Atypon/Wiley Inc, Amman, Jordan

Java and DevOps Bootcamp (Fall 2022)

Capstone Project

Decentralized Cluster-Based NoSQL DB System

Basheer Jarrah

Contents

Database Implementation

Designing decentralized cluster-based NoSQL database system involves deep understanding in many topics. Advanced database systems aim to serve large numbers of distributed users; such systems have dozens of interleaved tradeoffs that must be considered, and they have many functional requirements and non-functional requirements.

Database systems' clients expect performing their queries as quick as possible. Retrieving data quickly is a critical issue, there are many techniques to retrieve data quickly, such as indexing, replicating, caching and so on.

Writing queries may be acceptable to be some slower compared to reading queries, especially in system like our system, where we are dealing with replicas instead of shards; more formally; we are targeting synchronous consistency, each writing query must be directly replicated to ensure consistency in the whole cluster parts; this will be applicable as we expect rare writing operations although such thing can cause network traffic more than natural.

Secure connection is another matter in database systems, all mentioned standard and techniques must be fulfilled to achieve fast connection, registration need load balancing and hashing to be fast and secure, log-in has the same registration's needs and more, it may need indexing.

All these techniques and requirements will be discussed in the following sections.

File structure

Performing good writing produces better retrieving, designing well file structure system also will make retrieving easier, common NoSQL database systems have three level for a particular piece of data; database, collection and document, and document carry a set of properties, so in your way to access a specific property value, you need to identify all of these levels; but in our system, we eliminate

collection level, database which represented as a folder (or directory in Linux terms), it stores a set of documents, and each document carries some properties.

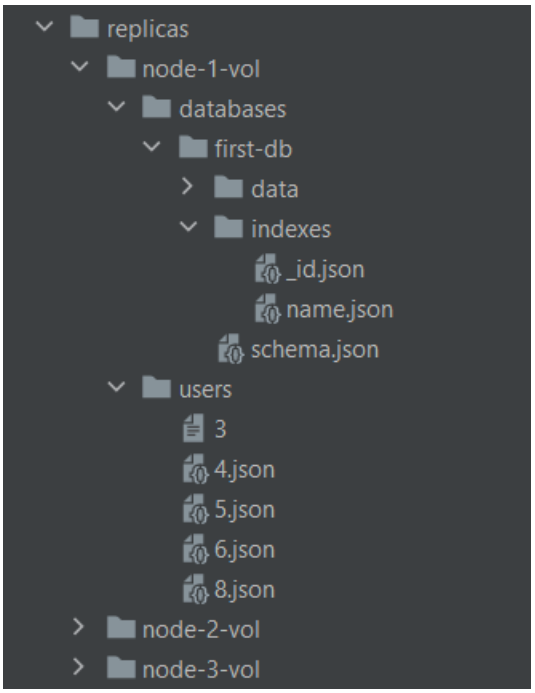

Figure1: File Structure

Figuare1 shows the used file structure for each node in my cluster, there are 10 nodes in my cluster, and each one of them has its Folder; which contains two inner folders, databases folder for created databases and their data, and users' folder for storing broadcasted users' information who registered in bootstrapping node. In databases folder, when a user sends database creation query, a folder is created and named with the passed name in the query, and then two folders are created inside created database folder; data folder and indexes folder, initially, data folder is empty until a document creation request is sent to that database. Index folder initially will contain one document called _id; this folder stores the indexed _id property for each document and will store all created indexes in that database in future.

Also at database creation moment, a JSON file called schema is created; in general, NoSQL databases are schema-less; this mean each document, which will be represented as a JSON file is not enforced to have a fixed structure or the same set of properties. The main goal for creating schema JSON file is to keep tracking for mandatory property for all documents, _id property; which must be indexed

directly after a database is created and must be updated once a new document is created.

Each folder is created intentionally in its place; separating data and metadata for each database will make the writing and reading queries functions cleaner.

Consider reading queries; suppose we but all data and metadata of a database in a single folder, we will face two bad options, first one is the results that are retrieved by queries contain some meaning-less data for the users, like indexes and schema files; even worse, retrieving authentication credentials for registered user, second bad option is cluttering reading methods with conditions for excluding such unrelated or secure data; but by the designed structure; all related data for a single job are combined in one and only one folder with meaningful name.

