ATYPON NOSQL DATABASE

BY AMMAR ABU YAMAN

Nov 2022

NoSQL Document Cluster Database

A Report describing My Design and implementation of a NoSQL Cluster database discussing the design patterns used, describing the data structures used and defending the design against SOLID principles and Clean Code Standards.

Atypon NOSQL Database

BY AMMAR ABU YAMAN

Table of Contents

Table of Figures

1. INTRODUCTION

1.1 Overview

In this Documentation I will discuss my Design of a NoSQL database running on a cluster and how I implemented the design. It was challenging project that required a lot of research and learning and experimentation.

The database is a document database that uses JSON objects to store data.

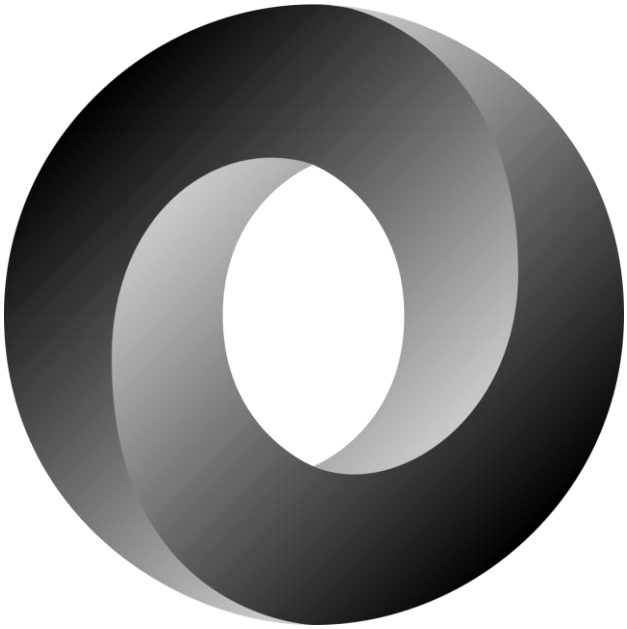

FIG 1. JSON LOGO

The database is running on a Multithreaded decentralized Clustered where the data, schemas and indexes are replicated on all the nodes in the cluster and each node uses multithreading to serve multiple requests simultaneously.

It allows authorized users to do CRUD operations to manipulate the data and indexes and databases (Collections) within the database based on a Role-Based Access Control (RBAC) to restrict certain operations to users with the right access.

2.2 Design Requirements and Objectives

This section describes design requirements and the objectives that needs to be met by the implementation. These requirements are obtained based on the provided project outline and research I did based on similar technologies and established solutions.

A small note, I use the term database to mean a table (or a collection) of documents, this is an important terminology to keep in mind.

2.2.1. Design Objectives

- The project should be designed with effective use of design patterns.
- The implementation should adhere to SOLID principles.
- The implementation should to clean code principles and practices.
- The implementation should use performant data structures and algorithms to perform critical operations such as indexing and caching.

2.2.2. Database Requirements

- The database should support the creation and delete databases (e.g., Student database, Class database, Products database).
- Create and delete Indexes on a single JSON property within documents in a database to speed up look up of documents.
- Support full CRUD operation on documents within the database.
- The database support caching to speed up read operations which are the majority of operations in the system

2.2.3. Clustering Requirements

- The database should operate on a decentralized cluster of nodes.

- Each node should support the full range of operations provided by the database system.
- A bootstrapping node is responsible for providing configurations information and users information to worker nodes so they are able to function correctly.
- Provide load balancing mechanisms for distributing users on nodes evenly.
- Migrate Read operations from a congested nodes to other nodes in the system.
- Ensure only the node with an affinity to a particular document to perform write operations on said document.

2.2.4. User's Requirements

- Users should be allowed only to login to their assigned nodes.
- Users should be restricted from performing certain operations if they have no access to them based on their roles.
- Users should interact seamlessly with the database without feeling the effects of requests redirections and load balancing in the cluster.
- New users should be able to register and be assigned to a node from the bootstrap node.

2.2.5. Users types

Users have levels of access based on a Role-Based Access Control (RBAC) model that control which operations they are allowed to perform in the database.

The users types and privileges

- Admin: Create and delete indexes and databases.
- Editor: Create, delete and update documents in a database.
- Viewer: Read document(s) from a database.

2. DESIGN PATTERNS

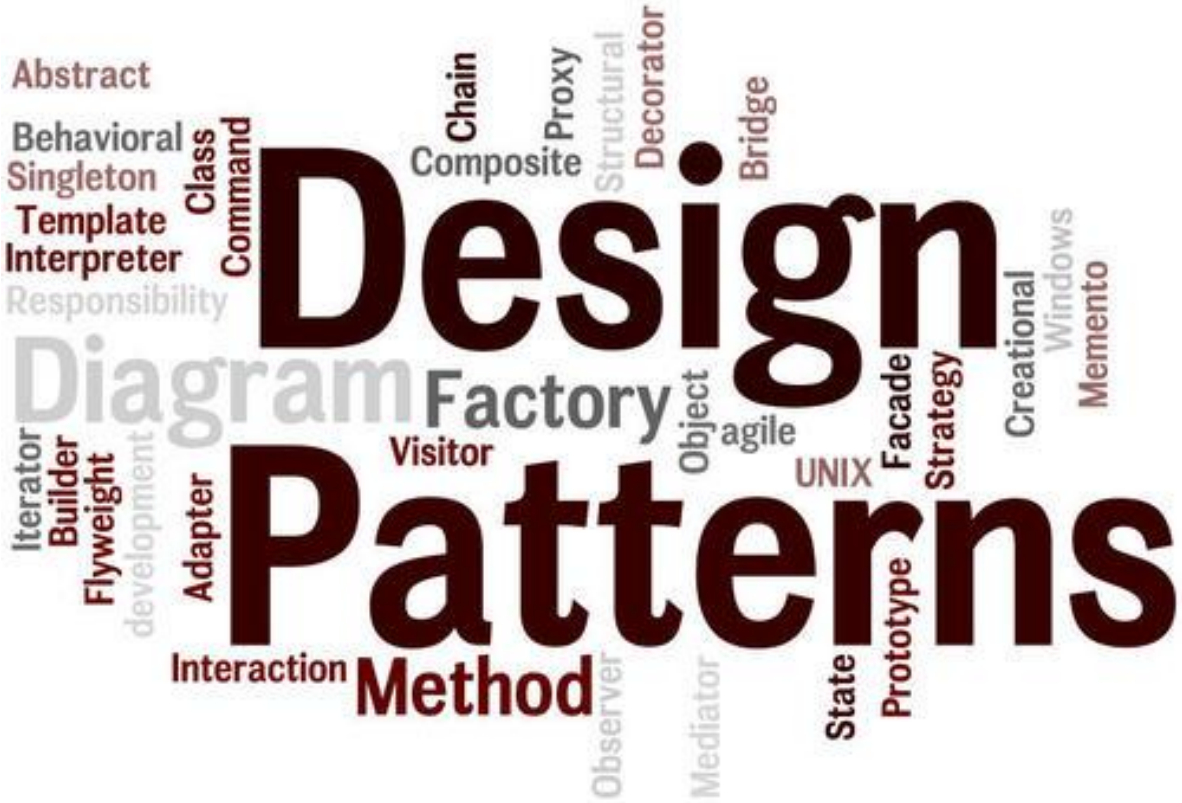

FIG 2. DESIGN PATTERNS

2.1. Why Use Design Patterns?

Software must be designed for while expecting change, often a project's requirements change during the development cycle and software is expected to evolve and change over the course of its life time. That is why it's called software after all because it should be mailable and easy to change.

Design patterns are employed to strengthen the design of a project, to help in organizing the various parts of the system, to reduce the **coupling** in the system's components to make them easier to change.

The following is a description of the design patterns used in the implementation.

