800**
…..				
Node N	**(N*1000) +500**	**(N*1000) +600**	**(N*1000) +700**	**(N*1000) +800**

As observed, spacing between the ports is designed for scalability, allowing for the addition of more features, such as RAIDs, in the future. There's also a direct correlation between the Node ID and the assigned ports, ensuring consistent binding and facilitating easier maintenance.

▪ Node.

The `Node` class is primarily designed to manage and represent node-specific information in the network. The use of the Singleton pattern ensures that there's a single instance of this class, thus avoiding potential conflicts or duplication. The methods provided allow for initialization and retrieval of node-specific details, including its status, listener ports, and associated user information.

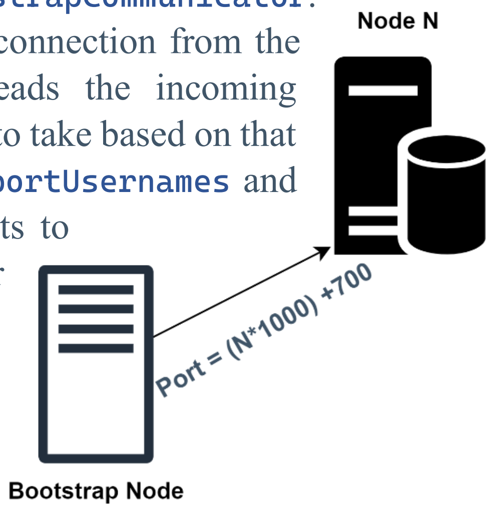
- `getNodeId()`: Returns the node's unique ID.
- `setListenerPorts()`: Initializes the list of listener ports based on the FIRST_PORT and skips the main port.
- `setUsersInfo(int nodeId)`: Sets the path to the user info file based on the provided node ID.
- `getUsersInfo()`: Returns the file containing users' information.
- `getListenerPorts()`: Returns the list of listener ports.

▪ Node-to- Bootstrap communication.

The communication component is implemented by the `BootstrapCommunicator` class, which facilitates communication between nodes and a bootstrap server. The primary functions involve sending user-related data to, and receiving such data from, the bootstrap server. This exchange is essential for assigning new user details or validating existing user requests along with their passwords and roles.

Let's break down the method details:

- `run()`: This is the main loop for the `BootstrapCommunicator`. When executed, it constantly waits for a connection from the bootstrap server. Once connected, it reads the incoming query/command and decides which action to take based on that command. The recognized actions are `ImportUsernames` and `ExportUserDetails`, representing requests to send usernames to the bootstrap server or receive user details from it, respectively.



- `sendUsernameToBootstrap(ObjectOutputStream sendObject)`: This method reads usernames from a file named `Node<NodeId>_Users.txt` located in the Users directory. For each username, it appends the role and the node's ID, and sends this aggregated list to the bootstrap server.
- `receiveUserDetailsFromBootstrap(BufferedReader receiver)`: This method receives user details from the bootstrap server, which includes the username, password, and role. After receiving these details, it prints them out to the console and then uses the `UserValidator` class to append these details to a file.

▪ Node-to-Node communication.

Node-to-Node communication is used to send and receive requests from one node to the affinity node when modifying a document associated with a specific node's affinity number. This type of communication is divided into two parts: the Listener part and the Communicator part. Let's examine each part separately.

NodeListener class

The `NodeListener` class is part of the network package and is responsible for managing communications specific to nodes. The class listens for incoming connections from other nodes and handles various database operations such as updating, deleting, and deleting all data.

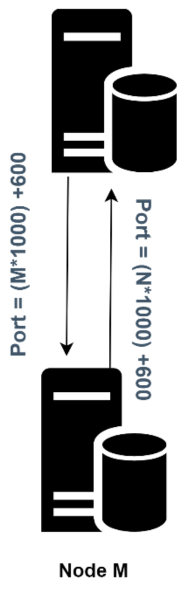
- `run()`: As part of the Runnable interface, the run method constantly listens for incoming connections. When a connection is established, it reads the type of query and its parameters and then delegates the action to the `affinityAction` method. The method handles exceptions and ensures the continuous operation of the listener by enclosing the listening process within a while(true) loop.
- `affinityAction(Query query)`: Based on the type of query received, this method decides which handler method (for CRUD operations) to call.
- `updateDataHandler()`: This method handles the update data operation. It initializes a CRUDHandler instance and triggers the data update action. After the update, a broadcast message is sent to inform other nodes about the change, and a success message is printed to the console.