Such file structure is also useful with writing queries, where each write query needs to open some folders and files to read from; by convention, minimize the number of stored files in a folder will make opening process for that faster.

some of them are mandatory and rest are optional; there are three mandatory properties in each document, **_id**, **affinity** and **version**, and their jobs are known and straightforward, _id is for indexing and fast retrieving, affinity is for load balancing, and version is for consistency. _id is for indexing means _id property is indexed by default once a database is created and its index cannot be deleted by any client, rest properties can be indexed according to user as desired.

Design

My project is divided into X packages; package is a special type of folders group related classes together provide a clear hierarchical structure to your project.

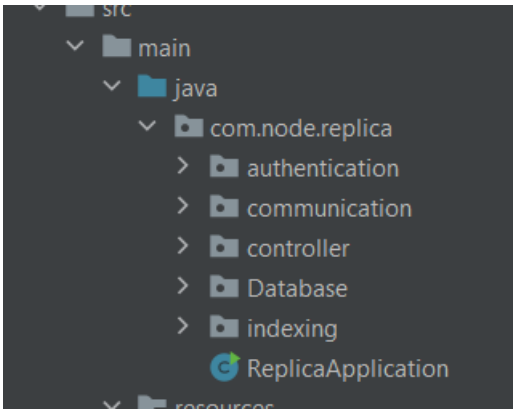

Figure2: packages in a node

Controller packages:

Controller is a special type of classes allows to handle all REST API requests, GET, POST and DELTETE, which reflect the basic queries in any database. With spring boot, a class can be treated as a controller simply by annotating it with **@RestController** annotation, and annotating its methods with mapping annotation prefixed with REST API request verbs; GET, POST, DELETE and so on.

In controller package, there are two controllers, **Query** controller and **Broadcast** controller.

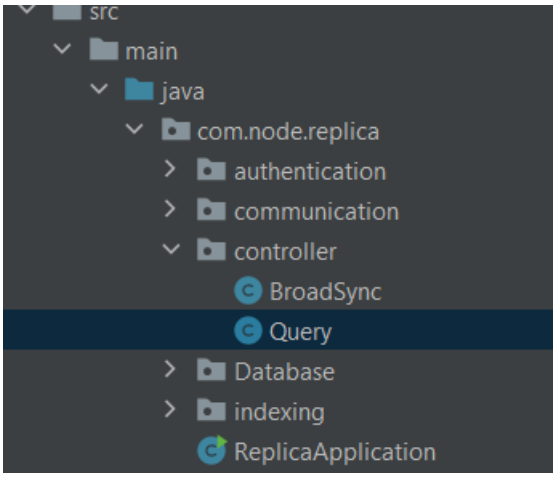

Figure3: controller package

Query controller is responsible for handling all requests that affects the data on the database system, it's a communication tool between the system and its users.

Broadcast controller is for communication between nodes in the cluster; it's main goal is to receive requests that are sent to ensure replicas consistency in the system.

Authentication controller is responsible for receiving requests from bootstrapping node for new registered users and mapped users.

Communication package:

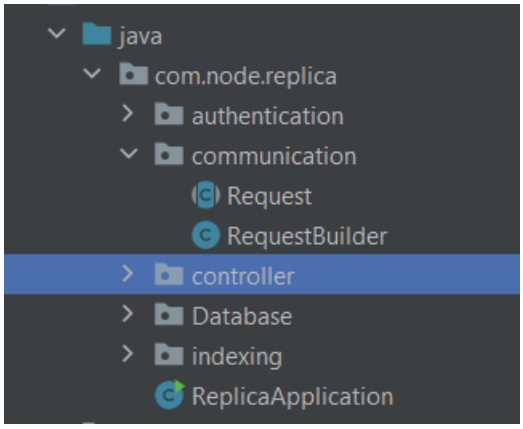

Figure 4: communication package

Communication package is simple and straightforward, it has Request class which is abstracted and a concrete extended class from Request class called RequestBuilder, this class follows Builder pattern (as I will explain in design patterns chapter), and their job is to make writing operations code more readable, standardized and data coupled.