2.2. Builder Design Pattern

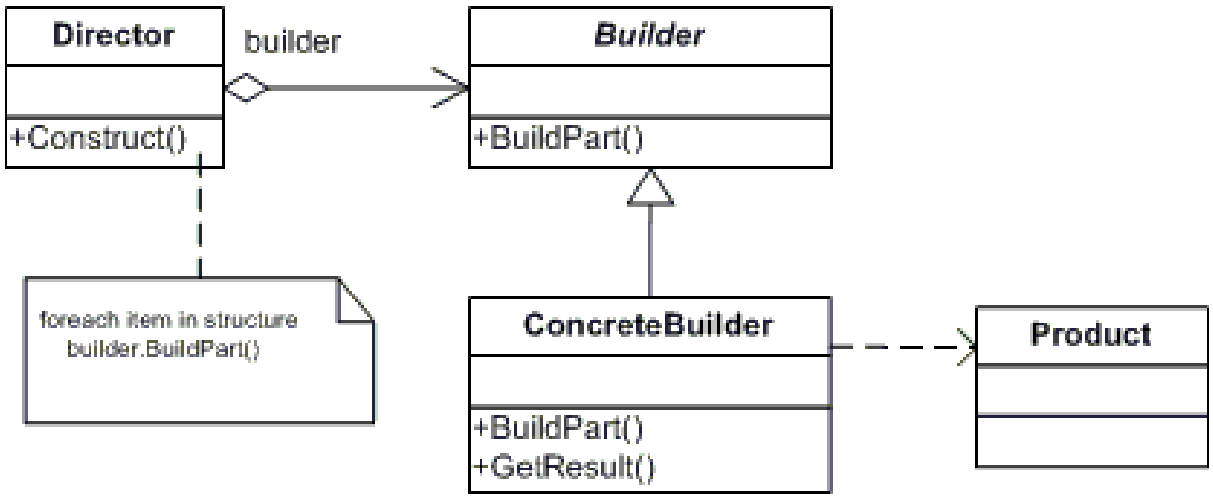

FIG 3. BUILDER DESIGN PATTERN

The Builder design pattern simplifies the creation of complex domain objects with many attributes, by facilitating the adding of attributes one at a time.

It greatly increases the flexibility and the code readability when creating complex objects over traditional constructors with many fields.

In my project I used **Lomok** library to automatically generate builder classes.

For example The **Query** class used for passing queries in the database is a good use case for the builder design pattern.

```
@Getter
@Setter
@Builder
public class Query implements Serializable {

    private String databaseName;
    private JsonNode oldData;
    @Builder.Default
    private boolean hasAffinity = false;
```

FIG 4. BUILDER PATTERN USE

2.2. Singleton Design Pattern

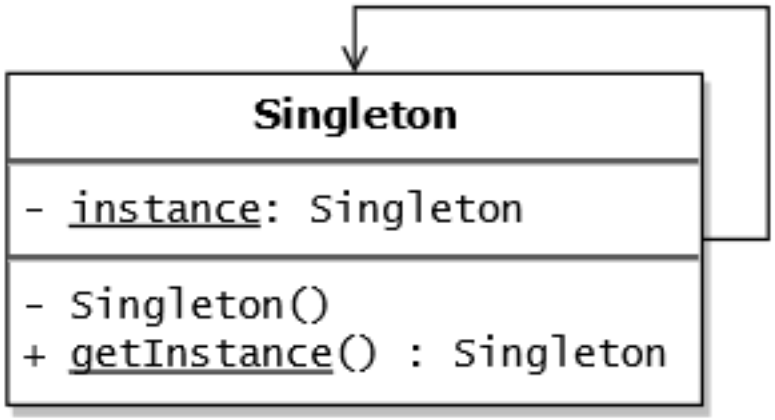

FIG 5. SINGLETON PATTERN

The singleton design pattern is employed when you need only one instance of an object to coordinate actions across the system.

In my use case I have **DatabaseManager** class that is responsible for initializing the system and hold the services needed in the system to function properly, but I only need to initialize the system once and I only need one instance of the services and data held by this class this is why I applied the singleton design pattern and can enforce the usage of this single instance by having **private constructors.**

```
public class DatabaseManager {

    private static DatabaseManager INSTANCE;

    private DatabaseManager() { }

    private static final int DEFAULT_CONGESTION_THRESHOLD = 1000;

    public static DatabaseManager getInstance() {
        if (INSTANCE == null)
            initialize();
        return INSTANCE;
    }
```

FIG 6. SINGELTON PATTERN USE

2.3. Data Access Object (DAO) Design Pattern

DAO design pattern is used to separate the data persistence logic in a separate layer. This way, the service is unaware of the low level operations to used to store and load the data. This design help to achieve separation of concerns.

One of the areas the DAO is used in my project is to implement a class abstract away dealing with reading and writing documents to a and from a database, hidden behind an interface.

```
public abstract class Database {
    private String name;
    private File dataDirectory;

    public abstract void drop();
    public abstract  void addDocument(String docIdx, JsonNode
document);
    public abstract void deleteDocument(String docId);
    public abstract void updateDocument(JsonNode fieldsToUpdate,
String docId);
    public abstract JsonNode getDocument(String docId);
    public abstract Stream<JsonNode> getDocuments(Collection<String>
docIds);
    public abstract Stream<JsonNode> getAllDocuments();
    public abstract boolean contains(String docId);
```

2.4. Chain of Responsibility Design Pattern

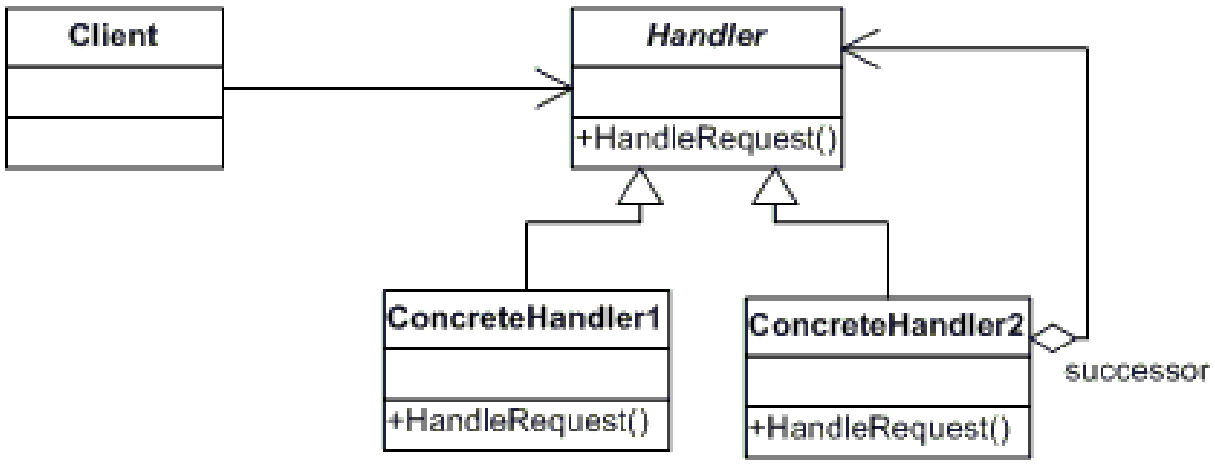

FIG 7. CHAIN OF RESPONSIBILITY

Chain of Responsibility is a design pattern that let you pass requests along a chain of handlers, it let you process requests in various ways and in a specific sequence of steps. It let each handler decide whether to pass the request to the next handler or to terminate processing of a request and return a result or an error.

It also promotes lose coupling between system components as each handler is responsible for doing one type of processing on the request and has one responsibility in the Total processing of the request.

