Atypon Training Program 2023

JAVA AND DEVOPS BOOTCAMP (FALL 2022) CAPSTONE PROJECT DECENTRALIZED CLUSTER-BASED NOSQL DB SYSTEM

BY: YUSSIF ABDALLA, The University of Jordan

I. ABSTRACT

This report illustrates the solution of the Atypon's 2023 assignment. The project requires to use Java to build an application that simulates the interaction between users and nodes inside a decentralized NoSQL DB cluster. Complete description of the application requirements and constraints maybe found in the capstone project description. This final projects integrates all the previous projects into a complete capstone application based project. In this work, detailed design and verification of the application highlighting the DB implementation, the data structures used, the multithreading and locks engineered, the data consistency issues in the DB, the node hashing and load balancing, the communication developed protocols between nodes, and the safety and security issues.

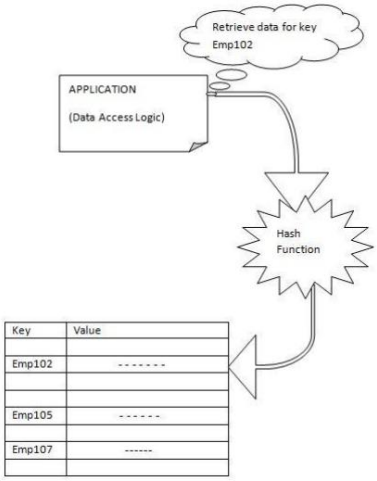
The design, implementation and code testing and verification utilizes standard techniques such as; the clean code principles (Uncle Bob), "Effective Java" Items (Jushua Bloch), the SOLID principles, design patterns and DevOps practices.

II. INTRODUCTION

NoSQL, also referred to as "not only SQL", "non-SQL", is an approach to database design that enables the storage and querying of data outside the traditional structures found in relational databases. While it can still store data found within relational database management systems (RDBMS), it just stores it differently compared to an RDBMS. Instead of the typical tabular structure of a relational database, NoSQL databases, house data within one data structure, such as JSON document. Since this non-relational database design does not require a schema, it offers rapid scalability to manage large and typically unstructured data sets. NoSQL is also type of distributed database, which means that information is copied and stored on various servers, which can be remote or local. This ensures availability and reliability of data. If some of the data goes offline, the rest of the database can continue to run [1][2].

A NoSQL database manages information using any of these primary data models [3-5]:

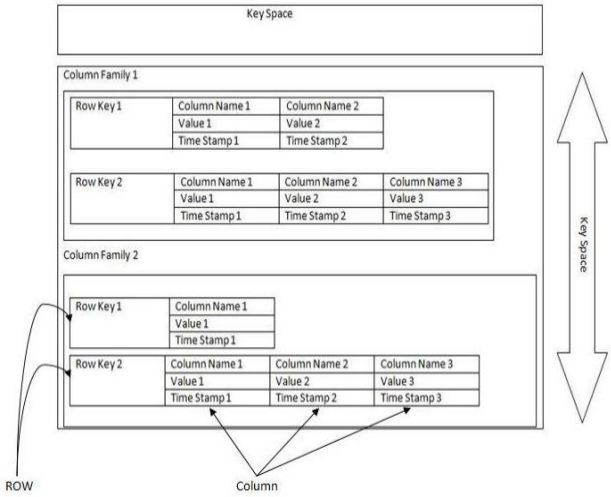
Key-value store: This is typically considered the simplest form of NoSQL databases. This schema-less data model is organized into a dictionary of key-value pairs, where each item has a key and a value. The key could be like something similar found in a SQL database, like a shopping cart ID, while the value is an array of data, like each individual item in that user's shopping cart. It's commonly used for caching and storing user session information, such as shopping carts. However, it's not ideal when you need to pull multiple records at a time. Redis and Memcached are examples of an open-source key-value databases.



Document store: As suggested by the name, document databases store data as documents. They can be helpful in managing semi-structured data, and data are typically stored in JSON, XML, or BSON formats. This keeps the data together when it is used in applications, reducing the amount of translation needed to use the data. Developers also gain more flexibility since data schemas do not need to match across documents (e.g. name vs. first_name). However, this can be problematic for complex transactions, leading to data corruption. Popular use cases of document databases include content management systems and user profiles. An example of a document-oriented database is MongoDB, the database component of the MEAN stack.

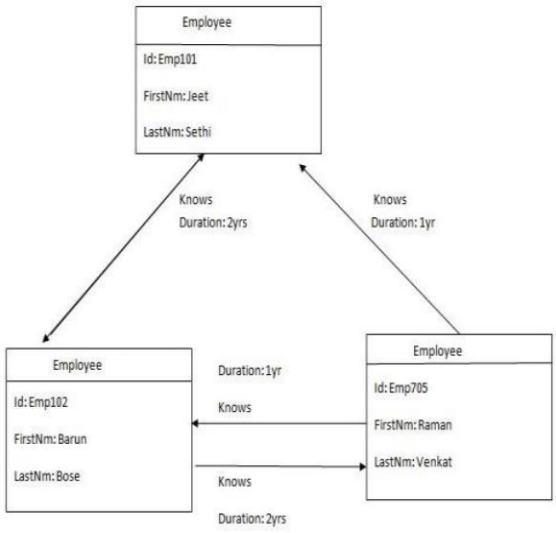
Key (Employee ID)	Document
Emp101	FirstNm:Jeet LastNm:Sethi Age:26 ..
...	...
....	....
Emp705	FirstNm:Meera LastNm:Patnaik Age:22 ...

Wide-column store: These databases store information in columns, enabling users to access only the specific columns they need without allocating additional memory on irrelevant data. This database tries to solve for the shortcomings of key-value and document stores, but since it can be a more complex system to manage, it is not recommended for use for newer teams and projects. Apache HBase and Apache Cassandra are examples of open-source, wide-column databases. Apache HBase is built on top of Hadoop Distributed Files System that provides a way of storing sparse data sets, which is commonly used in many big data applications.



Apache Cassandra, on the other hand, has been designed to manage large amounts of data across multiple servers and clustering that spans multiple data centers. It has been used for a variety of use cases, such as social networking websites and real-time data analytics .

Graph store: This type of database typically houses data from a knowledge graph. Data elements are stored as nodes, edges and properties. Any object, place, or person can be a node. An edge defines the relationship between the nodes. For example, a node could be a client, like IBM, and an agency like, Ogilvy. An edge would be categorized the relationship as a customer relationship between IBM and Ogilvy. Graph databases are used for storing and managing a network of connections between elements within the graph. Neo4j (link resides outside IBM), a graph-based database service based on Java with an open-source community edition where users can purchase licenses for online backup and high availability extensions, or pre-package licensed version with backup and extensions included.



There are plenty of advantages of NOSQL over Relational database, such as; providing a wide range of data models to choose from, easily scalable, database administrators are not required, some of the NOSQL DBaaS providers like Riak and Cassandra are programmed to handle hardware failures, faster, more efficient and flexible.

On the other hand, there are some disadvantages of NOSQL over Relational, such as; immature, no standard query language, some NOSQL databases are not ACID compliant, no standard interface and the Maintenance is difficult [1-5].

In this work assignment, it is required to use Java to build an application that simulates the interaction between users and nodes inside a decentralized NoSQL DB cluster. Complete description of the application requirements maybe found in the assignment description, where this assignment integrates all the previous assignments into a complete capstone project. Furthermore, a detailed report that includes information about the following must be presented: DB implementation, the data structures used, multithreading and locks, data consistency issues in the DB, node hashing and load balancing, communication protocols between nodes, security issues, and code testing [6].

Finally, the assignment required to defend the code against: the clean code principles (Uncle Bob), “Effective Java” Items (Jushua Bloch), the SOLID principles, design patterns and DevOps practices [6].

III. DB IMPLEMENTATION

3.1 Database Cluster System Design

A decentralized NoSQL cluster has been established, consisting of Spring application nodes. Each node within the cluster hosts its own set of REST APIs for executing queries and performing various operations, including login and initialization.

A bootstrap node is responsible for launching the cluster, orchestrating the startup process of all nodes, and assigning users to their designated nodes.

Users communicate with the cluster using a Java command-line application, where each command the user enters is translated into an API request and sent to the cluster.

Figure 1 depicts the proposed models of the cluster's system design.

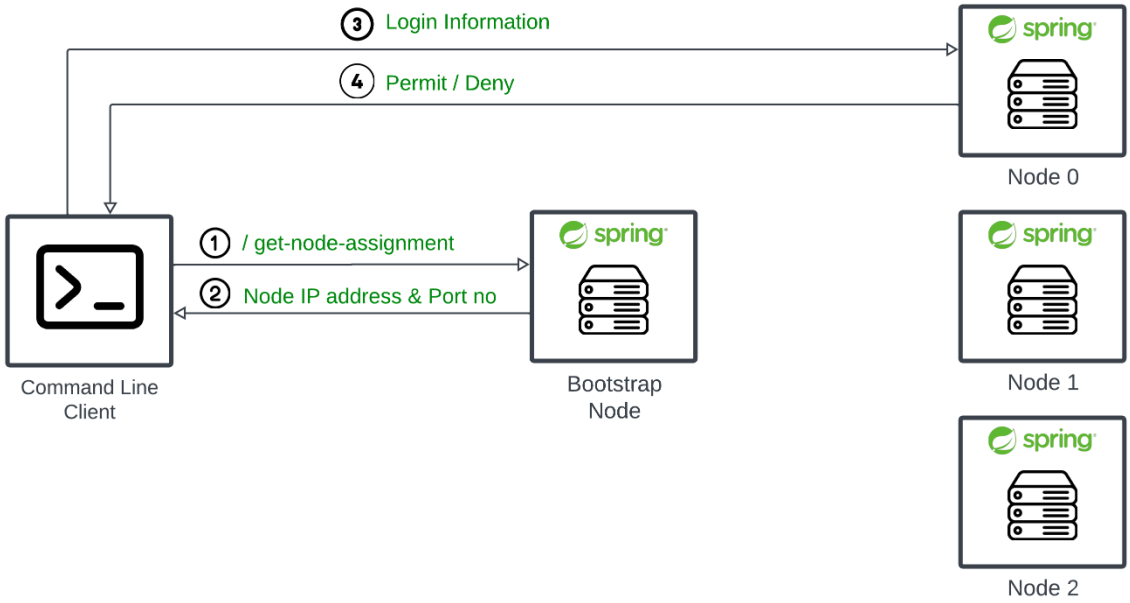


Figure 1: Cluster's system design

3.2 The node Spring application designs

Each node in our system is implemented as a Spring application. These Spring applications comprise REST controllers for building RESTful web services and services that organize and encapsulate business logic or application-specific functionality. This design is crucial for promoting modularity, maintainability, and the separation of concerns.

3.2.1 The Bootstrap Node Application Design

Figure 2 displays the main controllers and services within the bootstrap node Spring application with detailed explanations provided throughout the report.