Database Package:

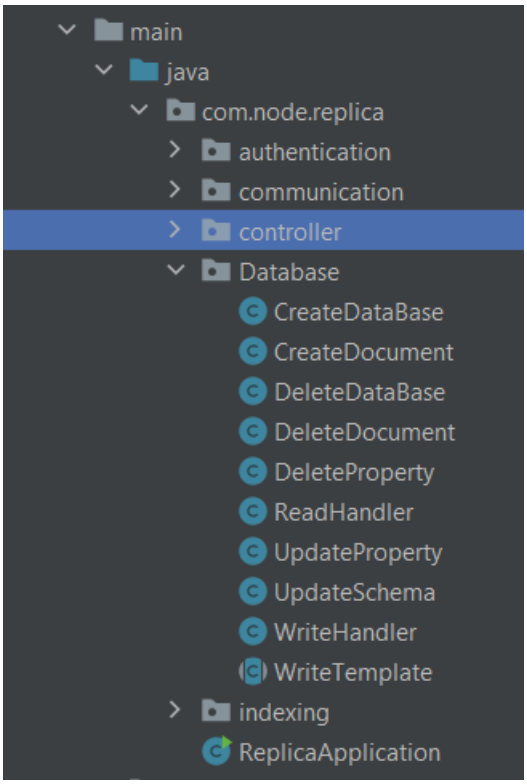

Figure 5: database package

This package is the most important package in the system, it is responsible for performing all queries.

Queries in the system are of two types, reading queries and writing queries. Reading queries are simple, they are looking for a piece of data on the disk and return it. Unlike writing queries; reading queries are performed on the same node assigned to the user, they have no affinity nodes, they are neither broadcasted nor expensive; especially with indexed properties.

Writing queries are more complicated; even though we assume they are rare, but they still have **unignorable** costs; many aspects must be considered when we are going to implement such queries; affinity authority, delegation, version matching, multithreading, indexing and broadcasting should be counted in such queries.

Although writing queries effect different areas in different ways, all of them have the same big image.

The following diagram shows writing query structure:

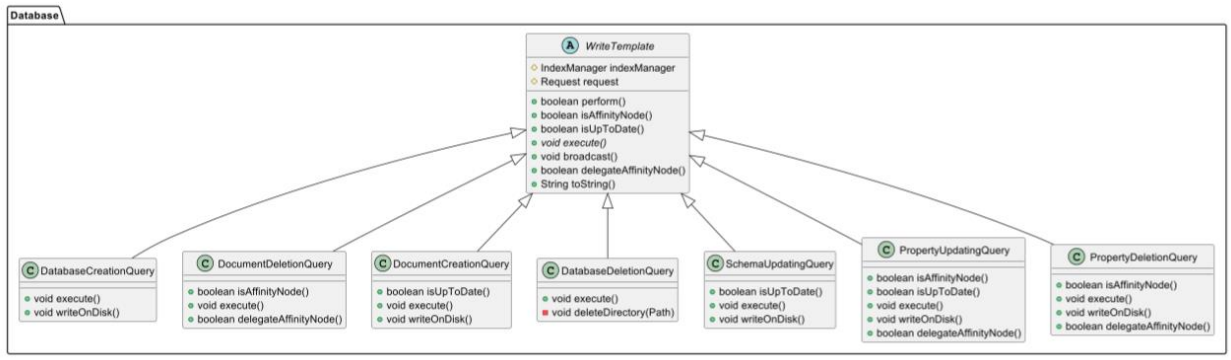

Figure 6: writing queries hierarchy

All writing queries classes are extended from WriteTemplate class, lets explain each class and its overridden methods:

1. **DatabaseCreationQuery**

   Before talking about database creation, lets remembers the file structure Writing queries are rare, creating a database is rarer, creating multiple databases by one user is rarest of all, regarding for all these possibilities, the affinity node for creating a database is the assigned node of the user on early registration; so we stick to the default implementation for **isAffinityNode** method, returning true directly.

   When a user sends a query to create a database, such query does not require to update any pre-existing data, each database is separated from other database, it just a process of creating a directory with some inner directories and files we discussed in in file structure; as we did with isAffinityNode, we did with **isUpToDate** method, stick to the default implementation.

Now, the only overridden method for this query is **execute** method, we create a JSON object for representing database schema, it has only two properties, affinity for storing assigned affinity for that document, and _id property for keep tracking for created documents under this database in future, then will write all these data on the disk, we separated write on disk logic in another method called **writeOnDisk** to satisfy **SRP** as much as possible.

Last step for this query, to keep all replicas consistent, is to broadcast this query to be executed in all others nodes in the cluster, this topic will be discussed in Broadcast section.