This design pattern is central to my implementation, as I modeled my system as a chain of handlers where queries pass through each handlers and get processed by one handler at a time and then passed to the next handler or stops in the chain and return a result.

```
public abstract class QueryHandler {

    protected QueryHandler nextHandler;

    public abstract void handle(Query query);
    public QueryHandler setNext(QueryHandler handler) {
        nextHandler = handler;
        return handler;
    }
```

```
    protected void pass(Query request) {
        if(nextHandler != null)
            nextHandler.handle(request);
    }
```

FIG 8. QUERYHANLDER CLASS

For example, I have a Cache handler that intercepts read requests and return the cached result if it is present in cache or pass the requests to the next handler to be read from disk if it's not present in cache.

3. CLEAN CODE PRINCIBLES

3.1. Motivation

Clean code principles keep our codebase readable and understandable and eliminate code smells, and with readability and understandability it becomes easier to maintain and refactor and extend the code base.

3.2. Comments

Comments are essential for understanding code and creating documentation of the project however comments can be misused and lead worsened understandability of code, few examples of that is by having

- Very long comments: this usually mean that the code is written with poor readability and is obscure and can't be understood without long comments and this is usually a sign of poor design.
- Obscure or comments containing misinformation as this can lead to wrong understanding of the code's functionality.

My approach in the project is to keep comments brief and descriptive and avoid misinformation.

3.2. Naming

Naming classes, variables, functions, ...etc. plays an important role in the readability of code and understanding the intention of the programmer who wrote it. Therefore, in my project I took good caring of naming the facilities in

my program and follow certain set of rules to guarantee consistency and readability of the names as follow.

- Use Camel Case consistently to name all facilities in the project
- Using Pronounceable names that are easy to read and search in the project
- Intention revealing names
- Avoid disinformation in naming or using names that obscure the meaning or intention behind the code
- Avoiding cute names as they are unprofessional.
- Use names that are distinct and are easily distinguished from other names as to not be mixed with each other. For example avoid names that start the same prefix but have a different number at the end for such as **Add1** and **Add2.**
- One word per concept to keep the naming consistent and avoid referring to the same concept by multiple names.
- Avoiding magic numbers as my codebase uses named constants instead of literals.

A good example of my naming convention can be found in the Index interface of my project.

```
public interface Index extends Serializable {
    // search the index for documents with a specific values for the
index's field
    List<String> search(JsonNode key);
    // add a mapping from a value to the document id
    void add(JsonNode key, String documentId);
    // delete a mapping from a value to a specific document id
    void delete(JsonNode key, String documentId);
    // check if the index contain the
    boolean contains(JsonNode key);
    // clear the index of all values
    void clear();
}
```

FIG 9. NAMING CONVENTIONS

3.3. Formatting

3.3.1. Vertical Formatting

Files are usually small and are usually under 300 lines. This is quite desirable as it aids in the understandability.

3.3.2. Variables are declared close to their usage

3.3.3. Indentation and horizontal alignment

Horizontal alignment is not a good practice as it can lead to missing part of an expression or missing the assignment statement of a declaration.

Most lines are short and don't take half the width of the screen.

This helps in keeping each line easily readable and hard to miss crucial parts of it.

Some challenges arose during the application of this principle, specifically with expressions that require long method chaining such as **Streams** and **Builders.** The solution to this problem is to break each method call to one line, for example

A builder expression

```
Query request = Query.builder()
        .originator(Query.Originator.User)
        .queryType(QueryType.CreateDatabase)
        .databaseName(databaseName)
        .build();
```

FIG 10. BUILDER EXPRESSION

A stream expression

```
return Files.walk(databaseDirectory.toPath())
        .skip(1)
        .map(path -> path.getFileName().toString().split("\\.")[0])
        .map(documentIndex -> getDocument(documentIndex));
```

FIG 11. STREAM EXPRESSION

Indentations are also used consistently in the codebase to improve the readability.

3.4. Functions

3.4.1. Names

As mentioned in the **3.2** names have to be readable, descriptive, consistent …etc. this also applies to functions.

3.4.2. Function Arguments

Long argument lists can harm the readability of the code thus they are avoided. Therefore, almost all functions in the project take 0 to 3 arguments and any facility that require more than that is usually handled by the Builder Design Pattern.

3.4.3. Dead Functions

Dead functions that aren't used anywhere in the program are bad to have in a code base, Therefore, any unused functions are eliminated from the codebase.

3.4.4 Simple functions

Most functions in the implementation are small and only do one thing, and complex functions are broken into smaller and simpler functions.

3.4.5. DRY principle

Common and reusable logic is extracted into functions to be reused instead of implementing the same functionality in multiple places

For example, the **sendToNodes** functions that broadcast updates to other nodes in the system is reused in multiple places.

```
private void sendToNodes(String action, String info, String body) {
    HttpHeaders headers = new HttpHeaders();
    headers.setContentType(MediaType.APPLICATION_JSON);
    HttpEntity<String> entity = new HttpEntity<>(body, headers);
    for (Node node : nodes) {
        try {
            new RestTemplate().postForEntity(format(URL,
```

```
node.getAddress(), action, info), entity, String.class);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

FIG 12. SEND TO NODES FUNCTION

3.5. Objects Structure

3.5.1. Member variables are always private and only accessed via Getters and Setters

3.5.2. Interfaces and Abstract Classes

Interfaces and abstract classes are used to expose functionality required by the component that require that it while encapsulating implementation details in derived classes and reducing the coupling with the implementation.

Examples.

```
public interface Index extends Serializable
```

```
public interface Cache <K, V>
```

```
public interface SchemaValidator
```

```
public abstract class QueryHandler
```

```
public abstract class Database
```

4. SOLID PRINCIPLES

3.1. Introduction

SOLID is an acronym for the five object oriented design principles.

These principles establish practices that drives writing readable, maintainable and extendable code and reduces the cost of refactoring and evolving the software as the project grows. They also help in reducing code smells and reduce the coupling between components of a system.

In this chapter I will describe how I applied each of the SOLID principles on my project.

3.2. Single Responsibility Principle (SRP)

Almost all classes in my project have only one thing to do and only one reason to change with complex functionality and responsibilities being split into multiple classes where each one is responsible for one part of the functionality.

For example, the schema responsibility in my project consists of three parts

- **SchemaValidator**: responsible for validating the schema and ensuring documents conform to the schema
- **SchemaStorage**: responsible for the persistence of database schemas and saving and loading them from disk
- **SchemaHandler**: responsible for the schema verification in the process of handling queries.

When a change need to a part of the system such as how to persist schemas only one class have to be changed or extended, and I follow this rule for most of the classes in my project.

3.3. Open-Closed Principle (OCP)

Classes should be open for extension and closed for modification.

In my project if I needed to add a new functionality I can add a new Handler and create a new chain to handle this new functionality without modifying the existing handlers.

And if I needed to add a new functionality say to the **DatabaseHandler** I would add a new class and add the construction of that class to the **DatabaseHandlers** factory class without modifying the **DatabaseHandler** class.

```
@Override
public void handle(Query query) {
    if (query.getQueryType() != QueryType.CreateDatabase
            &&
!databaseService.containsDatabase(query.getDatabaseName())) {
        query.getRequestOutput().append("No Such Database Exists");
        query.setStatus(Query.Status.Rejected);
        return;
    }

    QueryHandler handler = DatabaseHandlers.getHandler(query,
nextHandler);
    handler.handle(query);
}
```

FIG 13. DATABASE HANDLER USING FACTORY PATTERN

3.4 Liskov Substitution Principle (LCP)

the Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of its subclasses without breaking the application.

While I didn't use this principle much as I didn't use inheritance much and favored composition, I would have definitely kept it in mind if I did.

3.5 Interface Segregation Principle (ISP)

The ISP Clients should not be forced to depend upon interfaces/methods that they do not use, for example fat interfaces that force subclasses to implement methods they are not concerned with or default implementation of interface methods.

In my project I designed my interfaces carefully to include the minimum amount of functionality that must be implemented by subclasses and nothing and

keeping each interface concerned with only one responsibility and I didn't have any default implementations to interface methods.

But when I needed common functionality among all subclasses, I used abstract classes and added only the methods that need to be implemented by subclasses as abstract, for example the **QueryHandler** abstract class.