- `deleteDataHandler()`: Similar to the update handler, this method focuses on deleting a specific set of data. After deletion, it broadcasts the change and prints a success message.
- `deleteAllDataHandler()`: This method deals with deleting all data. After the complete deletion, it sends out a broadcast message and confirms the successful operation with a console message.

NodeCommunicator class

The `NodeCommunicator` class facilitates the process of sending messages from one node to another node (the "affinity node") within a distributed system or network.

- `sendToAffinityNode` method is used to establish a connection to the specified node (determined by the port) and send it a message. After connecting, the method sends the type of query (i.e., the action to be performed, specified by the Query enum) and then sends the actual data (queryName) associated with that query.



▪ Node-to- User communication.

This part applied by UserListener class. UserListener class is designed to handle communication with users, specifically in validating their credentials and then directing them according to their role privileges. This class implements the Runnable interface, allowing it to be used with threads and run concurrently.

- `run()`: This method is executed when the `UserListener` thread is started. It initializes input and output streams (`receiver`, `sender`, `sendObject`) for communication with the user. It reads the username and password sent by the user. Performs user validation using the `UserValidator`. If the user is not valid, it sends an error message. Otherwise, it sends a confirmation message. Retrieves the user's privileges (role) using `PrivilegeCheck`, which checks the user's privileges based on their username. Based on the user's role, it creates an instance of the corresponding Roles subclass (role-specific logic) using the `RoleFactory` and starts the role-specific thread (`user.start()`).

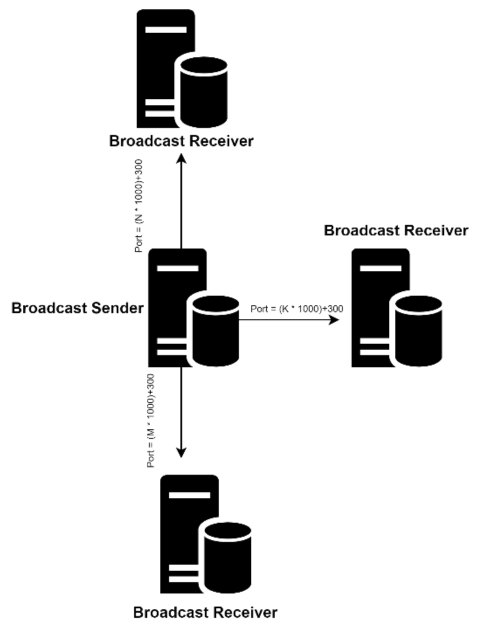
• Broadcasting

Broadcast communication differs from Node-To-Node communication and other types of communication. In broadcasting, updates are done in the affinity node, and all seems well, but the challenge lies in reflecting the change and sharing the new file and updates with all other nodes.

In our project, we find two classes that serve this purpose: the `BroadcastSender` class and the `BroadcastReceiver` class. Let's examine these classes:

▪ BroadcastSender class.

The `BroadcastSender` class provides utilities to broadcast messages and new documents to a set of nodes in a network. The communication details are encapsulated within the `NodeConnection` class. This class iterates through each listener port of the nodes (obtained from `Node.getListenerPorts()`). For each listener port, it attempts to establish a connection by creating a `NodeConnection` object. The port for this connection is the listener port plus 300. The class then sends the query to the connected node. It also sends the `queryName` List, which contains the modification details, to the node. The class leverages the default serialization behavior in Lists. After sending the data, it waits for and then prints the response received from the node. If any exceptions arise during this process, a runtime exception is thrown, encapsulating the original exception as its cause.



▪ BroadcastReceiver class.

The `BroadcastReceiver` class is designed to handle a variety of database operations received as broadcast requests. The operations include updating, creating, and deleting data. The class ensures thread safety by using locks during data modification. The communication with clients is done using sockets, and data serialization and deserialization are managed using the Jackson library.