2. **DatabaseDeletionQuery**

   As we claimed so far in database creation query, delete a database is executed directly on the assigned node, **execute** method will delete the mentioned database directly and broadcast this query. As a database is represented as a directory, delete such type of files must be done recursively, inner content must be deleted first, **deleteDirectory** method will perform a recursive deletion for a database directory, simply by passing the database path to it in **execute** method.

3. **DocumentCreationQuery**

   This query can be more complicated compare to queries discussed so far, each document has three mandatory properties, _id, affinity and version; For _id property, it must be unique and sequential, the last value of _id is stored in **schema** file for each database, when a user try to create a document, **isUpTpDate** method in DocumentCreationQuery will send a request to the affinity node of schema file of targeted database, this request is handled by **SchemaUpdatingQuery**, a user will pass its current value of the _id in his assigned node, and if the send value of _id matches the value of _id in the affinity node of the schema, this _id will be incremented by one and request will return true, otherwise it will return false; and document creation will failed; **SchemaUpdaingQuery** is able to

handle one request at a time, (this will be discussed in **Multithreading and locks** section). If the send the response of updating schema returned true, the creation will be done and broadcasted, otherwise the API provided for the users will retry the request of creation.

4. **DocumentDeletionQuery**

   For this query, where a stored document already had its affinity node, if a user sends such request to a node, it will check if it's the affinity node for that document, it will execute, more formally, it will delete that object; otherwise, it will redirect the request to the affinity node, this redirection is smooth, the user will not be aware about. Once the deletion done by the affinity node, it will broadcast this query as always.

5. **PropertyDeletionQuery**
   All methods except for **isUpToDate** method are overridden for this query. After affinity node validation, execute this query will affect **version** property in the document that has the property, it will be incremented by one.
   Deleting property requires identifying a database name, a document ID, and a property name, the document that has identified property needs to be rewritten, this document will be stored in a JSON object, remove that property from the object and re-write the object in the same document. Finally, broadcast the query.

All other classes follow the same approach, any needed method is overridden, otherwise, we stick to the default implementation; they follow the following pattern to perform their work.

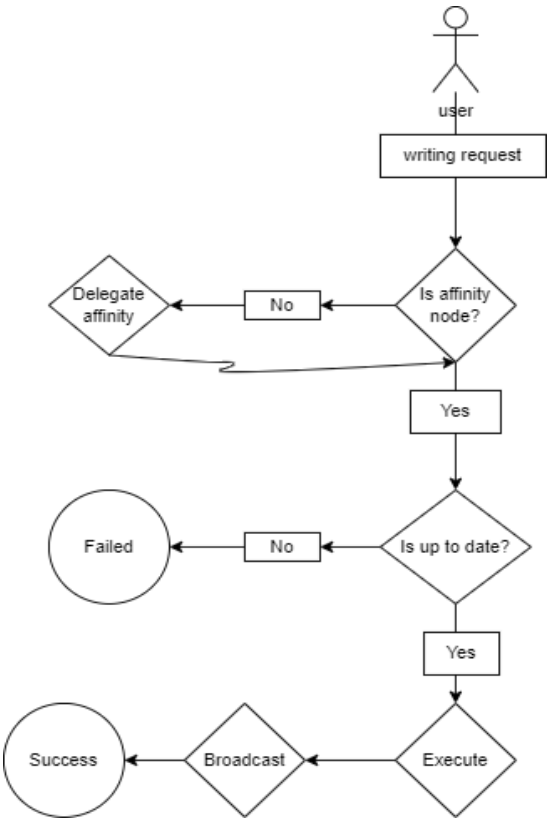

Figure 7: general writing algorithm

All of these queries effect on indexing. **Indexing** and how it is effected by writing queries will be discussed in Indexing section, also some of them are synchronized, and this will be discussed in **Multithreading and Locks** section.

For reading queries, **ReadHandler** class provides method to read all documents of a database, read a document has a specified ID, read documents have a specified property value.

Technology

For building this system, I used **spring boot**, spring boot is a Java-based technology makes built web based application easier, our system does not have to be web-based, but communication is essential.

Spring creates web applications that are able to handle communication process using **HTTP**, stands for Hypertext Transfer Protocol. Spring provides easy way for creating RESTful APIs, with some simple annotations like @RestController and @RequestMapping we can define API endpoints; these endpoints easily handle communication between any two sides.

Dependency injection is an important mechanism for building loosely coupled components, spring boot made utilizing such principle easier using annotations.