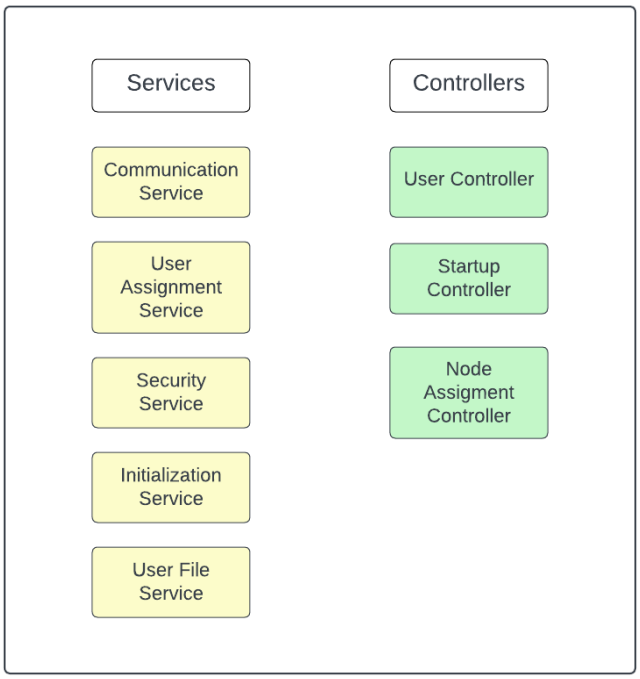


Figure 2: The Bootstrap Controllers and Services

User Controller:

Description: The User Controller exposes APIs for user management.

Services Used: User File Service (for adding users to the users file and), Security Service (for user authentication and authorization)

Startup Controller:

Description: The Startup Controller handles cluster initialization. It initiates the startup of the cluster and sends the list of predefined users to other nodes.

Services Used: Initialization Service (for initializing the cluster and distributing predefined user data and node information data).

Node Assignment Controller:

Description: The Node Assignment Controller deals with node assignments. It responds to API requests from clients, providing information about node assignment, including a node's domain name and port number.

Services Used: User Assignment Service (for facilitating node assignments).

3.2.2 The Cluster Node Application Design

Figure 3 displays the main controllers and services within the ordinary nodes in the cluster with detailed explanations provided throughout the report.

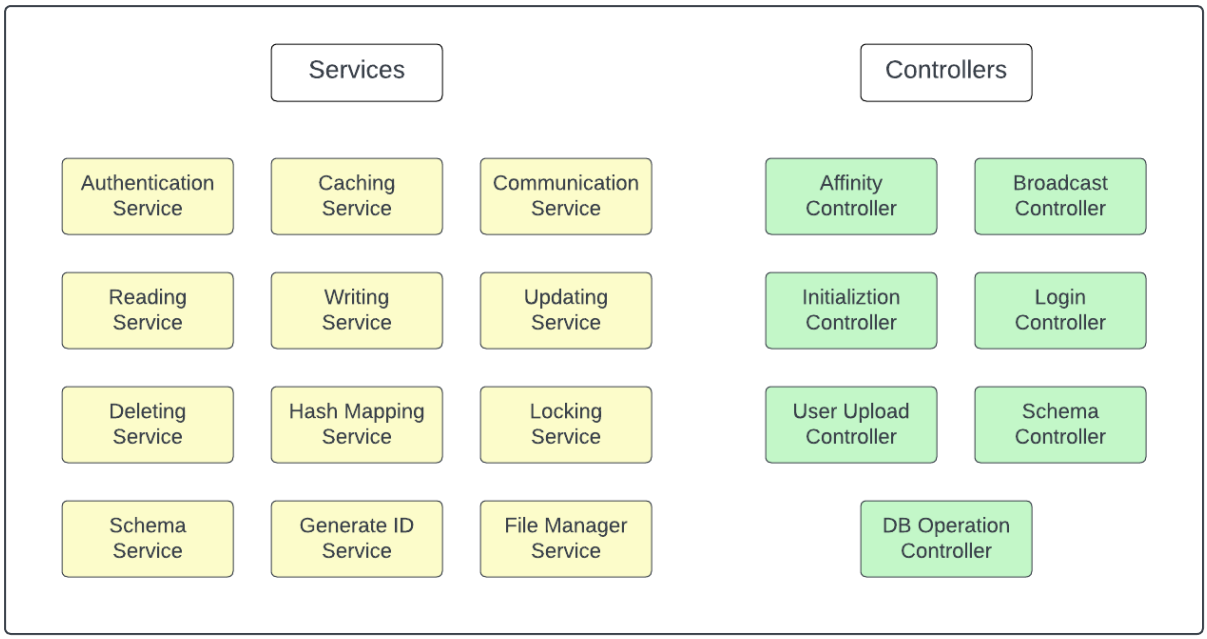


Figure 3: The Cluster Node's Controllers and Services

Initialization Controller:

Description: The Initialization Controller retrieves the list of nodes in the cluster from the bootstrapping node and stores this information in the Communication Service. This data is subsequently utilized by other services.

Services Used: Communication Service (utilized for storing cluster node information, which is later accessed by various services).

User Upload Controller:

Description: The User Upload Controller is used when creating a new user. It broadcasts this information to the bootstrap node and then writes the new user's details to the user file.

Login Controller:

Description: The Login Controller handles user login by accepting usernames and passwords.

Services Used: Authentication Service (employed for user validation based on usernames and passwords).

DB Operation Controller:

Description: The Database Operation Controller handles database operation requests and utilizes different services depending on the type of operation. Within the Database Operation Controller, there are two distinct request handlers: one exclusively performs the operation, while the other performs the operation and broadcasts it to other nodes in the cluster.

Services Used:

- Writing Service (for writing documents to the database)
- Reading Service (for reading documents and properties from the database)
- Updating Service (for updating properties)
- Deleting Service (for deleting documents from the database)
- Locking Service (for resolving race conditions in operations prone to race conditions)

Schema Controller:

Description: The Schema Controller manages database structure-related operation requests, including creating, deleting, and utilizing collections and databases.

Services Used: Schema Service (employed for preserving the current collections and databases in use).

Broadcast Controller:

Description: The Broadcast Controller handles database operation requests by broadcasting them to all nodes in the cluster, excluding the current node.

Services Used: Communication Service (for retrieving information about cluster nodes, such as domain names and port numbers)

Affinity Controller:

Description: The Affinity Controller processes database operation requests and routes them to the affinity node responsible for the operation.

Services Used: Hash mapping Service (for determining the affinity node handling write/update operations), Generate ID Service (for generating new IDs for document requests during client write operations).

3.3 The command-line implementation

The command line accepts user commands and validates them by querying a HashMap that associates the command's syntax with a corresponding command class implementing the command interface. This approach utilizes the command design pattern, which is further explained in the design pattern section, to execute the command. In the execute method of command classes, the required API request is generated by extracting information from the user's command, including parameters like IDs and names. Figure 4 and Table 1 illustrate the command line proposed implementation.

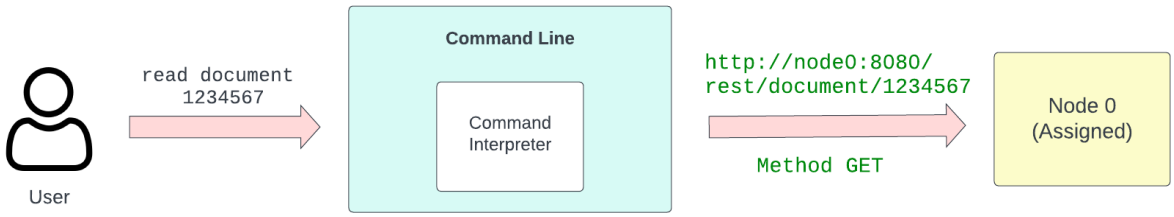


Figure 4: Command line implementation

Table 1: A list of valid commands along with their respective descriptions

start cluster	starts and initializes the cluster
Login	logs into an assigned node by username and password
write document *<document>*	create a new document
read document *<id>*	read a document by ID
read document *<id>* property *<property-name>*	read a document's property
update property *<id> <property>*	update a document's property
delete document *<id>*	delete a document by ID
create collection *<collection-name>*	create a collection
delete collection *<collection-name>*	delete a collection
use collection *<collection-name>*	use a collection
list collection	list the collections in the database
create database *<db-name>*	create a database
delete database *<db-name>*	delete a database
use database *<db-name>*	use a database
list database	list databases in the system

3.4 Operation Workflow of the Database Cluster System

The initial user interaction with the cluster begins at the bootstrap node, which is responsible for launching and coordinating the startup process of all nodes. Additionally, it efficiently assigns existing users to specific nodes to ensure load balancing across the entire cluster. If a new user wishes to join, they initiate communication with the bootstrap node to obtain login credentials and determine their designated node. Once successfully logged in to their assigned node, users can submit queries to the database using the respective node's REST API. These operations are then propagated to all the nodes, ensuring the replication of data, schemas, and indexes throughout the entire cluster. Please check Figure 5 for a visual representation of the operational workflow of the database cluster system.

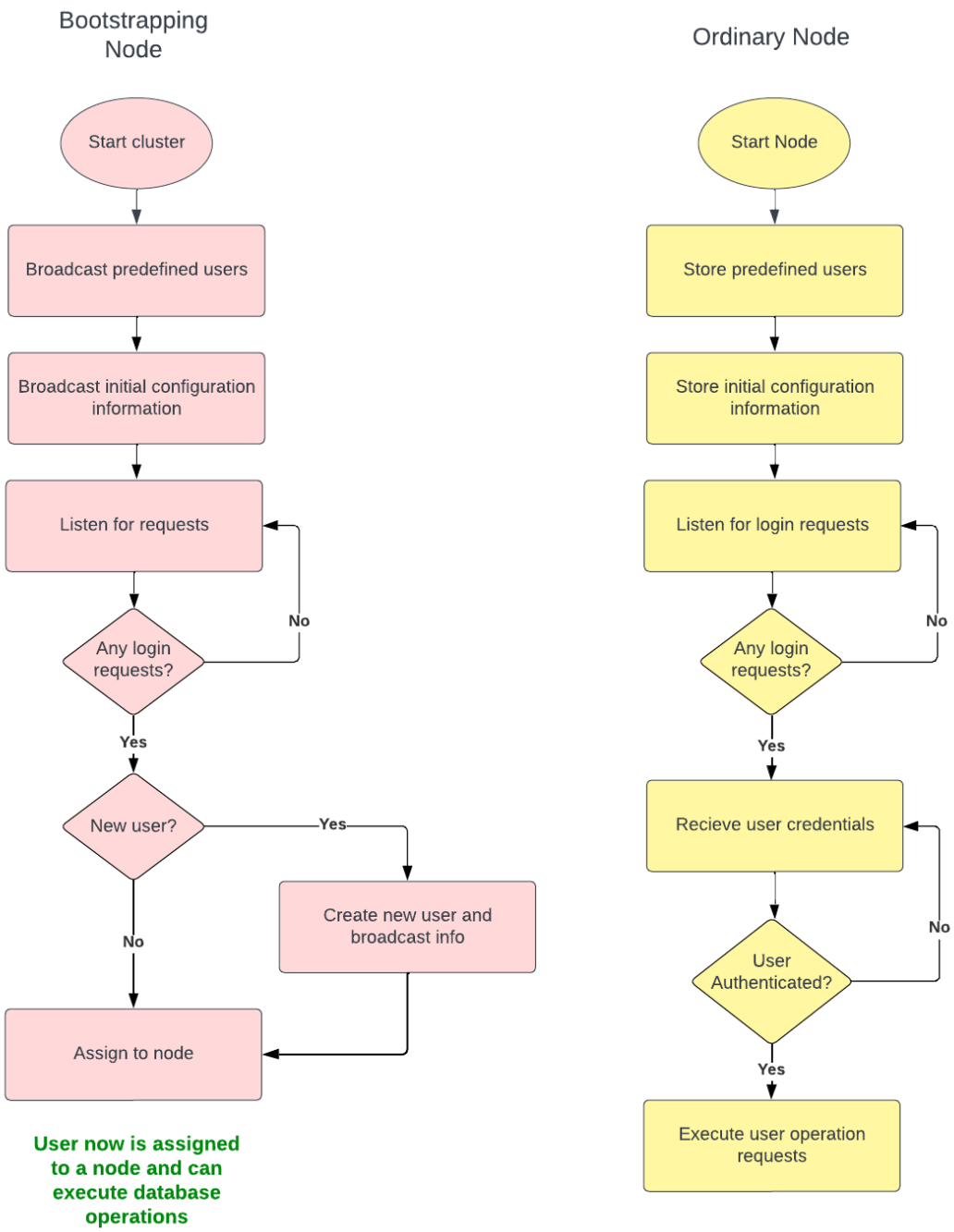


Figure 5: the operational workflow of the database cluster system

3.5 Broadcast predefined users

Predefined user credentials are stored in a **user.csv** file for simplification. This file includes usernames, hashed passwords, and a unique salt generated for each password. The content of the **user.csv** file is then distributed to all nodes using the **Initialization**

Service in the bootstrap node through POST requests during the initialization stage of the nodes.