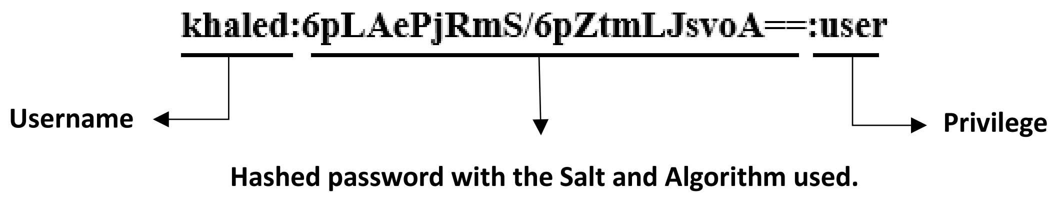
- `run()`: This method is the core logic when the class is executed in a thread. It enters an infinite loop to continuously listen for incoming connections. Upon receiving a new connection, the method sets up input and output streams for data communication. It reads the first line of the received data and interprets it as a `Query` enum value. Then reads an object from the `objectReceiver` and casts it to a list of strings, which populates the `pathNames` attribute. Using a switch statement on the received query, identifies the type of operation requested and delegates the task to the corresponding private methods. Once the operation completes, sends an acknowledgment message back to the sender.

• Security

Security is paramount in databases, both in terms of connections and communication. What differentiates a competent database from an inferior one is the assurance of the integrity of user information and data. It's crucial to ensure that if a file or database is compromised or lost, unauthorized parties cannot access the data and information.

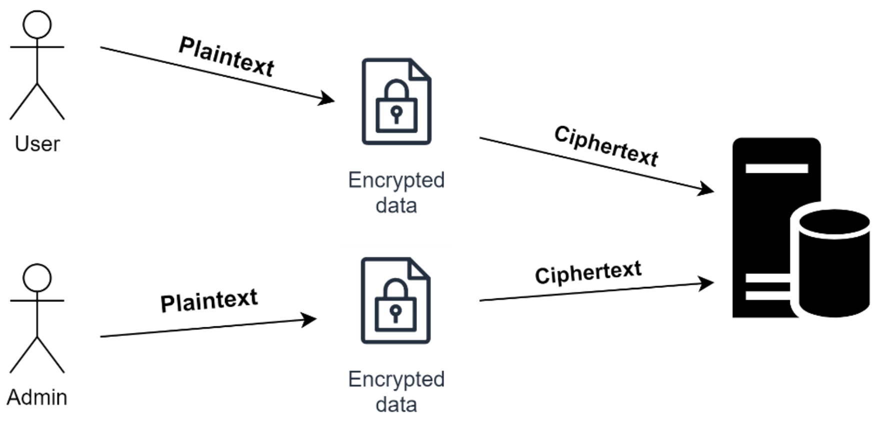
In our project, we emphasize information security. User passwords aren't directly stored in the database. Instead, we hash the passwords using a secure methodology and save them in the user's node file in an encrypted format.

The project contains a validation package, which houses the `UserValidator` class that encapsulates these security details and implements the `Validator` interface. The primary function of the `UserValidator` class is user validation, which it accomplishes by comparing encrypted passwords from a file. This class also provides utility methods to encrypt and decrypt passwords and to add user details to a file. The encryption is based on the **AES** algorithm, and the HMAC-**SHA**256 method derives the encryption key from a passphrase. User details are saved in the database as follows:


khaled:6pLAePjRmS/6pZtmLJsvoA==:user
Username
Privilege
Hashed password with the Salt and Algorithm used.

Let's delve into the class methods:

- `userValidation(String username, String password)`: This method reads the user information from a file, the path to which is fetched from `Node.getUsersInfo()`. Each line in the file, representing a user, is split on the colon : to retrieve the stored username and password. The method checks if the provided username aligns with the stored username and if the encrypted version of the provided password matches the stored password. If there's a match, it returns true; otherwise, false.
- `encrypt(String plaintext)`: This function employs the **AES** cipher to encrypt the given plaintext string (typically a password) using the secret key derived from the passphrase. The output is the encrypted string in base64 format.
- `addToFile(String username, String password, String role)`: After encrypting the provided password, this method writes the username, encrypted password, and role to a file. Each piece of information is separated by a colon :. If any exceptions arise during this process, the error and a corresponding message are printed.



And the `PrivilegeCheck` class provides utilities to determine and verify the privilege level of users, based on information stored in a file. The file is expected to have details about users and their associated privilege levels, and this class offers methods to fetch and check these privilege levels. For example, it can tell whether a specific user is an admin or just a regular user.