Spring boot eliminated the old configuration approaches, by enabling auto-configuration, all dependencies will be configured based on jar dependencies that have been added at the creation moment.

Indexing

Indexes are implement my project in indexing package, they are needed for reading, updated on writing, the following image shows indexing design:

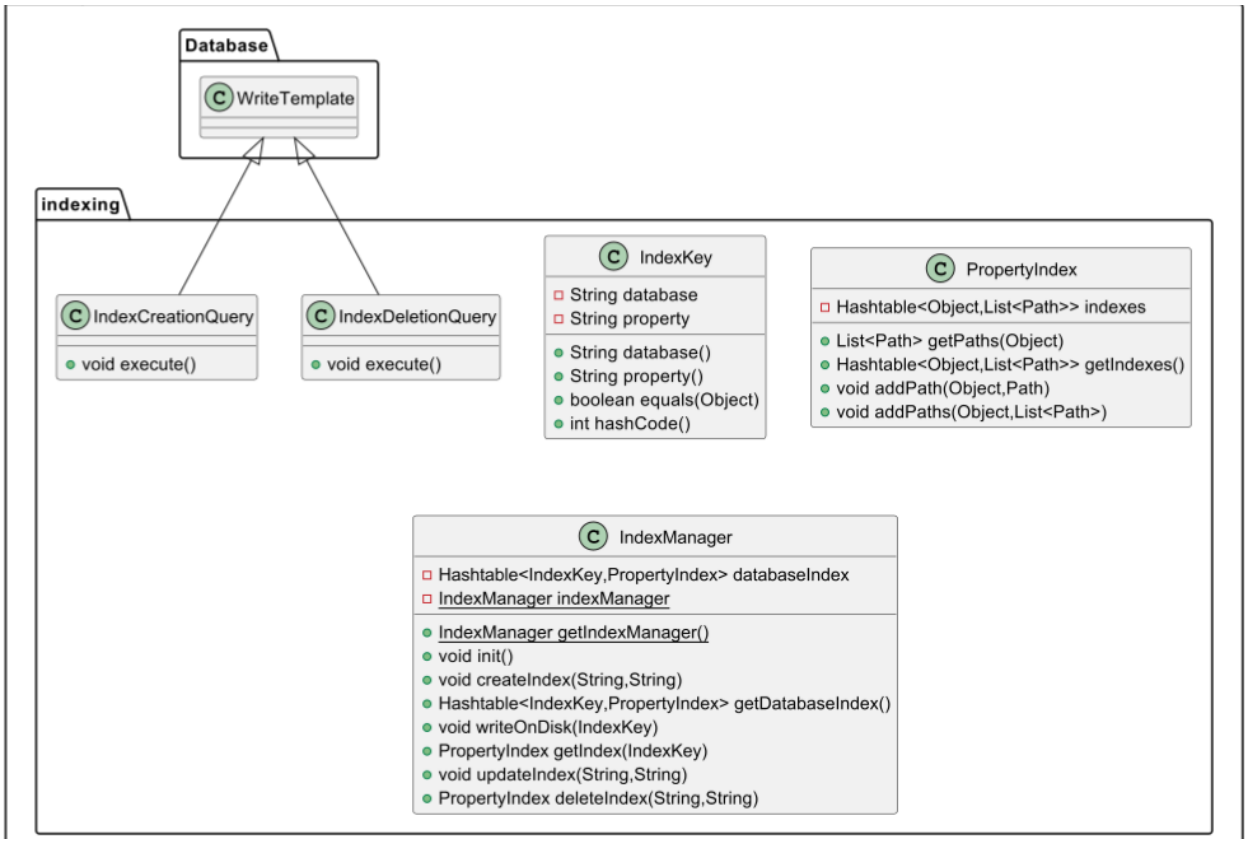

Figure 8: Indexing hierarchy

Indexkey is a class combine database name and property name to represent an index as a key for IndexManager.

PropertyIndex class represents the index itself, this class contains a Hashtable as the storage unit, the key is a value of a property, and value is the path of document contains that property value.

IndexManager is the main class of indexing, it contains a Hashtable named **databaseIndex** this will store all indexes at run-time, this class is singleton and eagerly initialized, because there will be one mandatory shared object.

IndexManager class contains **init** method, annotated as @PostConstruct to be called automatically once the project is initialized.

IndexManager class is injected in ReadHandler class to be used in reading queries, and in write template class to be updated after each writing query, and in WriteHandler to allow admins to create and delete indexes.