3.6 Broadcast Initial Configuration Information

In the initial phase, only the bootstrap node possesses information about the cluster's nodes. However, during the initialization stage, the bootstrap node shares crucial node details, including domain names and port numbers, with all nodes within the cluster.

3.7 Database Schema Design

The NoSQL database is organized as a file system within each node, following a hierarchical structure. A database serves as the top-level entity, containing multiple collections. Each collection consists of a set of buckets, which are essentially plain text files designed to store documents. A document is a Json that contains key-value pairs known as properties. Please refer to Figure 6 for a better understanding of the NoSQL database design.

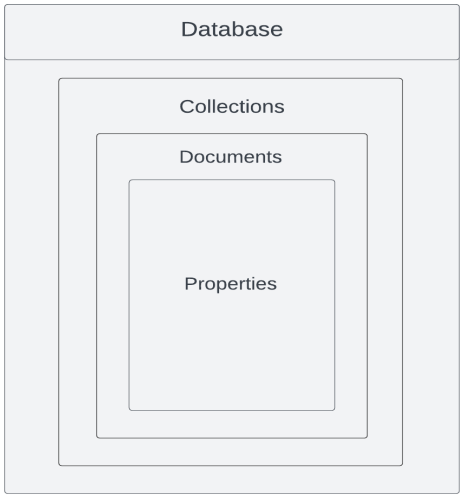


Figure 6: The NoSQL database design

3.8 Document Hashing and Indexing

Indexing is a data structure technique that enhances database performance by reducing the number of disk accesses needed to satisfy a query. It is a critical component for databases to efficiently retrieve requested data. This is why indexing is implemented at both the document and property levels in the database cluster system.

To implement indexing, collections are divided into buckets. Each bucket is further subdivided into blocks, with each block serving as a storage location for an individual

document. At the time of performing database operations, the document ID's hash is segmented to determine both the bucket number and the corresponding offset, represented as the bucket block number. Documents are allocated to these buckets and blocks through calculations applied to their IDs. After segmenting the ID, we calculate the modulus of these segments with respect to the available buckets and blocks. Figure 7 summarizes the DB layout.

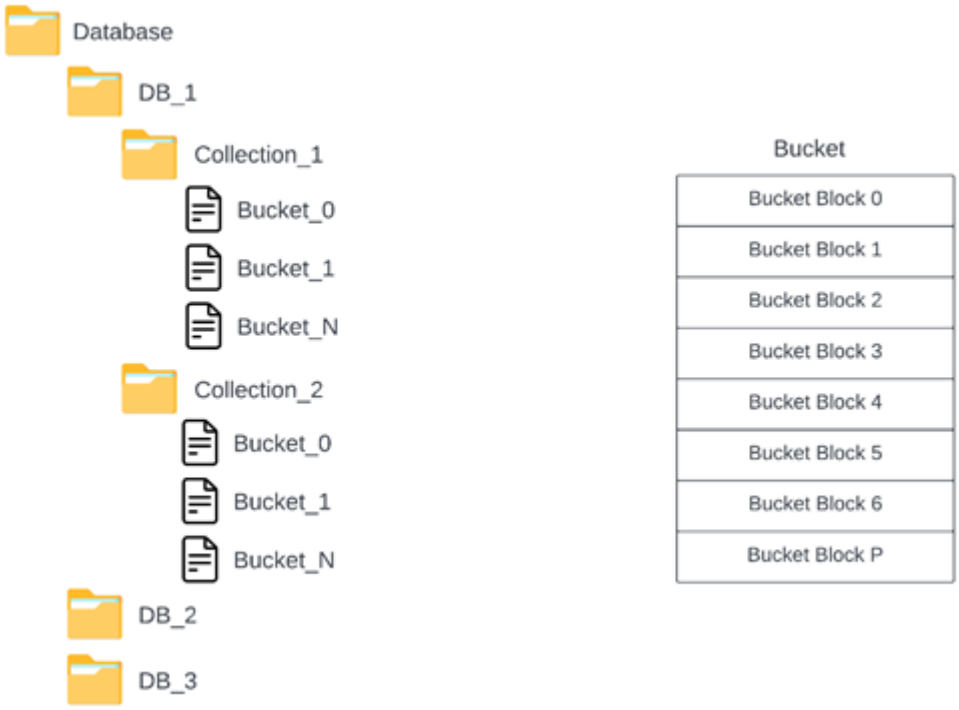
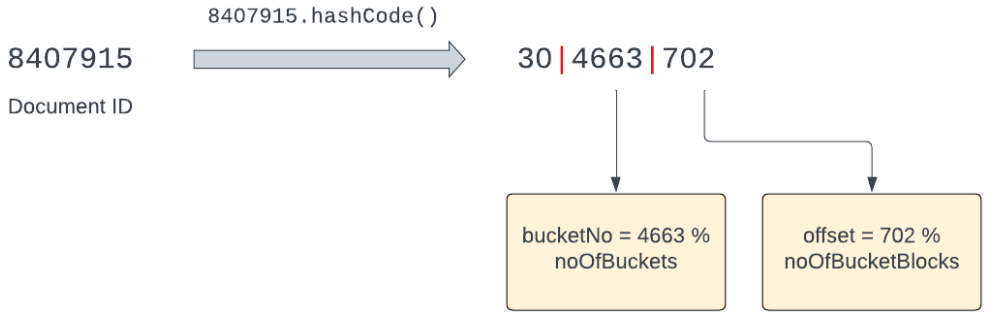


Figure 7: The NoSQL database design - Indexing



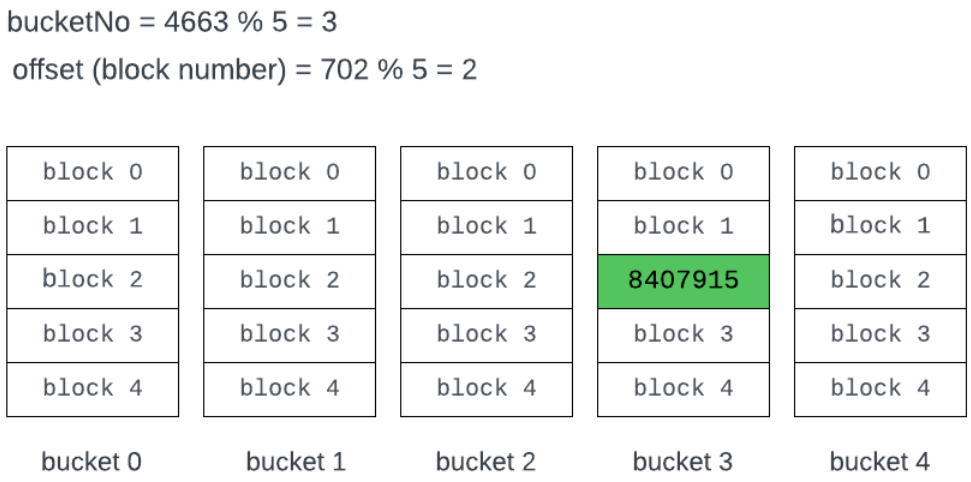


Figure 8: The NoSQL database design – Indexing

Example

In the example depicted above, we have a document with an ID of **8407915** that we intend to store in the database system. The initial step involves utilizing a hashing function to obtain the corresponding hash value, which in this case is **304663702**. Subsequently, we partition the hash code into distinct segments: one segment is used to determine the bucket number, while another segment is used to ascertain the block number. In this specific example, we utilize the indexes of the hash code ranging from 2 to 5 to determine the bucket number by performing a modulo operation with the total number of buckets in the file system. Additionally, the indexes from 6 to 8 are employed to determine the block number within the designated bucket, again through a modulo operation with the number of blocks in a bucket.

In this example, the bucket number is calculated as **bucket 3**, and the resulting offset corresponds to **block 2**. Figure 8 depicts the example detailed followed steps.

3.9 Property Hashing and Indexing

After discussing document indexing, which accelerates the process of reading and writing documents in databases, let's delve into document properties. Document properties within the same document are indexed, greatly enhancing the speed of reading, or updating a single property. But how is this accomplished? In the section discussing document hashing and indexing, I mentioned that documents are stored in bucket blocks, but that's only part of the picture. Bucket blocks are divided into three key components: the document id and property map that associates each property with a unique ID, and property blocks of equal size. When a user wishes to read a property, the node performs the necessary calculations to locate where the document is stored, including the bucket number and block number. Now that it has a pointer at the beginning

of the block, it can access the bytes containing the property map and retrieve the requested property's ID. With the ID in hand, the pointer can then directly navigate to the corresponding property block.

document ID	8765432
property map	{"username"=1, "timestamp"=2, "id"=3}
property_0	"username" : "travel_bug"
property_1	"timestamp" : "2023-09-13 11:10:00"
property_2	"id" : "8765432"
property_3	null
property_4	null

Bucket Block

Figure 9: Property hashing indexing

3.10 Accessing the Database and Performing Operations

After determining the current position for reading or writing within a document, we utilize the **RandomAccessFile** class to access the **bucket.txt file**. This class treats the file as an array of bytes stored in the file system, allowing us to manipulate the file pointer position by specifying a byte offset. By leveraging **RandomAccessFile**, we can seamlessly perform various operations, including writing, reading, deleting, and updating documents. You can find a simple example code for using RandomAccessFile in as below. These operations are executed after we have determined the positions and properties using hashing and indexing techniques.

```
int byteOffset = 700;

RandomAccessFile raf = new RandomAccessFile("bucket.txt", "rw");

raf.seek(byteOffset);
raf.write("Data".getBytes());
raf.close();
```

To facilitate these operations, the data operation services utilize the **AppProperties** class to determine the size of each database entity and calculate their respective start and end positions. Here is an overview of the primary database operations:

3.10.1 Writing a new document

The node is equipped with a **WritingService** that relies on the **HashingService**'s **getWritePosition** method to acquire the positions of the bucket number and block number for writing the document. The **WritingService** initiates the process by writing the document's ID, followed by the creation of a property map that links the document's properties with their corresponding indexes. This information is then saved to the block file. Subsequently, the **WritingService** proceeds to write the document's properties. While transitioning between writing the document ID, mapping properties, and the document itself, each size is obtained from the **AppProperties** class. Here are the calculations employed by the **WritingService** class to manipulate the **RandomAccessFile** pointer for writing.