```
public abstract class QueryHandler {

    protected QueryHandler nextHandler;

    public abstract void handle(Query query);
    public QueryHandler setNext(QueryHandler handler) {
        nextHandler = handler;
        return handler;
    }
    protected void pass(Query query) {
        if(nextHandler != null)
            nextHandler.handle(query);
    }

    protected boolean updateNotNeeded(Query query){
        return query.getStatus() != Query.Status.Accepted;
    }
}
```

FIG 14. QUERYHANDLER CLASS

3.6 Dependency Inversion Principle (DIP)

The DIP states that high level modules should depend on high level generalizations and not on low level details and that low level details.

In my project classes always depend on interfaces or abstract classes and not on concrete types, and implementations of these interfaces and abstract classes do not depend on Concrete classes, and if they do then these classes are usually a wrapper around an interface or an abstract class, for example the **DatabaseService** class is a wrapper around the **Database** Interface which doesn't depend on concrete classes.

5. EFFECTIVE JAVA

Effective Java is an influential book for anyone who want to write professional Java code. I used many of its principles that I found suitable to my project; in the following points I discuss some of them.

Point 1. Consider a builder when faced with many constructor parameters:

When I had classes with many fields I opted to the builder pattern instead of constructors to create instances of these classes; I utilized @Builder annotation from Lombok library to implement these builders.

```
Query request = Query.builder()
        .originator(Query.Originator.User)
        .queryType(QueryType.CreateDatabase)
        .databaseName(databaseName)
        .payload(mapper.valueToTree(schema))
        .build();
```

FIG 15. BUILDER PATTERN USING BUILDER NOTATION

Point 2. Enforce the singleton property with a private constructor or an Enum type:

When I needed a single instance of the **DataManager** class I opted to create a one static instance and hide it behind a **getInstance** method while making the class's constructor private.

Point 3. Avoid creating unnecessary objects:

I tended to reuse objects when possible, to avoid creating redundant temporaries. One notable example is the **ObjectMapper** class used to parse JSON, instead of creating multiple redundant instances of this class each time I needed it, I created one instance and reused it whenever I needed it.

Point 4. Avoid finalizers and cleaners:

In my project I avoided using finalizers or cleaners as they run in a non-deterministically and cannot be relied on.

Point 5. Prefer try-with-resources to try-finally:

I often used try-with-resources feature whenever possible as it is safer than finally blocks where you can forget to put clean up code.

```
try (PrintWriter writer = new PrintWriter(new
FileOutputStream(documentFile, false))) {
    writer.write(updatedDocument.toString());
}
```

FIG 16. TRY-WITH-RESOURCES

Point 6. Prefer lists to arrays:

Lists simplify working with generic code and I opted to using them whenever possible.

Point 7. Use enums instead of int constants:

In my project I used enums when faced with a fixed category of options as they are cleaner and safer and strongly typed compared to ints.

```
public enum Role {Admin, Viewer, Editor}
```

Point 8. Prefer primitive types to boxed primitives:

Primitive types are more space efficient, faster and safer. I almost always used them instead of their boxed counterparts (e.g., Integer, Boolean, Double …).

Point 9. Beware the performance of string concatenation:

String concatenation using + operator requires creating redundant copy on each operation as the String data type is immutable.

I opted to use the **String.format** method when I needed to concatenate strings instead.

Point 10. Consistently use the Override annotation:

In my project I always used the @Override annotation on overridden methods. This helps with readability of the code.

```
@Override
public void drop() {...}

@Override
public void addDocument(String docIdx, JsonNode document) {...}

@Override
public void deleteDocument(String docId) {...}

@Override
public void updateDocument(JsonNode fieldsToUpdate, String docId) {...}

@Override
public JsonNode getDocument(String docId) {...}
```

FIG 17. OVERRIDE ANNOTATION USAGE

Point 11. Refer to objects by their interfaces:

Avoid using concrete types when referring to object instances or in function parameters as this increase the flexibility and make it easier to change the implementation at a later point in the project.

```
Index index = new BTreeIndex(5, new JsonComparator());
```

Point 12. Use exceptions only for exceptional conditions:

Exceptions are only used for exceptional cases when something unexpected happen such as an IO error. One instance where I refrane from is when a method cannot return a value, I instead opt for using **Optional** instead.

Point 13. Always override hashCode when you override equals:

Whenever I need an equal or a hashcode override I always override them both and use the IDE to automatically override them both at the same time.

Point 14. Use interfaces only to define types:

I used interfaces only to define types with abstract methods to be overridden, no constants or other items were included in interfaces.

Point 15. Prefer for-each loop to traditional for loops:

Whenever possible I opted to use for-each loops instead of c-style loops, especially when iterating over collections.

```
for (String field : requiredProperties) {
    if (!node.has(field))
        continue;
    properties.put(field, node.get(field));
}
```

FIG 18. FOR EACH LOOPS

Point 16. Know and Use Libraries

In this project I researched libraries to help me in this project and made an effective use of them, I used

- **Spring boot**: for building the API and for networking aspects
- **Lombok**: for implementing builder classes and implementing getters and setters
- **Jackson**: for parsing and handling JSON processing
- **Junit**: for writing automated tests
- **JBcrypt**: for handling hashing passwords and comparing hashes with plain passwords

Point 17. Synchronize access to shared mutable data:

Access to mutable data is protected using the LockHandler which provides required locks to synchronize access to mutable resources.

Point 18. Prefer lambdas to anonymous classes:

Lambdas are cleaner and simpler than anonymous classes especially when it comes to simple one-line functions, I used them quite often especially when working with streams.

```
Optional<JsonNode> optional = DatabaseUtil.indexRequest(query,
database)
        .stream()
        .limit(1)
        .map(documentId -> database.getDocument(documentId))
        .findFirst();
```

Point 19. Document all exceptions thrown by each method:

It is important to document exceptions, as it helps with debugging and locating errors that happen in the program, thus each error is logged to aid in that purpose.