IndexCreationQuery and **IndexDeletionQuery** classes extend WriteTemplate class as they are considered writing queries.

Used Data Structures

Indexing

Indexing is a very important feature in any database system, its main goal is to make reading queries fast and efficient; instead of scanning large group of documents, you can jump directly to a related data.

Used data structure significantly effects index's performance. In Java, there are many built-in data structures that can be used to implement efficient indexing;

Hash Table and Hash Map has constant accessing complexity, by storing properties' values and their associated documents as key-value pairs, we can access desirable piece of data quickly.

I preferred Hash Table to Hash Map, for the following reasons:

1. Null resistance
   Unlike Hash Map, Hash Table does not allow neither key nor value to be null, this will ensure that indexed data is convenient.

2. Synchronized
   Hash Table is thread safe, all methods of Hash Table are synchronized, this reduce performance, but ensure consistency.

Reading queries

In reading requests' responses, I preferred lists to arrays, as Item 28 in Effective Java book identified. List are generics, generics are invariant and erased; they provide compile-time type safety unlike arrays.

Multithreading and Locks

Database systems or even any system with high traffic usage, must be multithreaded to serve the largest number of users simultaneously, and this is provided by REST controllers from spring, but such feature can harm the correctness and consistency of data, suppose that two users want to create a document on the same database at the same time, as you know, each document must have an ID, for indexing and fast accessing; when a document is created, we must check the last value of _id in schema file of targeted database, increment it by one, and return the last value to be the _id of the new document, if both execute this operation at the same time, this mean both documents will have the same _id value, this violates the logic of _id which must be unique, to solve this problem I used **Semaphore**.

In terms of operating systems, Semaphore is not more than a variable used to create critical sections, which are regions of code that must be executed by only one process at a time. By using semaphores, processes can coordinate access to shared resources, schema file in our case.

In Java, there is a class called Semaphore, it is used to limit the number of concurrent threads accessing a specific resource, this class offers methods to control the access of a critical regions, when an object of that class is created, the number of permitted threads must be determined, we permit only one thread to access a critical region at a time, to create a critical region, **tryAcquire** method is used to determine the start place for a critical section, this method returns true if the number of threads in the critical section is less than the number of permitted threads, and decrease the current number of unused thread by one until one of the threads releases the resource by calling **release** method.

I used this approach in two places, with document creation, where we need to modify schema file, and with property updating, to achieve optimistic locking idea using version property in each document, this property will be incremented by one on each update.

Data Consistency Issues

Writing queries affect the state of storage of the node that performed the write query.

I used a simple broadcasting approach to ensure data consistency in the cluster, simply, after performing any writing query, the node that performed this query will send HTTP request to all replicas in the system through the network that is created using **docker-compose** file. As we considered that the cluster has a fixed number of nodes, and each node runs on a known port, broadcast method in WriteTemplate class will use the inject request object to send the same written data to the Broadcast RESTful controller API, Broadcast controller is similar to query controller in terms of writing queries, but rather than performing the **WriteTemplate's** template method, it directly executes the query using execute method, because the main goal for broadcasting is to create a copy of a written data.

There are many common broadcasting algorithms; with variant complexities, broadcast and re-broadcast protocol, has O(N^2) complexity, even this algorithm extremely guaranteed that all nodes are received each message, but it has very bad performance, and cause high network congestion. One other is gossip protocol, it works in way each node first node or broadcast node send the message (writing query) for 3 random nodes in the cluster, and each other node do the same.

I used simple broadcasting algorithm in may project, the affinity node of any writing query will send a request to all nodes in the cluster to perform a query like performed query in the affinity node, may it is not the best algorithm, but it has the lowest possible complexity which is O(N), where N is the number of nodes in the cluster.

Node Hashing and Load Balancing

Load balancing must be considered in two aspects, users to nodes and documents to nodes, we assumed that the cluster have fixed number of nodes, 10 nodes in my case, these nodes expose ports from 8081 to 8090, each node knows it port and other nodes' ports, I implemented very simple straightforward load balancing approach.