Document ID Offset = Block Number x Block Size

Property Map Offset = Block Number x Block Size + Document ID Size

Property Offset = Block Number x Block Size + Document ID Size + Property Map Size + Property ID x Property Size

3.10.2 Reading a document

Within the node, a **ReadingService** is available, relying on the **HashingService**'s **getReadPosition** method to retrieve the necessary positions of the bucket number and block number for document retrieval.

The **ReadingService** kicks off the process by first retrieving the document's ID and confirming its match with the requested document. Subsequently, it moves on to retrieve the complete document segment from the bucket block. To accomplish this, the **ReadingService** utilizes a **FileInputStream**, along with a buffer to gather the bytes read from the file, which are then sent back to the client.

Here are the calculations employed by the **ReadingService** class to manipulate the **RandomAccessFile** pointer for reading.

Document ID Offset = Block Number x Block Size

Full Document Offset = Block Number x Block Size + Document ID Size + Property Map Size

3.10.3 Deleting a document

To delete a document, the **DeletingService** locates the start of the block and replaces the entire block with **null** values. This operation is performed using the following straightforward calculation:

Block Offset = Block Number x Block Size

3.10.4 Reading a property

In addition to reading entire documents from the database, the node also offers the capability to read specific properties. This is achieved by following the same process as when reading a document to retrieve the necessary positions. Subsequently, the property map is read, allowing the retrieval of a property's ID based on its name. Once the property's ID is obtained, the desired property is found by adjusting the **RandomAccessFile**'s offset through a straightforward calculation, as demonstrated below.

Document ID Offset = Block Number x Block Size

Property Map Offset = Block Number x Block Size + Document ID Size

Property Offset = Block Number x Block Size + Document ID Size + Property Map Size + Property ID x Property Size

3.10.5 Updating a property

When updating a property, the node follows the same steps as when reading a property, utilizing the **UpdatingService**. However, after calculating the property's offset, the UpdatingService first deletes the property by overwriting its slot with **null** values and then proceeds to write the updated property.

3.10.6 Solving Collisions when Writing Documents

The process of writing documents was previously explained with the assumption that the writing position is always empty. However, situations arise when the writing slot is already occupied. To address this issue, Quadratic probing is employed as a solution to handle collisions, which occur when two or more keys hash to the same position.

Quadratic probing utilizes a formula to determine the next available slot when a collision occurs. Instead of linearly incrementing the index, it follows a quadratic incrementing pattern. Below is a straightforward formula and example demonstrating the quadratic probing pattern.

For an example, in figure 10 we have a document with an ID of 8468432. After applying hashing and indexing through the hash mapping service, we obtain a hash code with a value of 213297457, which corresponds to block 1 within the bucket.

However, block 1 is already occupied by another document. To find the next available slot, the **getWritePosition** method employs quadratic probing. This technique utilizes a simple formula that exponentially increments a variable 'i' starting from 'i = 0' until it locates an empty slot. In this specific case, the method checks block 2 and 5, both of which are occupied, until it eventually reaches block 10, which is empty. Consequently, the document is stored in block 10.

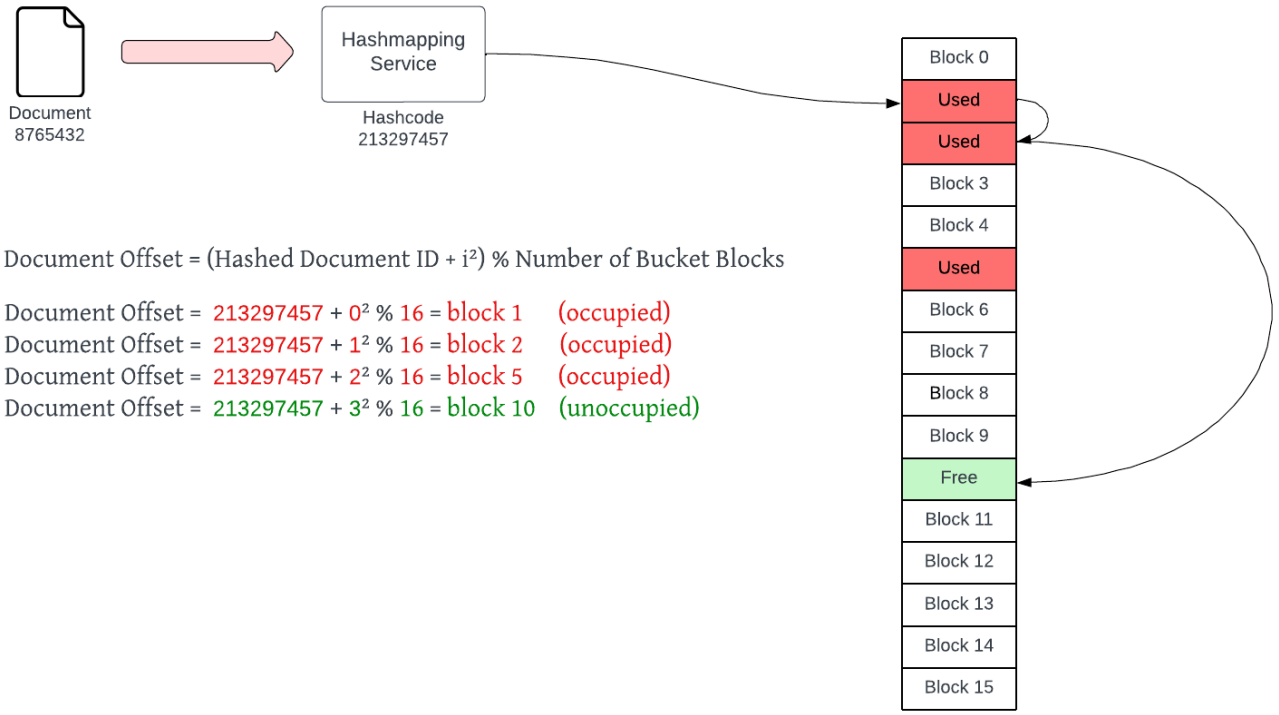


Figure 10: Practical Example

3.9 Node Properties and Configurations

In the project, there are several values crucial for hashing, indexing, and configuring the project structure. These values can be conveniently adjusted in the **application.properties** file. They include:

- **Node's Port Number:** This sets the port number for the node.
- **Number of Affinity Nodes**: Determines the quantity of nodes in the cluster used to distribute write and update operations.
- **Domain Name**: Specifies the domain name of the node
- **Number of Buckets**: Defines the number of buckets in each collection
- **Bucket Block Size**: Specifies the size of each bucket block.
- **Property Map Size**: Sets the size of the property map.
- **Property Size**: Determines the size of properties.
- **Document ID Size**: Specifies the size of document IDs.
- **Cache Capacity**: Sets the capacity of the cache.

Each of these values plays a distinct role in facilitating essential database operations within the project. The code snippet displays the default configuration that can be easily overridden in the environment section of the Docker Compose file, as discussed in the **DevOps Practices** section.

```
# Server Configuration
server.port=8082

# Application Node Configuration
app.nodePortNo=8082
app.nodeDomainName=localhost
app.noOfAffinityNodes=3

# Cache Configuration
app.cacheCapacity=100

# Database and Storage Configuration
app.noOfBuckets=100
app.noOfBucketBlocks=100
app.bucketBlockSize=530

# Property and Document Configuration
app.propertyMapSize=200
app.propertySize=30
app.documentIdSize=30
```

Code Snippet: Default Node Configuration

IV. DATA STRUCTURES USED

Data structures are fundamental building blocks used to organize and store data efficiently within computer programs. They define the way data is arranged and accessed, impacting the speed and efficiency of operations performed on that data. Different data structures are suited to different tasks, and selecting the appropriate data structure is crucial for optimizing program performance.

In the following section, we will provide a list of the specific data structures employed in our system. Each data structure plays a distinct role in enhancing data management and processing within the application:

4.1 HashMap and Deque for cache implementation

The cache employs the LRU (Least Recently Used) replacement algorithm, which swaps out the least recently accessed element upon reaching its capacity. It utilizes two primary data structures: a **HashMap** and a **Deque**. The HashMap serves as an efficient cache for the nodes, as it resides in the JVM heap memory, a runtime data area where Java applications store and access objects. This design choice enhances performance compared to searching for and accessing documents on disk. Below, you will find a brief overview of the key data structures used in the caching system:

1. HashMap

The HashMap stores key-value pairs where keys represent document IDs, and values represent the corresponding documents, as illustrated in Figure 11. This significantly speeds up document retrieval by ID compared to reading from a disk.

Key (ID)	Value (Document)
9876543	{"id": 9876543, "username": "hiking_guru", "timestamp": "2023-09-13 08:15:00"}
8765432	{"id": 8765432, "username": "travel_bug", "timestamp": "2023-09-13 11:10:00"}
4567890	{"id": 4567890, "username": "tech_geek", "timestamp": "2023-09-13 14:55:00"}
7890123	{"id": 7890123, "username": "nature_lover", "timestamp": "2023-09-13 16:18:00"}
1234567	{"id": 1234567, "username": "gaming_addict", "timestamp": "2023-09-13 17:09:00"}

Figure 11: HashMap stores key-value pairs

2. Deque

The deque holds the keys (document IDs), and the order of the elements in the deque represents the recency of use. The **front** of the deque represents the most recently used document, while the **rear** represents the least recently used document. For a visual illustration, please refer to Figure 12.

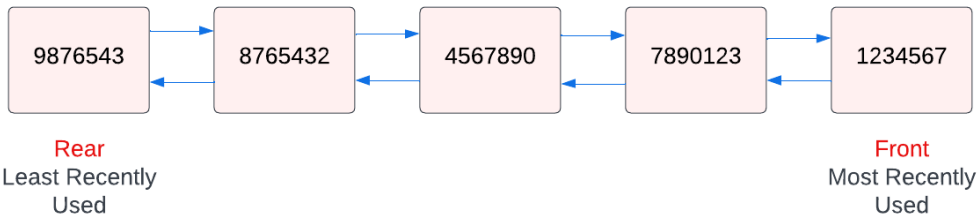


Figure 12: Deque Example

The caching system relies on the LRU (Least Recently Used) replacement algorithm to efficiently manage the cache's capacity by swapping out the least recently accessed element. Let's now delve into the details of this algorithm.

There are 3 different scenarios where nodes access the cache:

1) **Client Requests to Read a Document**
   In this scenario, the node first checks if the document is present in the cache's HashMap using the document ID. If it is found, the node retrieves the document and provides it to the client. The document's ID is then moved to the front of the deque. If the document is not available in the cache, the node reads the document from the disk, adds it to the HashMap and deque, and subsequently removes the least recently used document from the deque to make room for the new entry.