```
catch (Exception e) {
    e.printStackTrace();
    query.setStatus(Query.Status.Rejected);
    query.getRequestOutput().append(e.getMessage());
    logger.warning(e.getMessage());
}
```

6. IMPLEMENTATION

6.1. Introduction

This chapter discusses implementation details and how the overall system fits together.

It explains Bootstrapping, Users API, node to node communication, query processing, load balancing, security considerations.

6.2. Bootstrapping

The bootstrapping step is responsible for starting the cluster and providing configurations to worker nodes so they may function correctly and handler users' queries.

The bootstrapping step is split into two parts.

6.2.1. Bootstrapping Program

The bootstrapping program is responsible for starting Bootstrap node Docker container and the workers docker containers.

6.2.2. Bootstrapping Node

The bootstrapping node is responsible for providing Users' information and Nodes information and IP addresses to worker nodes in the.

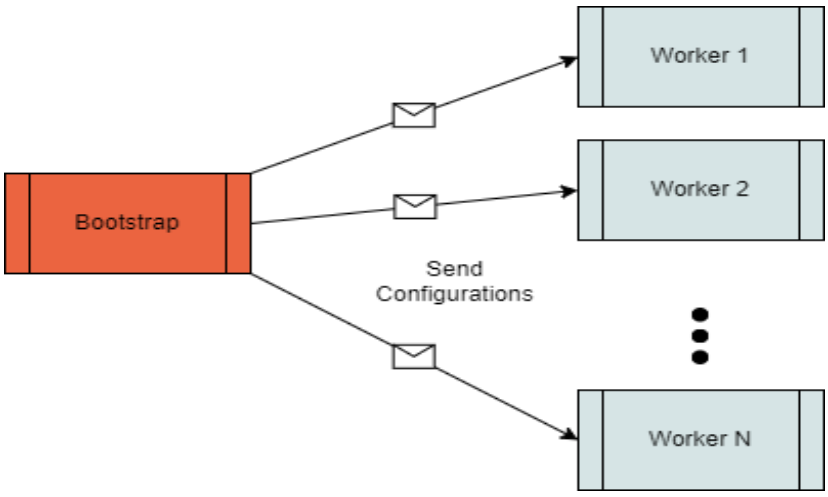

FIG 19. BOOTSTRAP NODE SENDING CONFIGURATIONS TO WORKER NODES

6.2.3. New Users

After the nodes are configured, the bootstrap node becomes an entry point for new users where they can register and get assigned a node by the bootstrap.

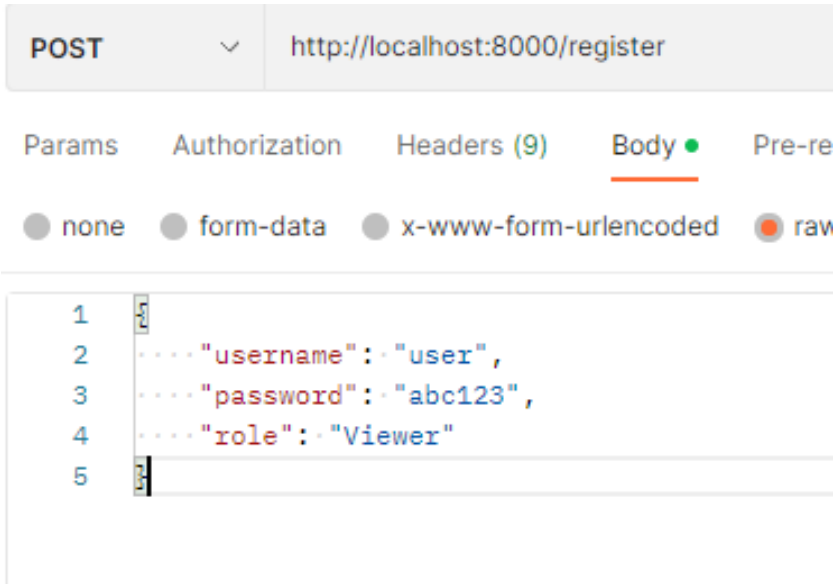

FIG 20. USER LOGIN CREDENTIALS

6.2.4. Users Assignment Load Balancing

To keep nodes balanced, the bootstrap node assign new users to the node with the least number of assigned users to keep the users equally distributed over the nodes in the system.

6.3. API

The users log in to their assigned nodes and issue queries to the database via a REST API built using Spring boot.

FIG 21. SPRING BOOT

After logging into their assigned nodes, users can issue http requests to endpoints to perform CRUD operations on the database.

The available endpoints for users are.

Database Create

POST: /database/create/{database name}

- Schema provided as a JSON body to the request.

Used to create a database with a specific schema.

Database Delete

POST: /database/delete/{database name}

Used to delete an existing database with all indexes assigned to it.

Index Create

POST: /index/create/{database name}/{property name}

Used to create an index on a specific property in a database.

Index Delete

POST: /index/delete/{database name}/{property name}

Used to delete a specific index on a property in a database.

GET: /document/find/{database name}

- Optional JSON property **filter** supplied in the body that can be used to search for documents with a specific key value pair (can be used to index the request if an index exists for the key).
- Optional JSON property **requiredProperties** supplied in the body that can be used to return only desired properties of the document to the user.

Used to find the first document in the database.

GET: /document/finds/{database name}

Same as previous except it finds all the matching documents

POST: /document/add/{database name}

Used to add document to a database, the document is provided as the JSON field **payload** in body of the request.

POST: /document/delete/{database name}

- Required JSON property **filter** supplied in the body will can be used to search for documents with a specific key value pair (can be used to index the request if an index exists for the key).

Delete the first matching document from the database

POST: /document/update/{database name}

- Required JSON property **filter** supplied in the body, will can be used to search for documents with a specific key value pair (can be used to index the request if an index exists for the key).
- Required JSON property **payload** supplied in the body, contains JSON properties with new values

Updates the first matching document from the database with **payload** fields

6.4. Query Processing

6.4.1. Overview

To Process a query a REST endpoint such as /**document/find/{database}** must

- Build an instance of **Query** class using the builder API provided.
- Create a chain of handlers to handle the request usually through the **HandlerFactory** class for that specific query
- Pass the query object to the chain and return the output to the user

In this process three design patterns shine.

1. **The Builder** design pattern for building Query objects.
2. **The Chain of Responsibility** Design pattern for handling queries through a chain of handlers.
3. **The Factory** Design Pattern for creating chains of handlers to be used to handle the query.

Especially the Chain of Responsibility. As the user make a query a chain is constructed depending on the query type to handle the processing of the query.

The types of handlers exist in the project.

6.4.2. Synchrounization and LockHandler

The **LockHandler** provides the synchronization mechanism necessary to avoid race conditions during query processing.

Two types of locks are used

- **Per Database ReadWriteLock**: one lock per database, each read request gets assigned a read lock allowing multiple read requests to execute in parallel, while writes are given write lock allowing only one write request to execute at a time
- **Global ReadWriteLock**: global lock is used to grant exclusive access to the entire system for administrative queries such as Creating and deleting databases or indexes while other by giving them a write lock, while other types of queries such as reading and writing and adding documents get a read lock.

this system allows multiple read queries to be processed simultaneously while allowing one write query to be processed per database.

6.4.3. CacheHandler

The **CacheHandler** provides caching mechanisms for the database, it intercepts read queries and caches their output or provide the output directly if it is present within the cache.

Each database has its own fixed size cache that implements the Cache interface.

```
public interface Cache <K, V> {
    public Optional<V> get(K key);
    public void put(K key, V value);
    public boolean contains(K key);
    public void remove(K key);
    public void removeIf(Function<K, Boolean> filter);
    public int size();
    public long capacity();
    public void clear();
}
```

FIG 22. CACHE INTERFACE

Please refer to 7.1 for a more detailed explanation on how caching is implemented.

6.4.4. IndexHandler

The **IndexHandler** intercepts read and write queries and searches for an appropriate one to provide for the query to speed up the search process. It is also responsible for creating and deleting indexes on specific JSON properties in the database.

Each database has its own set of indexes that implement the Index interface.