Authentication

The Authentication package in our project encompasses three primary classes pivotal to the authentication process. First, the `AdminRole` class caters to users with the 'Admin' role, granting them admin privileges. Next, the `UserRole` class serves users assigned the 'user' role. Both of these classes implement the Roles interface, capitalizing on polymorphism, particularly evident in the `UserListener` class, which discerns privilege using the `getPrivilege()` method found in the `PrivilegeCheck` class. Lastly, the `RoleFactory` class, embracing the Factory design pattern, dynamically determines type based on the privilege.

Delving deeper into class specifics:

Role interface.

The `Roles` interface prescribes a blueprint for subsequent classes, mandating the implementation of the start method. This method's exact purpose or function remains deliberately ambiguous within the interface, maintaining versatility and flexibility in defining roles and their corresponding actions.

UserRole class.

The `UserRole` class is designed to facilitate actions that a user with the "User" role can perform. It provides capabilities to handle CRUD operations like create, update, and delete, and to select specific data entities like databases, collections, and documents. The class ensures efficient data retrieval by leveraging caching mechanisms. All interactions are managed through established input and output streams, allowing seamless communication with a client or user interface.

The Overridden `start()` method, the system continually listens to user input until the user decides for termination (when `anotherQuery` toggles to false). Depending on the received directive, the task is entrusted to the relevant handler method, which could be `handleCreate`, `handleUpdate`, `handleDelete`, `selectDatabase`, `selectCollection`, or `selectDocument`.

▪ AdminRole class.

The `AdminRole` class encapsulates the operations an administrator can perform on the system, such as CRUD (Create, Read, Update, Delete) operations on databases, collections, and documents. The class makes extensive use of `BroadcastSender` to ensure that actions taken by the administrator are communicated across the network or system. This kind of setup is commonly seen in distributed systems where changes made on one node need to be propagated to others. Given its methods and functionalities, `AdminRole` plays a critical role in system management and data handling.

The Overridden `start()` method, It's also continuously waits for user (administrator) input to determine which action to perform. Based on the received choice, it triggers the appropriate handler method. It also handles invalid choices and informs the administrator if they made a wrong choice.

▪ RoleFactory class.

The `RoleFactory` class is designed to produce role-specific objects based on the provided role name. By abstracting the object creation process, this factory provides a centralized point of control, making the code more maintainable and ensuring that objects are created consistently. The class mainly supports two roles: "user" and "admin", each corresponding to the `UserRole` and `AdminRole` classes, respectively.

• Cache

Using caching in our project and specifically employing the Least Recently Used (LRU) caching strategy can offer significant performance benefits and reduce the load on underlying data stores.

Why Use Cache?

Performance Enhancement: One of the primary reasons to use caching is to speed up data retrieval. Reading data from main memory (like RAM) is faster than reading it from a persistent storage system (like hard drives or databases).

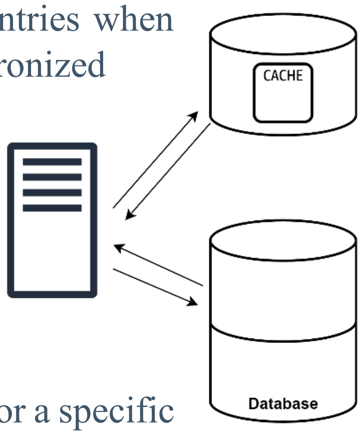
Reduced Latency: Accessing data from cache reduces the latency experienced by the user. Especially in database systems, fetching data from disk or making a network call to retrieve data can introduce significant latency. Caching alleviates this issue.

Consistency and Availability: In distributed systems, caches can provide a level of consistency and availability. If one of the databases or services is down, but its data is available in cache, the system can still serve user requests.

In our project I used the LRU Cache because it's specifically offering a balanced and efficient approach to manage the memory used for caching, ensuring that the cache is utilized effectively.

In the project:

The `LRUCache` is a generic LRU cache implementation built on top of `LinkedHashMap`. This design ensures that the cache size never surpasses a pre-defined capacity, and it removes the least recently accessed entries when this capacity is exceeded. Examining its methods, the synchronized `remove()` function discards outdated data from the cache and guarantees thread safety during removal operations. The `get(K key)` method retrieves the value linked with the given key from the cache, while the `put(K key, V value)` method introduces or updates a cache entry using the provided key and value.



Meanwhile, the `CacheManager` offers a singleton access point for a specific cache instance, ensuring it's shared across all users and connections.