2) **Client Requests to Write / Update a Document**
   In this scenario, the node initially checks if the document is present in the cache's HashMap using the document ID. If it is found, the node updates the cache by removing the old document from both the HashMap and the deque. Then, it writes the new document to the HashMap and places its ID at the front of the deque. If the document does not exist in the cache, the node removes the least recently used element from the deque and deletes it from the HashMap. Subsequently, it adds the new document to the HashMap and inserts its ID at the front of the deque.

3) **Client Requests to Delete a Document**

   In this scenario, the node checks if the document is present in the cache's HashMap using the document ID. If it is found, the node removes the document from both the HashMap and the cache.

V. MULTITHREADING AND LOCKS

5.1 Multithreading in the node applications

In a Spring application, multithreading is often employed to execute methods asynchronously using the @Async annotation. However, it's generally discouraged to use the default **SimpleAsyncTaskExecutor** configuration in production environments due to several reasons. Firstly, the **SimpleAsyncTaskExecutor** creates a new thread for each new request without reusing them, and secondly, it consumes a substantial amount of memory (up to 1 MB) and processing power. This can potentially lead to application performance degradation or even application crashes when many threads are created.

To address these concerns, a custom **TaskExecutor** configuration has been implemented for our Spring application node. This custom configuration is specifically designed to efficiently handle large-scale operations.

Within the task executor bean the **ThreadPoolExecutor** is configured, and the following parameters are set, as depicted in the code as below:

- **Core pool size**: This specifies the number of threads created for each new request to execute tasks.

- **Queue capacity**: When the number of new threads exceeds the pool size specified, new threads are added to a queue. The queue capacity specifies the maximum number of threads to be added to the queue.
- **Max pool size**: If the queue is full and the number of threads is less than the size defined in the max pool size, a new thread is created. However, if the number of threads exceeds the size defined in the max pool size, the task is rejected.

```
spring.task.execution.pool.core-size=10
spring.task.execution.pool.max-size=20
spring.task.execution.pool.queue-capacity=50
```

5.2 Locks used in the Database System

Locking is a mechanism that allows a thread to exclusively acquire a lock on an object or a class, preventing other threads from accessing the locked object or class until the lock is released, there are three critical situations where locks were needed in the database system. These situations are:

1. **Two instances of the same spring application want to write on the same document's property**

   This scenario is effectively managed through the implementation of a **LockingService** within the Node application. The **LockingService** establishes a concurrent HashMap, associating a combination of the document ID and the property's name with a lock object. Consequently, each time a new property update is requested, it acquires a lock. If another request seeks to update a property that is already locked, the property modification operation will be queued and wait until the lock becomes available. Figure demonstrate the locking service.

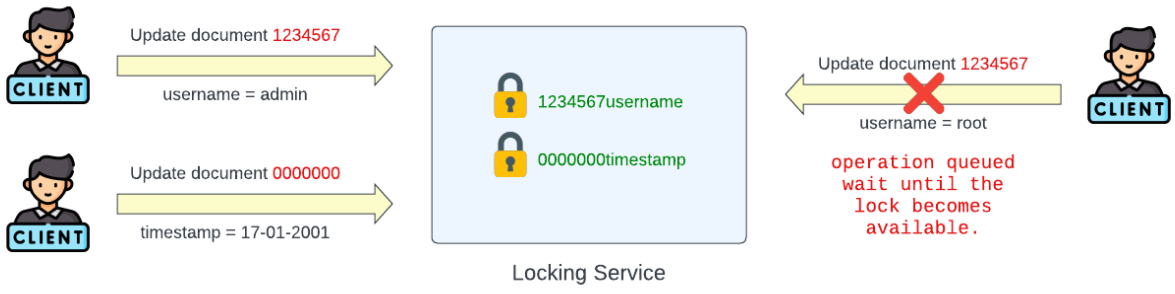


Figure 13: Solving Race Conditions Using Locks

2. **A node sends an update request to the write affinity node**

   When node-a determines the node ID (node-b) to which it will send the update request, it is essential for node-a to possess the most recent data before the affinity node can accept the request. To ensure this, optimistic locking is employed. Node-a includes its current data knowledge with the write update request. Node-b then compares node-a's knowledge with its current data. If they match, the request proceeds, and node-b updates the document. However, if there's a mismatch, node-b responds with a rejection message, prompting node-a to wait and resend the request. Node-a might have outdated data because it may not have received a broadcast request yet. Figure 14 illustrates the node concept.

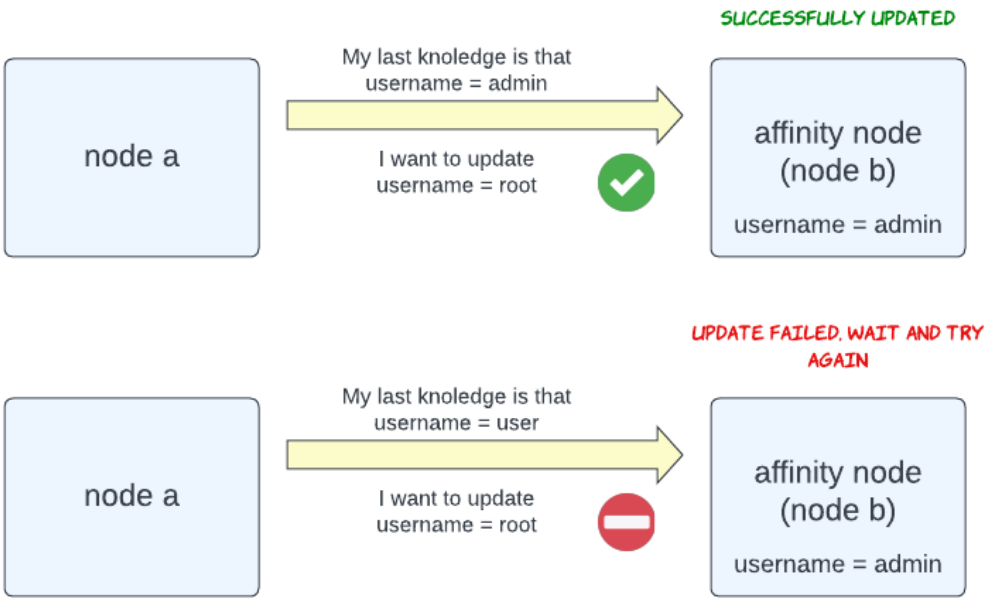


Figure 14: Optimistic Locking

3. **Adding, updating, and removing from the cache**

   Accessing the cache concurrently with multiple threads can result in race conditions. One suggested approach is to replace HashMap and Deque with thread-safe concurrent versions. While this solution can help when accessing each data structure independently, in a caching algorithm, operations often require simultaneous access and modification of both the HashMap and the deque. This can still introduce race conditions, even when using concurrent data structures.

VI. DATA CONSISTENCY ISSUES IN DB

In the project, it is crucial to ensure that data, schemas, and indexes are replicated across all nodes to maintain data consistency. Examples of data consistency issues that may arise include situations where two users assigned to the same node attempt to update the same property within the same document. Another scenario occurs when a node redirects an update request to an affinity node; data consistency issues can arise if the redirecting node's data is not up to date and still attempts to execute the request. Both data consistency challenges are effectively addressed through the implementation of locking mechanisms. For a comprehensive solution, please refer to Section **Multithreading and Locks**.

VII. NODE HASHING AND LOAD BALANCING

Hashing is a technique used in computer science and distributed systems to efficiently map data to specific locations or resources. It involves applying a hash function to a piece of data to generate a fixed-size value (hash code), which is used to index or locate the data. Hashing is widely employed for tasks such as data retrieval, indexing, and ensuring data consistency in distributed systems.

Load balancing, on the other hand, is a strategy used to distribute incoming network traffic or computational workloads evenly across multiple resources or servers. This approach ensures that no single resource is overwhelmed, maximizing system performance, fault tolerance, and scalability.

In the following section, we will list the specific hashing and load balancing techniques implemented in our system. These techniques are vital for optimizing data distribution, system reliability, and resource utilization:

7.1 Load balancing assigning nodes to users

The bootstrapping node is responsible for load balancing the users across the nodes in the cluster. This can be achieved in various ways, such as **round robin**, **sticky round robin**, **hashing**, or **least connections**.

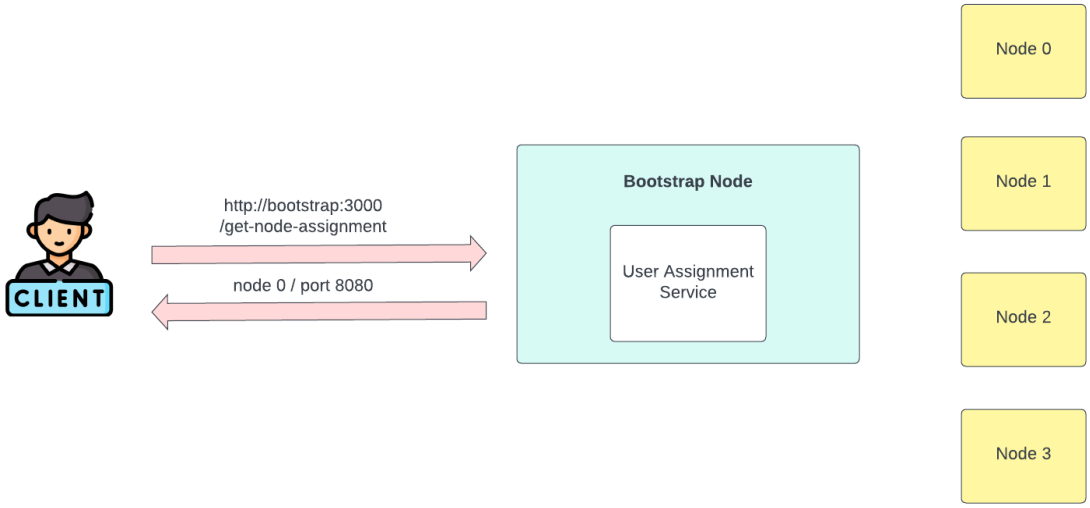
The **least connections** approach seemed to be the best option, but centralization was not permissible in the system. To implement the least connection load balancing, the bootstrap node would need to communicate with the nodes to inquire about the number of connected users. Based on the responses received, the bootstrap node would decide which node is best to send traffic to. However, due to its centralized nature, this approach was not allowed.

Hashing is considered a good option for load balancing, particularly if the requirements included maintaining session persistence and consistency. However, let's envision a scenario in the project: If all users on a single node decide to log out simultaneously, this would result in that node not being assigned to any user unless a new

user arrives, and their hash code happens to be mapped to that specific node. Similarly, if one node becomes congested while other nodes have no logged-in users, the overloaded node will continue to receive requests if the hash code is mapped to it.

For simplicity, the round-robin approach was implemented. In this approach, the bootstrap iterates through the list of nodes in the cluster and assigns each new user to the nodes sequentially. Once the list is exhausted, it returns to the beginning and repeats the process.

To assign a node to the client, the bootstrap node responds with the domain name and port number of the assigned node. Figure 15 illustrates how users request to be assigned to a node from the bootstrap node and provides a table showing user node assignments using the round-robin approach.