```
public interface Index extends Serializable {
    List<String> search(JsonNode key);
    void add(JsonNode key, String documentId);
    void delete(JsonNode key, String documentId);
    boolean contains(JsonNode key);
    void clear();
}
```

FIG 23. INDEX INTERFACE

Please refer 7.2 for a mor detailed explanation on how indexing is implemented.

6.4.5. SchemaHandler

The **SchemaHandler** handles the creation and deletion of schemas, it also ensures newly added documents and updates to documents conforms to the database schema.

6.4.6. DatabaseHandler

The database handler is responsible for the actual reading and writing on disk and creating, deleting, and updating databases and documents.

6.4.7. BroadcastHandler

The **BroadcastHandler** typically sits as the last handler in the chain and is responsible of notifying other nodes in the cluster of updates to the database state such as adding new document or updates on a specific document to ensure data consistency between nodes.

Broadcast operations are done in the background on separate threads to those of query threads, this ensures that a query exist the system as soon as it is able to and release any locks or resources it holds to allow other queries to use them.

6.4.8. LoginHandler

The **LoginHandler** is responsible of validating users authentication information.

6.4.9. RegisterHandler

The RegisterHandler is used to register a new user into the system and assign a node to that user to be used by them. This handler's use is exclusive to the bootstrap node.

6.4.10. Putting handlers together

Different types of queries require different chains of handlers to be processed correctly, this section explains few of these combinations of chains.

Login Chain

FIG 24. LOGIN QUERY CHAIN OF HANDLERS

Registration Chain (Exclusive to Bootstrap node)

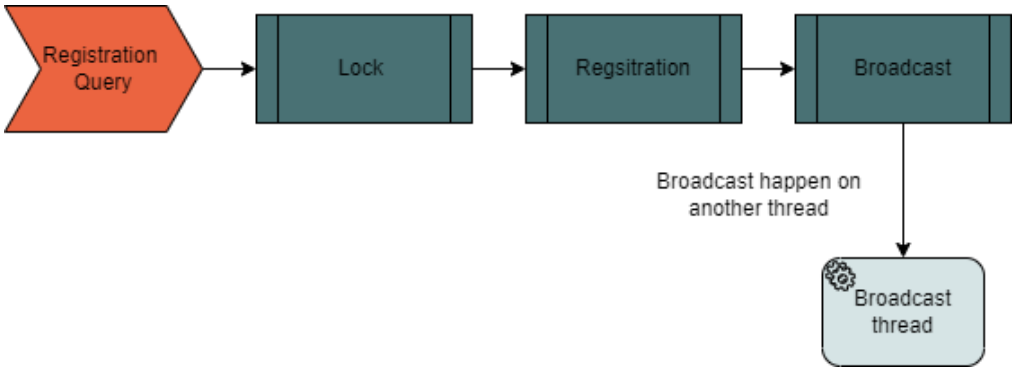

FIG 25. REGISTRATION QUERY CHAIN OF HANDLERS

Read Query Chain

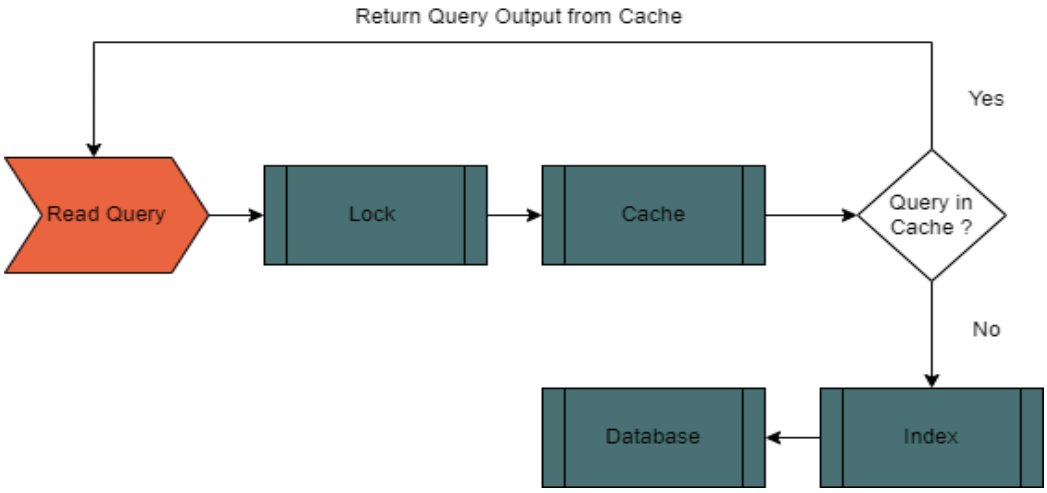

FIG 26. READ QUERY CHAIN OF HANDLERS

Write and Database creation and deletion chain

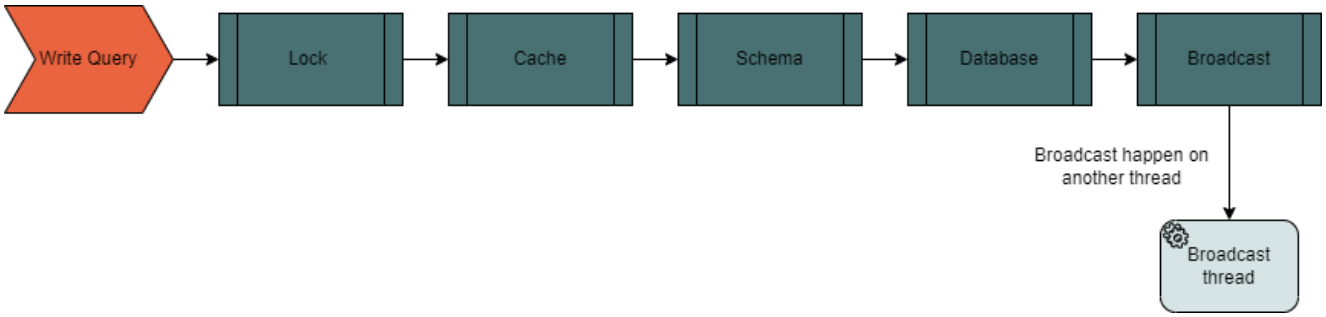

FIG 27. WRITE QUERY CHAIN OF HANDLERS

Index Creation and deletion chain

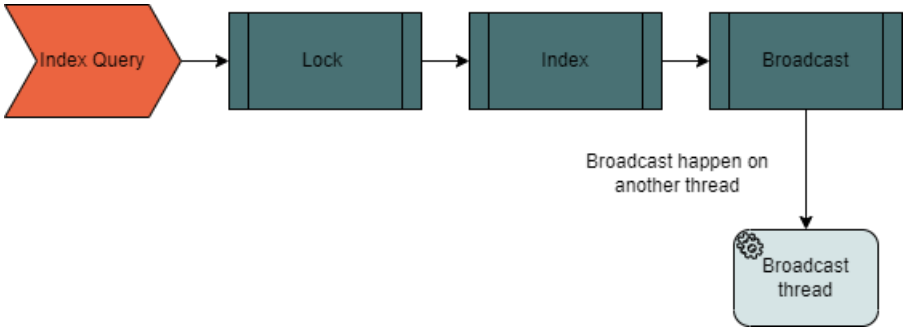

FIG 28. INDEX CREATION AND DELETION CHAIN OF HANDLERS

6.5. Node to Node communication

Nodes communicate over a docker network via a special REST API under /**_internal**/… path that is exclusive to nodes in the cluster.

Bootstrap node uses it to send configuration information to worker nodes in the cluster, and to send information about new users in the cluster.

Worker nodes use it to send broadcast messages about new documents, databases, indexes, schemas and redirecting write queries to on documents to nodes with affinity to that document.

6.6. Security Considerations

6.6.1. User's Security

Each user in the system is represented as follow.

Field	Type
Username	String
PasswordHash	String
Role	Enum { Admin, Viewer, Editor }
_Affinity	String

Users passwords aren't being saved directly into the database, instead a hash of their password is stored instead as this is more secured.

BCrypt is used to generate passwords hashes with a randomly generated salt and is also used for comparing plain passwords with passwords hashes when a node authenticates users' credentials.

A password hash looks like this

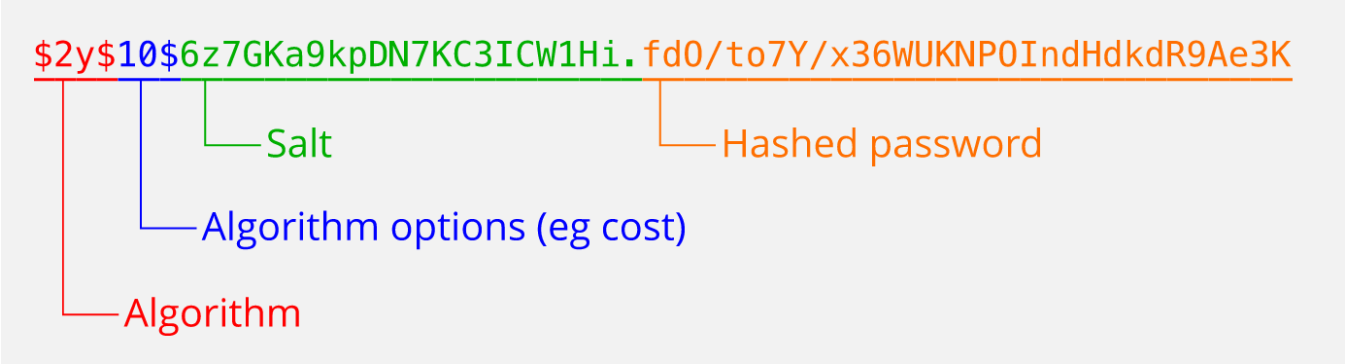

FIG 29. BCRYPT PASSWORD HASH BODY

When a user authenticates into the system a session is created for that user and is used to store the user's information such as their role, and this session is used to authenticate the user on subsequent requests.

6.6.2. Securing Node to Node communication

The REST API has two types of endpoints

- User Endpoints: This is for users to communicate with the database and issue queries, please refer to 6.3 for an explanation of this api
- Node Endpoints: This is reserved for node to node communication, it used for broadcasting messages, sharing configuration information, ..etc. please refer to 6.5 for a more in depth explanation.

The node to node communications are prefixed with /**_internal**/* path, and there is a need to secure this path so only nodes can use it, and this is done in two ways.

First Spring is configured to listen to connections on two ports, **8080** and **8000** and only **8080** is exposed via docker for users' connections and the other one is only accessible through a docker internal network where only the nodes in the cluster have access to, and it allows connections to /**_internal**/* set of paths only through port **8000** and reject any other ports.

Second any attempt entity that connect to /**_internal**/* though port **8000** is verified through it's IP address matches one of the IP nodes IP addresses.

6.7. ACID Requirements

atomicity, consistency, isolation, durability or ACID for short is a set of properties that a database transaction must guarantee despite errors or infrastructure failure.

Here I explain how these properties are implemented in the project

6.7.1. Atomicity

Atomicity treats each transaction as a single unit and if it fails at any stage the hole transaction is rejected and the database is left unchanged.

In my project if a query fails at any point it will not affect the information stored on disk and the query will not be broadcasted to other nodes in the network in case of write queries.

6.7.2. Consistency

Consistency means that a transaction can bring the database from one valid state to another without breaking database invariants.

In my project each query is checked for correct input from the user before being processed further for example when a user adds a new document to the database, this document is checked to ensure it matches the database schema before being stored and broadcasted and rejected if it doesn't.