• Bootstrap node

The bootstrap node has two main functions in our Decentralized Cluster-Based NoSQL DB System. Firstly, it's responsible for initializing the cluster and supplying configuration information to the worker nodes. Secondly, it manages new user connections and distributes users across nodes using a load-balancing mechanism. This will be discussed further in the next section. This mechanism determines the node ID and its port number and communicates it to the user to establish a connection with the designated node.

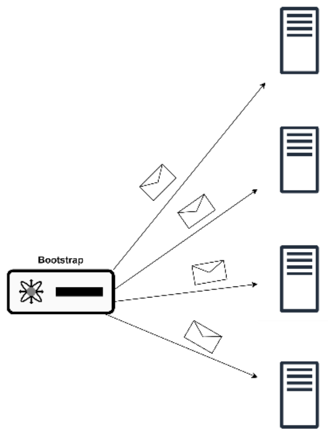
The bootstrap project is divided into three primary packages. The first is the Balancer package, which we'll delve into in the subsequent section. The Functionality package encompasses the core operations of the bootstrap, including importing configurations, exporting user details, and node management methods. Lastly, the `UserHandler` package listens to user requests, processes them, and sends node ID information along with the port number.

▪ Functionality package.

In the Functionality package:

The `BootstrapQuery` class is an enum containing queries related to bootstrapping actions that are executed during the bootstrapping phase, such as `ImportUsernames`, `ExportUserDetails`, `Login`, and `Signup`.

The `Exporter` and `Importer` classes facilitate the transmission of new user details and configurations to worker nodes and the import of usernames and roles from worker nodes, ensuring sign-up requests are validated correctly.



The `FileConfig` class houses the configuration details, like the number of nodes in the cluster.

Lastly, the `RunNodes` class is a utility designed to automate the process of launching multiple Docker containers from a Java application programmatically. Each Docker container represents a "Node" instance, running from a Docker image under the "Khaledhf" namespace on a Docker repository (potentially Docker Hub).

The specific number of nodes to run is set by the `FileConfig.NODES_NUMBER` configuration.

▪ UserListener

The `UserListener` class, designed to operate in a separate thread, is tasked with managing individual user requests related to login and signup procedures. User data is archived in a file titled "UsersDetails.txt". This class utilizes auxiliary utilities like Utility, Exporter, and Importer for a range of tasks, including message transmission and user data storage handling.

- `run()` method: This method is an override from the Runnable interface. When a thread is initiated with a `UserListener` instance, the execution commences within this method. It first discerns an action from the client via the receiver. Based on whether it's a Login or Signup request, the method proceeds correspondingly. In the case of a Login request, the node ID and port number are dispatched to the user to enable a connection with the node. For Signup requests, the `signupNewUser` method is invoked, which checks the validity of the entered username to ensure it's unique. It then receives the password and desired role. Subsequently, a node ID is generated using the load-balancing class, guaranteeing that users are efficiently allocated across nodes.

• Load balancing

The `LoadBalancer` class functions as a Round-Robin load balancer. Its primary purpose is to determine the next node to manage an incoming request, ensuring an even distribution of traffic across all accessible nodes. The class follows the Singleton design pattern, which means only one instance of the load balancer will be created and used throughout the application. This is important to ensure a consistent and balanced load management.

In our project, the load balancer comes into play when a new user signs up. At this point, the user needs to identify the node ID and the corresponding port number to connect to and affinity number. This is where the Round-Robin technique is applied, ensuring that every new user is efficiently allocated to the nodes.

Why Round Robin?

Round Robin is a popular load balancing method due to its:

- **Simplicity**: Easy to implement and understand.
- **Predictability**: Provides an even distribution of requests.
- **Fair Distribution**: Ideal when servers have similar capacities.
- **Minimal Overhead**: Quick decision-making due to its straightforwardness.

```
User 1 is assigned to Node_1
User 2 is assigned to Node_2
User 3 is assigned to Node_3
User 4 is assigned to Node_4
User 5 is assigned to Node_1
User 6 is assigned to Node_2
```

The following is an illustration of how Round Robin distributes users among the Nodes.

• Database client

The "Database client" project facilitates user interaction, effectively serving as the user interface. Within this project, users communicate with the bootstrap to login or sign up. Based on their role, they can forward queries to the worker node. It's crucial to synchronize all read and write requests and messages, whether interacting with the bootstrap during the login phase or with the worker node, to optimize the user experience.