User	Domain Name	Port No
User 1	node0	8080
User 2	node1	8081
User 3	node2	8082
User 4	node3	8083
User 5	node0	8080
User 6	node1	8081
User 7	node2	8082
User 8	node3	8083

Figure 15: Node concept example

7.2 Load balancing writing operation to affinity nodes

In this project, writing operations are the most time-consuming. To enhance performance, we distribute write operations across multiple nodes. When a client requests to write a document, the **GenerateIDService** is responsible for generating a ID. This ID is then processed through the **HashMapping** service, which employs a hashing algorithm. The resulting hash value is used in conjunction with modulo arithmetic, considering the number of nodes in the cluster (excluding the bootstrapping node). This assignment designates a specific node to handle the document writing process. Once written, the node broadcasts the completion of the writing operation. This process ensures that the load of writing operations is balanced among the nodes.

Here's a simple example of document with an ID 2347891 requested to be written and gets assigned to affinity node 0, which is shown in Figure 16.

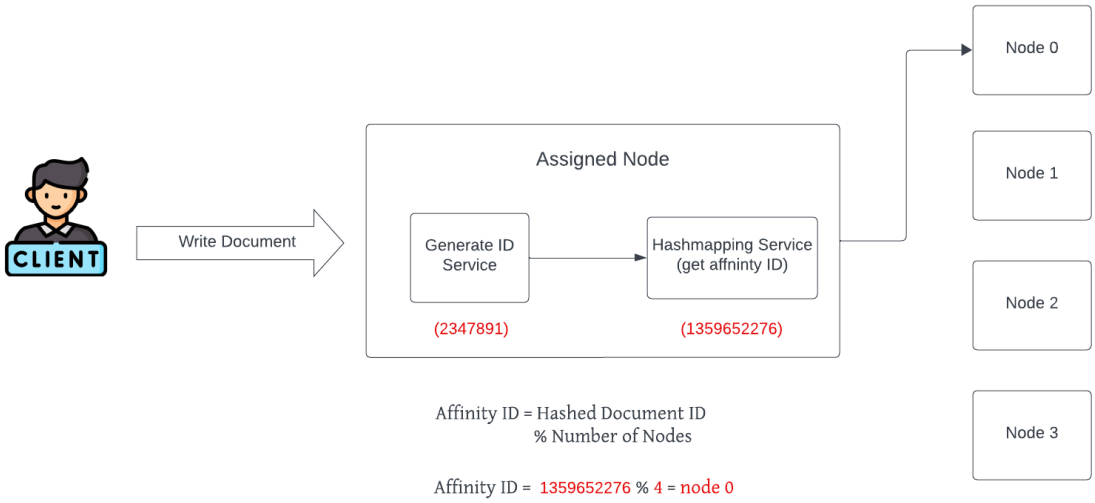


Figure 16: Load balancing example

VIII. COMMUNICATION PROTOCOLS BETWEEN NODES

All database operations are carried out through HTTP requests. In this setup, the node exposes API endpoints that clients can use to send GET and POST requests. This approach is both easy and effective. For instance, operations like reading documents, which do not alter the server's structure, are performed using GET requests, while actions like writing, updating, and deleting are accomplished using POST requests.

The client initiates the process by sending a GET request to the bootstrap node, prompting it to respond with a node's domain name and port number. This information is then stored in the **NodeAdapter** class within the client project. Subsequently, any database operations requested by the client are directed to the designated node, which is stored in the **NodeAdapter**.

Here's a table that shows the API endpoints in both the bootstrap node and nodes, and the controllers responsible for every request.

API Endpoints	Controller	Method	Operation
`http://domain:port/get-node-assignment`	Node Assignment Controller	GET	Return a node's domain name and port number to the client.
`http://domain:port/start-cluster`	Startup Controller	GET	Start the cluster by broadcasting the user list and cluster node information.
`http://domain:port/create-user`	User Controller	POST	Create a new user and broadcast it.
`http://domain:port/rest/document/update-affinity/`	Affinity Controller	POST	Find the affinity node responsible for updating a property and perform a POST request to the node responsible for updating the property.
`http://domain:port/rest/document/write-affinity/`	Affinity Controller	POST	Generate an ID and find the affinity node responsible for writing the document, then send a POST request to it.

`http://domain:port/send-broadcast/{operation}/{data}`
 	
`http://domain:port/send-broadcast/{operation}/{id}/{data}`
	Broadcast Controller	POST	Broadcast database operations across the cluster. Once a node writes, reads, updates, or deletes a document, it requests this API and broadcasts the operation to all the nodes in the cluster.

`http://domain:port/initialize-node`	Initialization Controller	POST	Accept a cluster list with all the nodes in the cluster and their information, such as domain name and port numbers.
`http://domain:port/authentication`	Login Controller	POST	Used to validate the user's credentials against the usernames and passwords in the user's file.
`http://domain:port/rest/document/read/{id}`	DB Operation Controller	GET	Read and return a stored document by ID.
`http://domain:port/rest/document/read/{id}/{property}`	DB Operation Controller	GET	Read and return a stored document's property by ID and the property's name.
`http://domain:port/rest/document/write-bcast/{id}`	DB Operation Controller	POST	Write a document after appending an ID to it. Note that this API endpoint broadcasts the write operation across the nodes in the cluster.
`http://domain:port/rest/document/write/{id}`	DB Operation Controller	POST	Write a document after appending an ID to it.
`http://domain:port/rest/document/update-bcast/{id}/{property}`	DB Operation Controller	POST	Update a document's property by the ID of the document and the new property value. Note that this API endpoint broadcasts the update operation across the nodes in the cluster.
`http://domain:port/rest/document/update/{id}/{property}`	DB Operation Controller	POST	Update a document's property by the ID of the document and the new property value.
`http://domain:port/rest/document/delete-bcast/{id}`	DB Operation Controller	POST	Delete a document by ID. Note that this API endpoint broadcasts the delete operation across the nodes in the cluster.
`http://domain:port/rest/document/delete/{id}`	DB Operation Controller	POST	Delete a document by ID.
`http://domain:port/create-collection/{collectionName}`	Schema Controller	POST	Create a new database collection with a given name.

`http://domain:port/delete-collection/{collectionName}`	Schema Controller	POST	Delete a database collection with a given name.
`http://domain:port/use-collection/{collectionName}`	Schema Controller	POST	Navigate to a specific collection and use it.
`http://domain:port/create-database/{databaseName}`	Schema Controller	POST	Create a new database with a given name.
`http://domain:port/delete-collection/{collectionName}`	Schema Controller	POST	Delete a database with a given name.
`http://domain:port/use-collection/{collectionName}`	Schema Controller	POST	Navigate to a specific database and use it.

IX. SECURITY ISSUES

9.1 Robust User Authentication: Utilizing Salted Hashing for Enhanced Security

Passwords must be securely stored to prevent the retrieval of the original password. One-way cryptographic hash functions are employed for this purpose, as passwords cannot be deduced from the hash alone.

Numerous hashing functions exist, yet not all are suitable for password hashing due to vulnerabilities to brute-force attacks. After thorough research, PBKDF2 and bcrypt emerged as excellent hashing algorithms, with the added advantage of being among the NIST-recommended algorithms. Consequently, the system has implemented PBKDF2 bcrypt in the **AuthenticationService** class.

When a user is created, a password is specified. This password is then hashed using the **getEncryptedPassword** method and stored in the user.csv file. However, a challenge arises when users have identical passwords, resulting in the same hash codes. Exploiting this weakness, hackers can launch rainbow table attacks. To counter this threat, we employ a salt — a random string— appended to each password before hashing. This ensures unique hashes even for identical passwords, effectively mitigating the risk of rainbow table attacks.

For user authentication, once a user provides their username and password, the username is searched for in the database. If the user is located, their unique salt is retrieved, and the password is hashed using this salt. We subsequently compare the resulting hashed password with the stored hashed password in the database. If the two match, the user is successfully authenticated and can perform database operations. If they do not match, the user is not authenticated. For a visual representation of this process, please refer to Figure 17.

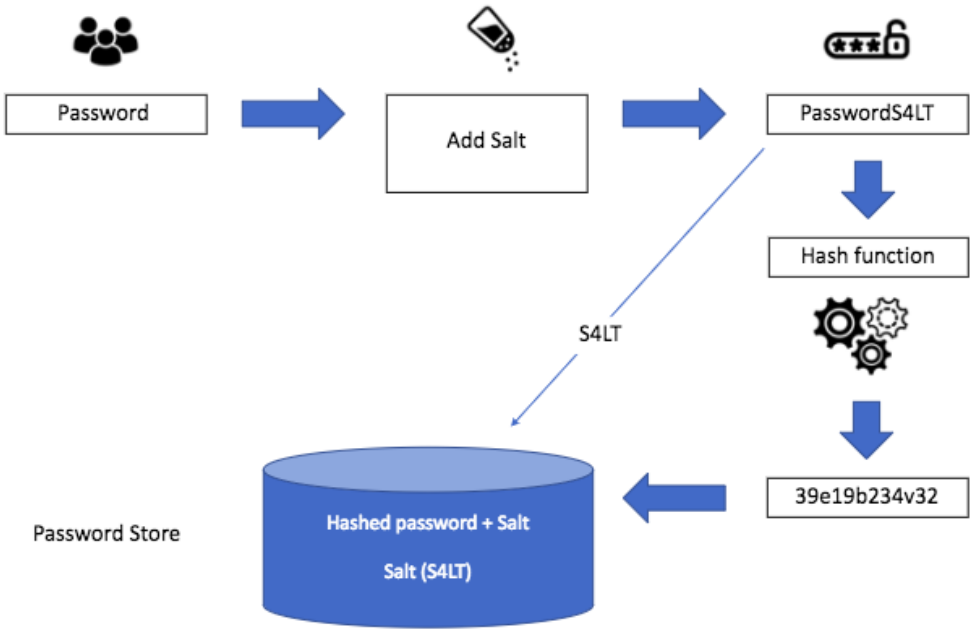


Figure 17: Authentication Process

9.2 User Roles and Access Control in the System

Users with varying roles can access different APIs. After successfully passing the authentication test, users are categorized as either administrators or standard users. Standard users possess the capability to perform a wide range of operations, including document and property management, such as writing, reading, updating, and deleting. In contrast, administrators enjoy an additional privilege, which allows them to create and delete databases.

X. CODE TESTING

Each component of the system has undergone independent testing to ensure its individual functionality works effectively. Additionally, after the system integration, each component has been subjected to black-box testing, known as functional testing. This testing approach assesses the software's functionality and behavior from an external perspective, without taking the internal code or structure into consideration. Below, you can find details on how various parts of the project were tested:

10.1 Database Operations Testing using Postman

Postman is an API development tool used for building, testing, and modifying APIs. It is particularly valuable for backend testing, where an endpoint URL is entered. Postman sends the request to the server and then receives the response from the server. All API endpoints have been independently tested using Postman. After sending database requests to the server, the database file system has been thoroughly examined to ensure

that all calculations have been performed correctly. Furthermore, document writing, deletion, and updating have been verified to occur in the correct positions, confirming the successful execution of these operations.

10.2 Collision Resolved Testing

Quadratic probing has been independently tested by configuring a single node with a single bucket and sending numerous write requests to the database. The document offset writing positions have been examined and successfully resolved collisions, confirming the accuracy of calculations and the behavior of the buckets themselves.

10.3 Testing Writing to the Same Property Lock

When multiple instances of the Spring application attempt to write to the same document property, a race condition can occur. This issue is resolved by implementing locks, as explained in the **multithreading and locks** section. However, testing this solution presents a challenge. Attempting to write to the same property at the exact same time is difficult to replicate. Even when using Postman, it's possible to execute multiple requests simultaneously, but this doesn't guarantee the occurrence of a race condition since everything happens so quickly.