```
if(!validator.validateDocument(schema, query.getPayload())) {
    query.setStatus(Query.Status.Rejected);
    query.getRequestOutput().append("Document added doesn't conform
to the database schema");
    return;
}
```

FIG 30. VALIDATING DOCUMENTS USING SCHEMAVALIDATOR

6.7.3. Isolation and Synchronization

As Transactions executed on multiple threads in parallel Isolation ensures that concurrent execution of transactions leaves the database in the same state that would have been obtained if the transactions were executed sequentially.

To ensure isolation of queries, synchronization mechanisms were used to ensure that changes made by queries take effect as if they executed sequentially and to prevent **race conditions**.

This is done by using locking mechanisms on each query before it enters the system, please refer to 6.4.2 for better explanation of the locking.

An interesting race condition can happen if two nodes had an update query on the same document but both don't have the write affinity on that document, in this case both send the write query to the node with the affinity for that document but only one of them will succeed as the first one to arrive will write the new data to the document.

To ensure data consistency between nodes in such a case the node that arrive late to the race condition will first check if the data in the node with the write affinity has the same data as its own. If it doesn't then the node with the affinity will reject the update query and node will attempt to update its current data to match that of the node with the write affinity and try to send the write query again.

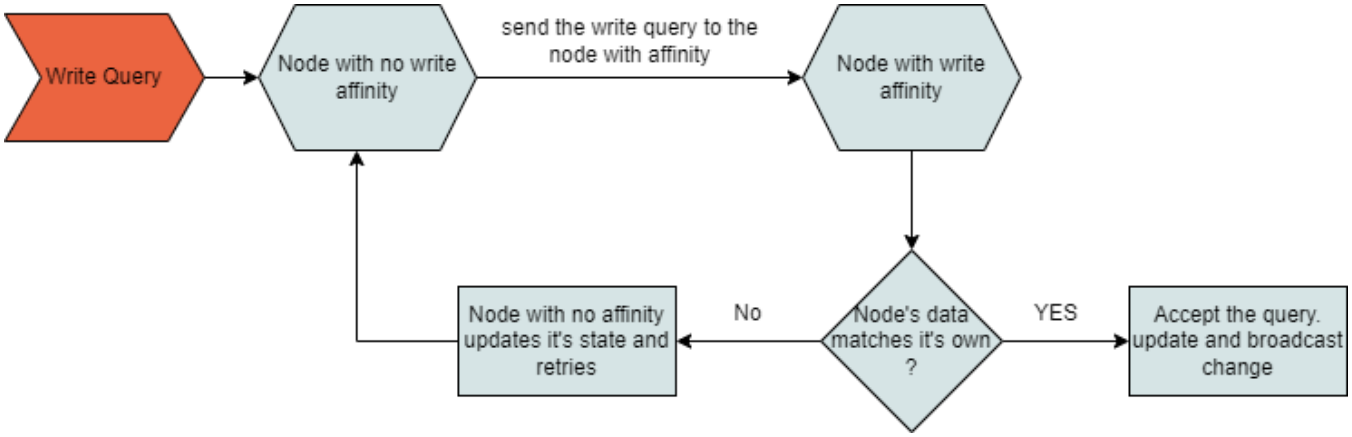

FIG 31. WRITE QUERY REDIRECTIONS DIAGRAM

6.7.4. Durability

Durability ensure that the data in the database and the current state in the database is not lost even in the case of a system failure.

To ensure durability in the database in case of a power failure, all the data, schemas, indexes and configurations are stored on disk permanently and all updates to these structures is committed to disk immediately to be restored at a later point in case of a system failure.

6.9. Testing

Testing on the system was conducted in two ways

Automatic testing using Junit which was used to test implementation correctness of various structures such as caching.