The project is divided into three packages: one for bootstrapping communication, another for configuration, and the last one named "Windows", which contains details of various interface windows, such as the admin, user, and welcome windows.

▪ BootstrapCommunicator.

The `BootstrapCommunicator` class offers a means for the client-side to communicate with the Bootstrap server. It streamlines user login, signup, and username verification processes. Using the Singleton design pattern, it ensures only one instance exists throughout the application's lifespan.

- `getNodeID(String username)` method: Retrieves the associated Node ID from the load balancer.
- `usernameValidator(String username)` method: Verifies the given username by consulting the Bootstrap server, ensuring its availability for sign-up..
- `signupNewUser(String username, String password, String role)` method: To sign up a new user by sending the necessary credentials to the Bootstrap server..

▪ Configurations.

This package contains configuration classes. For instance, it includes queries that's for create, delete, update, and read on the worker node and Bootstrap-specific actions like login and signup. Utilities, like the `sendMessage` method, define the approach to send and instantly process messages to the recipient.

▪ Windows.

The "Windows" package plays the role of the user interface. Initially, users encounter the login window to specify their desired action - login or signup. Once users log in and request their privilege level from the bootstrap, the system utilizes polymorphism. Depending on the received user privilege, an object of the appropriate role class gets instantiated using the Factory design pattern.

- `WelcomeWindow` class: As the primary interface for users, it simplifies the login and signup processes. It uses clear prompts and feedback to guide users.
- `UserWindow` class: Serves as an intuitive interface for users to interact with the database. It abstracts technical details and offers a user-friendly, menu-driven interface.
- `AdminWindow` class: Tailored for administrators, it allows them to perform extensive operations on the database, including managing the database's structure.
- `WindowsFactory` class: This factory class dynamically creates and provides instances of different window classes, catering to specific user roles. This approach embodies the Factory Method design pattern.

To make it more user-friendly, I developed version 2 of the Database client using the JavaFX mechanism, which enhances the user experience and makes the interface more appealing.

• Code Testing

In the project, I conducted testing through various methods:

Automated Testing: I employed JUnit to assess the correctness of various structures, including optimistic locking.

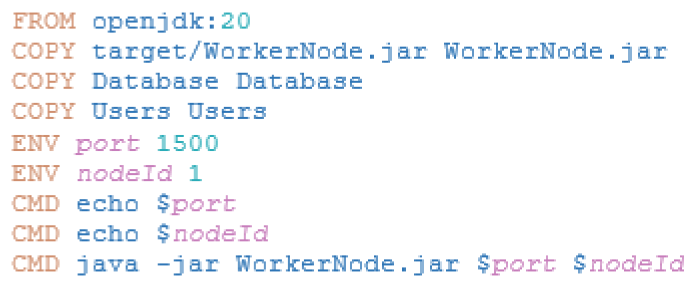
Manual Testing: I utilized IntelliJ's debugging tools to inspect connections and verify the correctness of the project's lifecycle.

• DevOps

▪ Docker.

For efficiency, I used docker to run each VM or node in a separate container.

Docker file



```
FROM openjdk:20
COPY target/WorkerNode.jar WorkerNode.jar
COPY Database Database
COPY Users Users
ENV port 1500
ENV nodeId 1
CMD echo $port
CMD echo $nodeId
CMD java -jar WorkerNode.jar $port $nodeId
```

the following is a snapshot of the Dockerfile of worker node id 1.

▪ Git and GitHub.

I used git and GitHub as an online repository for the project, to keep track of changes happening to the codebase.

▪ Maven.

I used Maven in my project to manage dependencies and plugins. It automates my project’s build process and packages it into a JAR file for use in a Docker container.

• Clean Code principles (Uncle Bob)

- **Meaningful Names**: Variables, classes, and methods are named descriptively to communicate their intent, e.g., `BootstrapCommunicator`, `getNodeID`, `databaseName`, etc.
- **Switch Statements**: Both `AdminWindow` and `UserWindow` use switch statements to handle different user actions, improving readability and making it easier to add or remove options in the future.
- **Encapsulation**: Instance variables in the classes are set to private, ensuring that they can only be accessed and modified through methods in the same class. This hides the internal details and provides a public interface for interaction.
- **Use of Enums**: Enums like `BootstrapQuery` and `Query` are used to define a fixed set of related constants, ensuring type safety, and making code more readable.
- **Error Handling**: In several places, exceptions are caught and handled. For example, catch (IOException e) is used to handle potential IO errors, with meaningful messages or actions in response.
- **Avoiding Magic Numbers or Strings**: Instead of hard-coding numbers or strings, meaningful constant names, or configurations, like `PortConfig.BOOTSTRAP_PORT`, are used.