The locks were tested by intentionally having the thread holding the lock enter a sleep state for a designated period. During this time, another request to modify the same property was sent. The test proved successful, as the new thread patiently waited for the lock to be released before proceeding to write or modify the property.

10.3 Testing Optimistic Locking

Optimistic Locking, as described in the multithreading and locks section, serves to maintain data consistency across cluster nodes. To test optimistic locking, we manually altered a property of a document in an affinity node, effectively placing the optimistic lock in a 'waiting' state. This was done because the property known to the requesting node differed from the property in the affinity node. The test successfully demonstrated that the requesting node remained in a waiting state until a broadcast containing the property modification reached it.

To enhance visibility during the testing process, the broadcast was intentionally slowed down.

10.4 Testing the Distribution of Write Operation across affinity nodes

The process of writing documents in a distributed system is spread across multiple nodes in the cluster. This distribution is necessary because writing documents is a resource-intensive operation that can exert significant pressure on the server. When a node receives a write document request, it redistributes these requests across the nodes in the cluster to ensure even distribution, as explained in the sections on **node hashing and load balancing**. However, how can we test and verify that this distribution is happening effectively?

To address this concern, a simple script has been developed to test the even distribution of write operations among the cluster nodes. The script executes 1000 iterations, making GET requests to a random data API random-data-api.com/api/v2/credit_cards that responds with random documents. During each iteration, the script initiates a write operation directed to a specific node, using the retrieved document from the random data API. After each write request, the assigned node records the affinity node responsible for the document's write operation in an Array List. Additionally, a HashMap is utilized to keep track of the frequency of each affinity node in the write operations.

Figure 18 illustrates the even distribution of write operations among the cluster nodes, confirming effective load balancing across the cluster.

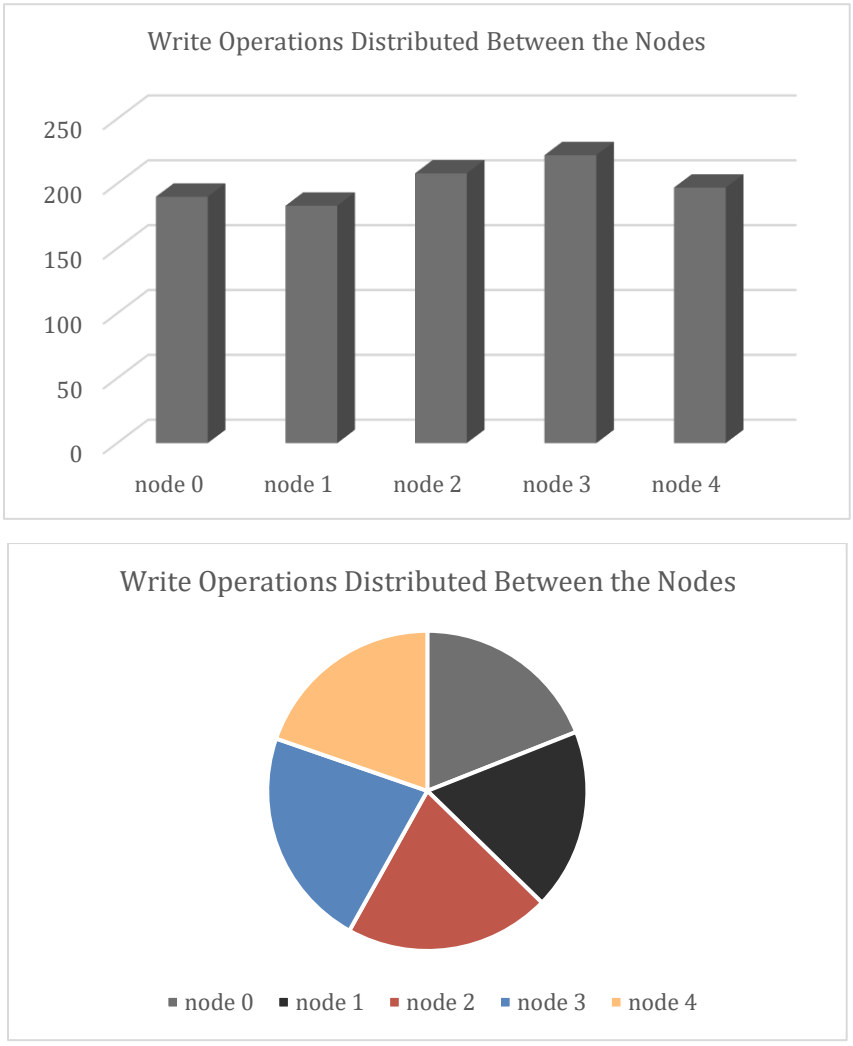


Figure 18: Load Balancing Write Operations in the Cluster

XI. CLEAN CODE PRINCIPLES

Clean code principles make sure that you have a well-structured, readable, and maintainable code that is easy to understand and modify. It focuses on making code more understandable, reducing complexity, and minimizing potential for errors and bugs.

The provided code adheres to the clean code principles outlined by best practices. Here are a few examples:

Naming

- Class names such as **CommandInterpreter**, **ClusterNode**, **LoadBalancingService**, **AuthenticationService**, and **BroadcastController** are clear and indicative of their purpose.
- Variable and method names, like **startCluster** (), **getEncryptedPassword** (), and **getUserList** () are descriptive and convey their intentions.

11.1 Functions/Methods

The code is divided into methods, improving readability and maintainability. Functions and methods have clear and focused purposes, such as **startCluster ()**, responsible for starting the cluster, and **getEncryptedPassword** () responsible for encrypting a given password.

11.2 Comments

Well-placed comments effectively explain the purpose, behavior, and complex logic of certain components and algorithms.

11.3 Formatting

- The code follows consistent formatting conventions and maintains a clear and readable structure.
- Indentation and spacing are used consistently throughout the codebase, enhancing readability and maintainability.

11.4 Class Organization

- The classes in the codebase are organized and grouped based on their responsibilities.
- The Single Responsibility Principle and object-oriented design principles guide the class organization.

11.5 Error Handling

The code implements proper error handling mechanisms, such as try-catch blocks, and provides meaningful error messages and fallback mechanisms.

XII. EFFECTIVE JAVA

11.1 Interface types to represent abstractions

The code makes good use of interface types (**DatabaseCommand**) to represent different behaviors and abstractions. This promotes modularity and allows for multiple implementations to fulfill those contracts. It enables loose coupling and provides flexibility to extend the code with new behavior implementations in the future.

11.2 Composition over inheritance

The code demonstrates composition in different places. For example, the **StartClusterCommand** and **CreateDatabaseCommand** classes could potentially be composed of a common **DatabaseCommand** implementation instead of directly extending different abstract classes (**CommandWithParams**, **CommandNoParams**). This design choice promotes flexibility, as it allows for easier modification and extension of behavior without being constrained by a rigid class hierarchy.

11.3 Always override the toString method

Providing a well-implemented toString method significantly enhances the readability of classes. This practice can be observed in various parts of the code, such as the overridden toString method in the **CachingService** class, where attempting to print the cache will display its content. Similar instances can be found throughout the code.

11.4 In public classes, use accessor methods, not public fields

Public access should be avoided for degenerate classes, and instead, they should be replaced with accessor methods (getters) and mutators (setters). This practice is demonstrated throughout the code. For instance, in the **AppProperties** class, you can access and modify various properties using getters and setters.

11.5 Use interfaces to only define types

When a class implements an interface, the interface serves as a type that can be used to refer to instances of the class. This is clearly demonstrated by the DatabaseCommand interface, which defines the type for classes that implement it.

11.6 Prefer class hierarchies to tagged classes

A tagged class is a design pattern in which a single class represents multiple types of objects or data structures. This is achieved by including a "tag" that indicates whether the execute method of a command takes parameters. However, in this project, tagged classes were avoided due to their drawbacks. Tagged classes can be verbose, error-prone, and

ineffective, often involving excessive boilerplate code, reducing code readability, increasing memory usage, and having other shortcomings. Instead, a chose a class hierarchy approach has been chosen. In this hierarchy, commands that require parameters extend an abstract class named **CommandWithParameter**, while commands that do not require parameters extend another abstract class called **CommandNoParameter**.

11.7 Prefer lists to arrays

In the project, a list is utilized instead of an array, which is the preferred choice. Specifically, a list of cluster nodes is employed within the **CommunicationService** of a node. This facilitates the nodes in acquiring knowledge about the other nodes within the cluster and enables seamless communication among them.

11.8 Consistently use the Override method

Apply the **@Override** annotation to each method declaration that overrides a superclass declaration. This practice is demonstrated in command classes that extend **CommandWithParams** and **CommandNoParams**, where the **execute** method is overridden.

11.9 Prefer Foreach loops to traditional for loops

For-each loops are extensively employed throughout the project. One instance is seen when nodes need to broadcast; they use a foreach loop to iterate through the nodes within the cluster. Another example can be found in the **isValidCommand** method, where it iterates through the **validCommands** HashMap to validate user-entered commands.

11.10 Know and use libraries

Utilizing libraries provides access to the expertise of their authors and the collective experience of those who have used them previously. This approach eliminates the need to invest valuable time in crafting solutions for problems that are only indirectly related to your core work. Furthermore, libraries tend to enhance their performance over time, resulting in code that is more easily readable, maintainable, and reusable. In this project, various libraries have been leveraged instead of creating custom implementations. For instance, pre-made data structures such as HashMap, deque, and linked lists from the **java.util package** have been used. Additionally, the **Jackson library** has been employed for JSON processing, thus eliminating the need to create a custom parser.

11.11 Beware the performance of string concatenation

Repeatedly using the string concatenation operator to concatenate n strings requires quadratic time in n. That's why StringBuilder is employed to achieve acceptable performance. An example of this can be found when fetching the response of a POST request in the **Requester** class.

XIII. SOLID PRINCIPLES

The SOLID principles are a set of five design principles in object-oriented programming that aim to create more understandable, flexible, and maintainable software. These principles were introduced by Robert C. Martin and have become fundamental guidelines for designing clean and efficient code. Each letter in the word "SOLID" represents one of these principles:

13.1 Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should have only one reason to change. In other words, a class should have a single responsibility or purpose and should be focused on performing that responsibility.

The proposed code adheres to the SRP for example:

- The **DBOperationController** class is responsible for handling database operation API requests, such as writing, reading, updating, and deleting.
- The **FileManagerController** class is responsible for executing file operations, including creating and deleting files and directories, checking for directory existence, and listing directories within a directory.
- The **Requester** class is responsible for executing API requests, including GET and POST requests.
- The **AuthenticationService** class is responsible for authenticating users by matching a username and password with a database.
- The **LockingService** class is employed to acquire a lock before updating a document property. This ensures that only one thread at a time can modify a property."

13.2 Open/Closed Principle (OCP)

The code demonstrates how various types of commands can be effortlessly added without requiring modifications to the existing codebase. This is achieved through the utilization of inheritance and polymorphism techniques. New valid commands can be created by extending either the **CommandWithParams** or **CommandNoParams**

abstract classes and subsequently overriding the **execute** method for each specific command. These abstract classes implement the **DatabaseCommand** interface.