```
public class CacheTests {

    @Test
    public void testInsertions() {...}

    @Test
    public void testDeletion() {...}

    @Test
    public void testLRUEviction() {...}

}
```

FIG 32. CACHE TESTS

All automatic tests are run during build time using **Maven.**

Manual Testing was conducted using **POSTMAN** on a case by case basis

These tests were conducted utilizing docker where a bootstrap node was running and **Three worker nodes** were running, tests focused on ensuring load balancing between nodes and that a change in a state to one node was reflected on other nodes in the cluster.

The following is a listing of the tests conducted

New Users Testing

Test	Verdict (Pass/Fail)
New users can register using the bootstrap node	Pass
Users can only login to their assigned nodes	Pass
Bootstrap node assigns users to nodes in a load balanced way	Pass

Cache testing

Test	Verdict (Pass/Fail)
Cache was intercepting cached queries and returning correct results.	Pass
Writing to a database removed invalid cache entries	Pass
Cache results were consistent with direct database results	Pass

Index Testing

Test	Verdict (Pass/Fail)
Creating an index in one node reflected on the other nodes	Pass

Deleting an index in one node reflected on the other nodes	Pass
Database used indexes to speed up searching when one was available	Pass
Index content was consistent with content of the database	Pass

Schema Tests

Test	Verdict (Pass/Fail)
Schemas provided by users were validated correctly in the case of an invalid schema	Pass
Creating a schema in one node reflected on the other nodes	Pass
Deleting a schema in one node reflected on the other nodes	Pass
Newly added documents and document writes were correctly validated by the database in case of an invalid document	Pass

Write Tests

Test	Verdict (Pass/Fail)
Adding a new document in one node reflected on the other nodes	Pass

Deleting a document in one node reflected on the other nodes	Pass
Writes on a document with write affinity was correctly broadcasted and reflected on other nodes	Pass
Writes on a document with no affinity was correctly redirected to the node with write affinity on that document	Pass
Write affinities on documents were distributed among nodes in a load balanced way	Pass
Creating a database in one node reflected on other nodes	Pass
Deleting a database in one node removed all it's data, schema, indexes, cache on all nodes	Pass

6.8. Dev Ops Practices

6.8.1. Git and Github

Each project requires a version control system to keep track of changes happening to the codebase and mine is no difference, I used git as my choice for version control and Github as an online repository for the project.

FIG 33. GIT

6.8.2. Docker

Docker is an Open Source platform for building, deploying, and managing containerized applications, it uses OS-level virtualization to deliver packaged software and libraries bundled together in an isolated containers.

In my implementation I used docker containers to implement the cluster's nodes and used docker network to allow containers to communicate with one another.

FIG 34. DOCKER

6.8.3. Maven

Maven is a build and dependency management system for JVM languages and especially Java, it automate the build process and install required dependency utilizing the Maven Repository, there is also plethora of plugins to customize the build process and do automatic testing on each build.

I used Maven extensively in my project to handle dependency and automate my project's build process and package it into a JAR file to be used in docker containers.

FIG 35. MAVEN

6.8.4. Junit

Junit was used for writing unit tests and automate the execution of tests on each build of the project. Manual testing was also conducted on the project as well utilizing Postman.

FIG 36. JUNIT

7. DATA STRUCTURES USED

7.1. Caching Data Structures

7.1.1. Separate Cache per database

Each database (collection) has a unique cache assigned to it that caches recently accessed read queries on that database.

The **CacheService** class is responsible for managing caches and creating them and deleting them, it uses a **HashMap** data structure to map database name to its cache for efficient lockup of the appropriate cache to be used.

7.1.2. Cache Internal Structure (LRU)

Each cache has a fixed capacity of queries outputs it can hold and uses the **Least Recently Used (LRU)** mechanism to evict items from the cache when the number of cache entries exceeds the capacity.

The cache is implemented in an efficient manner to guarantee constant time **lookup** and **insertion** and **removal** of cache entries.

It uses a **Doubly Linked List** to hold the cache entries and to guarantee constant time **removal** and **insertion** of cache entries.

Data Structure of a Doubly Linked List Cache

FIG 37. DOUBLY LINKED LIST

It also uses a **HashMap** to map read queries to cache entries to guarantee constant time **lookup** of cache entries.

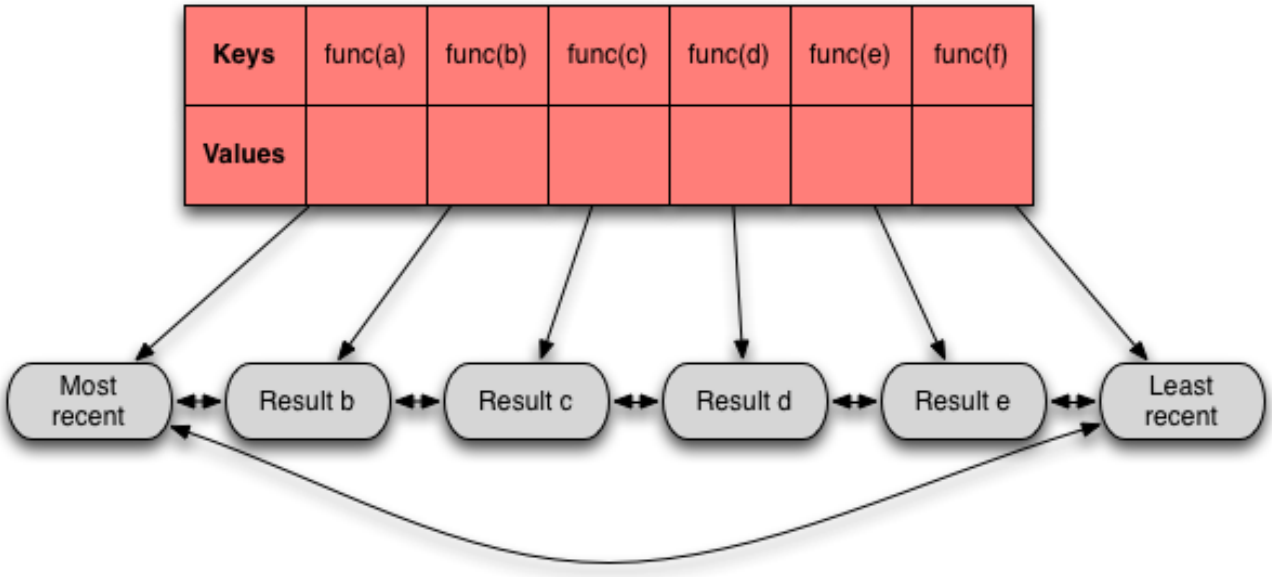

FIG 38. HASHMAP MAPPING KEYS TO LINKED LISTS VALUES

7.2. Indexing Data Structures

7.2.1. Mapping queries to Indexes

When the database needs to search of a document or a set of documents to serve a query it needs to efficiently find an appropriate index to use if one exists.

Queries are mapped to the appropriate index using the database (collection) name and name of the property to search on. A **HashMap** is used to do this mapping for constant time look up of indexes.

The **IndexService** class hold this mapping and is responsible of managing the creation and deleting and lookup of indexes.

7.2.2. Index Internal Structure (BTree)

Indexes are implemented using the BTree data structure. They are self-balancing trees that guarantee near logarithmic time look up, insertion and deletion of entries.

They are very similar in concept to Binary Search Trees however they allow each node in the tree can hold more than one key and can have more than two descendants.

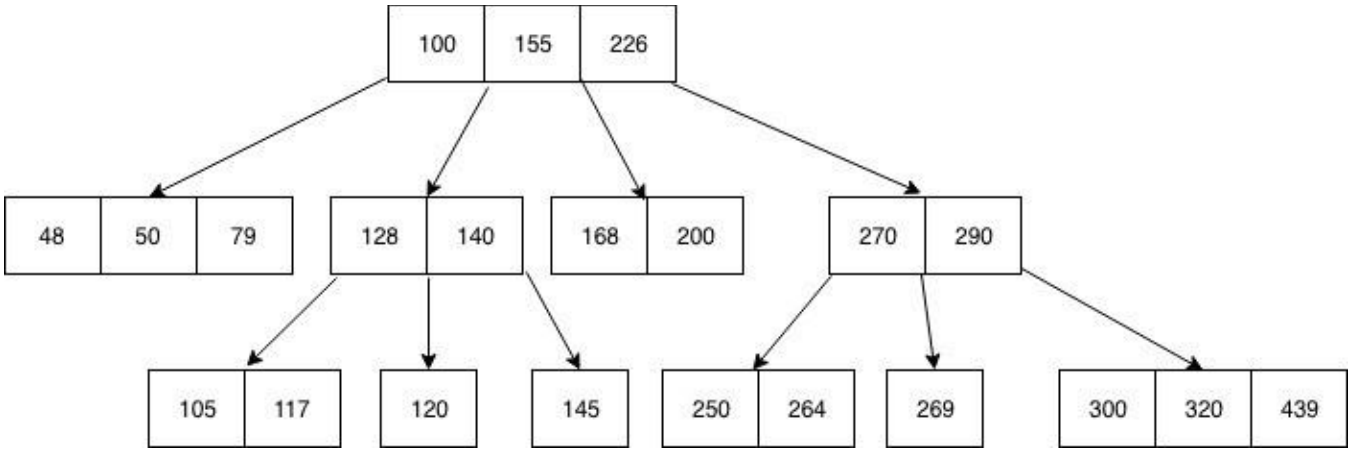

FIG 39. BTREE STRUCTURE, ALLOWING MORE THAN TWO CHILDREN AND MULTIPLE KEYS

The key advantage of this structure is it can handle arbitrary number of insertions and deletions while keeping the tree balanced and with minimum height especially as the number of entries.

As Nodes in Btrees nodes are often quite large they produce very flat trees.