- **Separation of Concerns**: Classes are organized into different packages based on functionality, like windows for GUI windows and communicator for communication functionalities.
- **Clear Structure**: Classes are structured with clear sections for instance variables, constructors, and methods, making the code layout predictable.
- **Use of Comments (sparingly)**: While the code mostly relies on self-explanatory naming, in the few cases where comments might be helpful (like for complex logic), they could be added. However, the provided code has minimal comments, suggesting the code is meant to be self-documenting.
- **Consistency**: The code formatting, naming conventions, and structure are consistent across classes, making the codebase easier to understand and maintain.

• Effective Java Items (Jushua Bloch)

item 1: Consider static factory methods instead of constructors: This is seen in the use of the factory methods in `RoleFactory` and `WindowsFactory` classes. These methods return instances of classes based on certain conditions.

Item 3: Enforce the singleton property with a private constructor or an enum type: `BootstrapCommunicator`, `LoadBalancer`, and `CacheManager` all implement the Singleton pattern, ensuring that only one instance of the class can exist in the application. This is achieved using a private constructor and a static `getInstance()` method.

Item 4: Enforce non insatiability with a private constructor: While not explicitly shown, classes with utility methods (like a hypothetical Utility class) might follow this principle by having a private constructor to prevent instantiation.

Item 6: Avoid creating unnecessary objects: The Singleton pattern, as used in `LoadBalancer` and `BootstrapCommunicator`, avoids creating unnecessary instances by reusing the single instance.

Item 17: Minimize mutability: Some classes, like `LoadBalancer`, have attributes that are set once and are not meant to be changed, making them effectively immutable.

Item 19: Use interfaces only to define types: The Window interface is used to define a type that has the `popupWindow` method. Implementing classes (`AdminWindow`, `UserWindow`, etc.) provide concrete implementations.

Item 23: Prefer class hierarchies to tagged classes: Instead of using a single class with a tag to distinguish between different behaviors, the provided code uses a hierarchy. For example, `AdminWindow` and `UserWindow` both implement the Window interface to provide different behaviors based on user roles.

Item 57: Use exceptions only for exceptional conditions: Exceptions, like IOException, are caught in several places in the provided code, and they are handled by either throwing a new RuntimeException or by providing a user-friendly message.

Item 64: Refer to objects by their interfaces: The code uses more general types (like `BufferedReader` and `PrintWriter`) to refer to objects, even if the actual implementation might be specific (like `FileReader` or `FileWriter`). This promotes flexibility and adheres to the principle of programming to an interface.

- **Item 78: Synchronize access to shared mutable data**: The `getInstance` method in the `BootstrapCommunicator` class uses the synchronized keyword, ensuring thread-safe access to shared mutable data.

• SOLID principles

Single Responsibility Principle - A class should have one, and only one, reason to change. For example: The classes such as `LoadBalancer`, `BootstrapCommunicator`, and `CacheManager` appear to be focused on specific tasks (load balancing, communicating with the bootstrap, and managing cache, respectively).

Open/Closed Principle - Software entities should be open for extension but closed for modification. For example: The use of interfaces like Window ensures that the code is open for extension (you can always add more implementations) but closed for modification (you don't have to change the Window interface every time you add a new type of window).

Liskov Substitution Principle - Subtypes must be substitutable for their base types. For example: `AdminWindow` and `UserWindow` both implement the Window interface. This means that wherever a Window type is expected, any of its implementations (`AdminWindow`, `UserWindow`, or any future windows) can be used without breaking the program.

Interface Segregation Principle - Clients should not be forced to implement interfaces they do not use. For example: The Window interface provides a singular `popupWindow` method, which makes it focused.

Dependency Inversion Principle - High-level modules should not depend on low-level modules. The use of the Window interface in `WindowsFactory` is a clear example of depending on an abstraction rather than a concrete implementation. The factory doesn't need to know the specifics of `AdminWindow` or `UserWindow`; it only needs to know about the Window interface.