13.3 Liskov Substituation Principle (LSP)

The Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of its subclasses without disrupting the application's functionality. This principle becomes evident when substituting various database commands that extend CommandWithParams, resulting in a seamless transition with no issues.

13.4 Interface Segregation Principle (ISP)

The Interface Segregation Principle states that a class should not be compelled to implement interfaces it doesn't utilize. In this project, there are no instances where this principle is violated.

13.5 Dependency Inversion Principle

The Dependency Inversion Principle dictates that high-level modules should not rely on low-level modules; instead, both should rely on abstractions. This is shown in the **CommandInterpreter** class, which avoids direct dependencies on low-level modules like **DeleteDatabaseCommand** and **CreateDatabaseCommand**, choosing to depend on the **DatabaseCommand** abstraction instead.

XIV. DESIGN PATTERNS

Design patterns are established solutions to recurring design problems in software development. They serve as templates or blueprints that help developers create software that is more maintainable, scalable, and extensible. These patterns encapsulate best practices and decades of collective industry knowledge, making them invaluable tools for designing robust and efficient systems. By applying design patterns, developers can address common challenges in a structured and proven way, resulting in more efficient development and improved code quality.

In the following section, the design patterns applied in the system are listed for a clearer understanding of how they contribute to its overall design and functionality:

The Command Design Pattern for executing commands

The Command pattern is a behavioral design pattern that encapsulates a request as an object, known as a command, which is then passed to an executor object for execution. This design pattern is found in the **CommandLine** project, specifically within the **executeCommand** method of the **CommandInterpreter** class. In this context, the **executeCommand** invokes the **execute** method defined within each command object.

These command objects extend the **CommandWithParams** or **CommandNoParams** class, allowing for the execution of various database commands in a flexible and extensible manner.

Please refer to figure 19 for a better understanding of the used objected oriented design and design pattern.

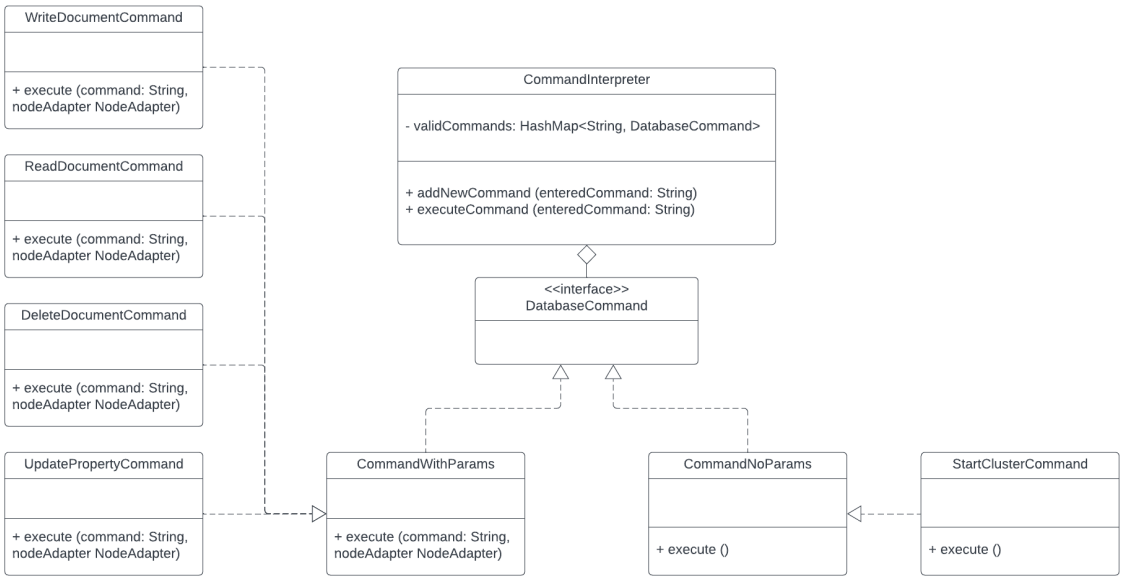


Figure 19: The Command design pattern

XV. DEVOPS PRACTICES

DevOps (Development and Operations) is a set of practices, principles, and cultural philosophies that aim to improve collaboration and communication between development teams and IT operations teams. The goal is to deliver software more rapidly, reliably, and efficiently. Here are some DevOps practices implemented in the project:

15.1 Using Version Control

The Git version control system was employed in the project to manage and track changes in code and configuration files, thereby preserving a history of modifications and simplifying the process of reverting to previous versions when required. Additionally, GitHub, a version control platform, played a significant role in overseeing and documenting these changes within the project while ensuring the repository remained private to prevent access by other individuals working on the same project.

15.2 Containerization and Orchestration

In the project, Docker was utilized as an open platform for developing, shipping, and running applications. This technology allowed the separation of applications from infrastructure, facilitating rapid software delivery.

15.2.1 Node Dockerfiles

All the system's **Dockerfiles** for different nodes share a similar structure. Here are the commands used in these Dockerfiles:

From openjdk:20: This line specifies the base image for the services, utilizing the OpenJDK version 20 image.

COPY ./target/ClusterNode-0.0.1-SNAPSHOT.jar ClusterNode-0.0.1-SNAPSHOT.jar: This command facilitates the transfer of the JAR file from the local file system into the root directory of the Docker container.

CMD ["java", "-jar", " ClusterNode -0.0.1-SNAPSHOT.jar"]: This launches the spring boot application contained in the JAR file.

15.2.1 The Docker Compose File

The **docker-compose.yml** file serves to define and distribute multi-container applications. It facilitates the execution of images and establishment of networks, enabling seamless communication between nodes. Figure 19 summarizes the docker-compose configuration for the mentioned system.

```
version: "3.8"
services:

  bootstrap-node:
    image: bootstrap-node
    ports:
      - "3000:3000"
    networks:
      - cluster-network
    environment:
      - server.port=3000
      - app.nodeDomainNames= cluster-node-0, cluster-node-1, cluster-node-2
      - app.nodePortNumbers= 8080, 8081, 8082
```

```
  cluster-node-0:
    image: cluster-node-0
    ports:
      - "8080:8080"
    networks:
      - cluster-network
    environment:
      - spring.task.execution.pool.core-size=10
      - spring.task.execution.pool.max-size=20
      - spring.task.execution.pool.queue-capacity=50
      - server.port=8080
      - app.nodeDomainName=cluster-node-0
      - app.nodePortNo=8080
      - app.noOfAffinityNodes=5
      - app.noOfBuckets = 100
      - app.noOfBucketBlocks = 100
      - app.cacheCapacity = 100
      - app.bucketBlockSize = 530
      - app.propertyMapSize = 200
      - app.propertySize = 30
      - app.documentIdSize = 30
    depends_on:
      - bootstrap-node


cluster-node-1:
    image: cluster-node-1
    ports:
      - "8081:8081"
    networks:
      - cluster-network
    environment:
      - spring.task.execution.pool.core-size=10
      - spring.task.execution.pool.max-size=20
      - spring.task.execution.pool.queue-capacity=50
      - server.port=8081
      - app.nodeDomainName=cluster-node-1
      - app.nodePortNo=8081
      - app.noOfAffinityNodes=5
      - app.noOfBuckets = 100
      - app.noOfBucketBlocks = 100
      - app.cacheCapacity = 100
      - app.bucketBlockSize = 530
      - app.propertyMapSize = 200
      - app.propertySize = 30
      - app.documentIdSize = 30
    depends_on:
      - bootstrap-node
```

```
cluster-node-2:
    image: cluster-node-2
    ports:
      - "8082:8082"
    networks:
      - cluster-network
    environment:
      - spring.task.execution.pool.core-size=10
      - spring.task.execution.pool.max-size=20
      - spring.task.execution.pool.queue-capacity=50
      - server.port=8082
      - app.nodeDomainName=cluster-node-2
      - app.nodePortNo=8082
      - app.noOfAffinityNodes=5
      - app.noOfBuckets = 100
      - app.noOfBucketBlocks = 100
      - app.cacheCapacity = 100
      - app.bucketBlockSize = 530
      - app.propertyMapSize = 200
      - app.propertySize = 30
      - app.documentIdSize = 30
    depends_on:
      - bootstrap-node

networks:
  cluster-network:
```

Table 2: The Docker Compose File

In the project, each node is represented as a virtual machine (VM). Consequently, a mechanism for sending and receiving data between VMs is required. To achieve this, all nodes in the cluster are connected to the same network within the Docker Compose file the **cluster-network**.

The project is highly scalable and expandable. You can easily extend it by adding new nodes with unique port numbers and notifying the bootstrap node accordingly. Additional configurations can be adjusted based on the cluster size, workload, and other parameters, as detailed in the **node configurations, multithreading, and locks** sections.

Every node within the system is configured to listen on a specific port. In the docker-compose file, both a host port and a container port are defined for each node. These port assignments, as specified in the docker-compose file, are then shared among the various services outlined in the docker-compose configuration. Table (3) illustrates the host and container ports that are used for different services.

Node	Ports (Host:Container)
Bootstrap-node	3000:3000
cluster-node-0	8080:8080
cluster-node-1	8081:8081
cluster-node-2	8082:8082

Table 3: System Services Port Assignment

XVI. CONCLUSIONS

This report illustrated the solution of the Atypon's 2023 java and devops bootcamp (fall 2022) Capstone project decentralized cluster-based NoSQL dB system. The assignment is successfully solved and coded Java and Spring Framework technology. Java is used successfully to build an application that simulates the interaction between users and nodes inside a decentralized NoSQL DB cluster. All assignment constraints and requirements are fulfilled based on the given capstone project description statement.

In this work, detailed design and verification of the application is presented, highlighting the DB implementation, the data structures used, the multithreading and locks engineered, the data consistency issues in the DB, the node hashing and load balancing, the communication developed protocols between nodes, and the safety and security issues.

The design, implementation and code testing and verification is successfully carried out using standard techniques such as; the clean code principles (Uncle Bob), "Effective Java" Items (Jushua Bloch), the SOLID principles, design patterns and DevOps practices..

Finally, the Docker Engine was used successfully to deploy the application in an efficient and secure manner in the form of containers. Packaging of the software code with just the operating system (OS) libraries and dependencies required to run the code to create a single lightweight executable that runs consistently on any infrastructure was possible with using Docker's Engine generated Containers.

V. REFERENCES

[1] What are NoSQL databases?, https://www.ibm.com/topics/nosql-databases, September 21, 2023

[2] Nayak, A. & Poriya, A. & Poojary, Dikshay. (2013). Article: Type of nosql databases and its comparison with relational databases. International Journal of Applied Information Systems. 5. 16-19.

[3] What are NoSQL databases?, https://www.mongodb.com/nosql-explained, September 21, 2023

[4] Biswajeet Sethi, Samaresh Mishra, Prasant ku. Patnaik. (). Article: A Study of NoSQL Database. International Journal of Engineering Research & Technology (IJERT). Vol. 3 Issue 4. 1131-1135.

[5] What are NoSQL databases?, https://www.oracle.com/jo/database/nosql/what-is-nosql/, September 21, 2023

[6] Dr. Fahed Jubair, Motasem Aldiab, Rami Zeineh, Java and Devops Bootcamp (Fall 2022), Capstone Project Decentralized Cluster-Based Nosql Db System, Atypon Training Program 2023.