• Design Patterns

Singleton Pattern: Ensures a class has only one instance and provides a global point of access to this instance. `CacheManager` uses the Singleton pattern to ensure that there's only one instance of the `LRUCache`. `LoadBalancer` and `BootstrapCommunicator` also exemplify the Singleton pattern.

Factory Pattern: Provides an interface for creating instances of a class, with its subclasses deciding which class to instantiate. `WindowsFactory` creates an instance of a Window (`AdminWindow` or `UserWindow`) based on the provided role. `RoleFactory` determines which Role to return based on the given string descriptor.

Strategy Pattern: This pattern defines a set of algorithms, encapsulates each one, and ensures they are interchangeable. The Window interface acts as the strategy interface, with its implementations (`AdminWindow`, `UserWindow`) serving as concrete strategies. This approach allows for varying algorithms or processes to display a window, which can be interchanged at runtime. It's also straightforward to introduce additional roles, such as a viewer.

Template Method Pattern: This pattern specifies the framework of an algorithm, permitting its individual steps to be realized by child classes without modifying its overall structure. The `popupWindow` method in the Window interface, as realized in classes like `AdminWindow` and `UserWindow`, can be interpreted as a sequence:

displaying a welcome message, presenting role functionality, and then displaying a concluding message.

Proxy Pattern: This pattern introduces a substitute or stand-in for another object to regulate access to it.

• The data structures used.

Arrays: Arrays stand as the foundational data structure and find application in various contexts.

Lists: The code utilizes the `ArrayList` to manage the list of nodes in the `LoadBalancer` class. In a similar vein, it's employed to track database names, collection names, and document names within the `AdminRole` class. Additionally, its inherent Serializable behavior is advantageous for sending and receiving objects.

LinkedHashMap: The `LRUCache` class capitalizes on the `LinkedHashMap` to actualize the least recently used (LRU) caching strategy. This structure's ability to maintain order is pivotal to its operation.

HashMap: In the `HashIndexer` class, we utilize a `hashmap` to store the index along with its values. And in `AttributeIndex` class also.

• Conclusion

In our project, we delved into Decentralized Cluster-Based NoSQL DB System and the intricacies of database implementation, exploring the structural hierarchy and the interactions among documents, collections, and database classes. The utility of handler classes, especially the CRUDHandler class, is evident in facilitating operations like creating, updating, reading, and deleting. We emphasized the significance of checking the affinity number prior to any document modifications. Our approach to solving local race conditions through Locks and the application of optimistic locks to address global race conditions were meticulously discussed. The project highlighted the pivotal role of indexers in databases, emphasizing their efficiency in data retrieval compared to linear searches. We navigated through various communication types - Node-to-Node, Node-to-Bootstrap, and Node-to-User, illustrating the essence of broadcasting in updating and reflecting modifications across nodes. The project underscores the importance of security, demonstrating encryption and decryption mechanisms and secure storage of user details. We delved into the bootstrap tasks a, from initiating the cluster to handling user requests. The application of load balancing, ensuring efficient user-to-node assignment, was explored in detail. User interactions with the system were enhanced by adopting JavaFX in the second version, elevating user experience. The integration of DevOps, especially through containerization using Docker, emphasized our commitment to efficiency. Lastly, the project epitomizes adherence to coding best practices, embodying the principles of clean code, Effective Java, and SOLID. The deliberate implementation of various design patterns and judicious use of data structures underscore our strategic approach to crafting a robust and efficient Decentralized Cluster-Based NoSQL DB System.

• Appendix A

Time Combplexity:

- **Without the Index**

	Best	Average	Worst
Insertion	**O(1)**	**O(n)**	**O(n)**
Deletion	**O(1)**	**O(n)**	**O(n)**
Lookup	**O(1)**	**O(n)**	**O(n)**

- **With the Index**

	Best	Average	Worst
Insertion	**O(1)**	**O(1)**	**O(n)**
Deletion	**O(1)**	**O(n)**	**O(n)**
Lookup	**O(1)**	**O(1)**	**O(n)**

Memory Efficiency:

If you have A unique attributes, each with V unique values the space complexity for the indexing structure is roughly:

$$O(A \times V)$$

Instead of saved each value and attribute that's need approximately:

$$O(A \times A)$$