For users to nodes; each user has ID, the last value of that ID is stored in a file in bootstrapping node, once a new user registers, the last value of the ID is incremented by one and assigned to that user, by calculating the modules of user ID over 10 (number of nodes), the result represents the node where the user should connect, if the modules is zero, the user will be assigned to node 10, on port 8090, using this equation:

First port = x

Last port = y

Number of nodes = y – x +1
Affinity node = ID % Number of nodes == 0 ? node y : node (ID % Number of nodes)

The same approach is used with documents, after assign ID for a new document, its modules over 10 will determine its affinity in using the same equation; version, affinity and _id properties are not updatable, _id and affinity are assigned for a document at creation moment, version is assigned at creation moment, and update on each update for a property of that document.

This approach is easy and straight forward, and we can ensure that users and documents are balanced in all nodes; and it is also dynamic, if the number of nodes is changed, there will be no need to change anything in code or in data, because the equation is adaptive; if all used ports by nodes are sequential.

Communication Protocol

Communication forms a big part of the project, more than one pair need to communicate with each other, bootstrapping node with users, bootstrapping node with replicas, user with its assigned node or replica and replicas with each other.

HTTP is the foundation of communication on the internet, HTTP is used communication protocol in the project, because of using spring boot, and RESTful APIs, which facilitate the communication using HTTP, it provides a set of verb called HTTP methods, such GET for retrieve data, POST to submit data, DELETE to delete data, etc.

Spring provides annotations to handle these request, for any HTTP verb, there is an annotation in the following form: @[HTTP REQUEST VERB]Mapping

When a user wants to access an endpoint, the user will use URL and HTTPURLCONNECTION objects, first object injected with the targeted URL in form of:

http://DOMAIN-Name:PORT/ENDPOINT

then this object will open a connection to initialize the second object, the second object will specify the parameters of the endpoint and used method for this endpoint, parameters are set in form of key-value pairs, where key is the key specified in **RequestHeader** annotation in endpoint's definition, any number of then call getInputStream method to send the request and retrieve the results if it exist.

Security Issues

Security is involved in authentication, authorization, and writing queries.

Authentication

When a new user wants to register in the system for the first time, he must send a registration request to the bootstrapping node, this node will create credentials for the user, and broadcast them; this credentials include three fields, ID, password and assigned node, all these fields are broadcast for all nodes in the cluster, and ID and password only is returned to the user and stored in a JSON file called info on the user machine. When the user wants to connect to the cluster, he sends a connection request, it reads ID and password from “info.json” file and send them to bootstrapping node, bootstrapping node will validate the send password by calculating its hashing value and compare it the hashed value that is associated with the send ID, if they are identical, the bootstrapping node will return the URL of assigned node to the user to be used with queries.

Authorization

Authorization is related to users’ roles in the system, each database has an admin specified at creation moment, it is only authorized person to create and delete indexes, and delete the database, simply, the admin of a database is the creator of the database.

Writing Injection

SQL injection is a code technique that may destroy the database, but our system is NoSQL using JSON documents for storing, I used **writing injection** name because such issues can happen in writing queries.

Rather than creating a structured language, I used RESTful APIs, it would be easier and safer to separate logic from data.

Each JSON document has a partial fixed structure enforce some properties to be exist for indexing, load balancing and consistency reason, a user is not allowed to remove these properties or change their values, validating writing queries’

parameters is a mandatory to ensure correctness and consistency, for example, when a user create a database, the code will create a document for that database with the name passed by the user, this name should not contain any character that is fixed part in any Path, like slashes, dots, colons and so on, a name of a database should be of upper letters, lower letters, and digit characters only.

For properties in a JOSN document, they also follow the same criteria, their names may be used for created index, which stored in JSON files to be permanent, using the property key; which must be unique.

Code Testing

I created a demo application provides API for communicating with the cluster, I provides methods for registration and connection to the database system, and methods for sending reading and writing queries.

When I user uses the database for the first time, he must register by sending a request to the bootstrapping node, the bootstrapping node will create a user with ID and password and assign a node for that user, storing the ID and hashed password and its assigned node and broadcast this new user information for all replicas in the cluster, and it will return the ID and password for the registered user, this credentials will be stored locally in the user machine in a JSON file called info, it will be used for connecting to the cluster and sending queries.

The last version of demo application has a menu for all allowed queries and gates for registration and connection.

Firstly, the user needs to register, once he is registered, he needs to connect, then either create a database or activate pre-existing database, and execute queries.

This demo shows how the system will act with one user, but this is not our expectation, our expectation is a large number of users, and they performing many queries at a time, so how I ensured that my implementation for synchronization is correct? How I test that no race condition in case that multiple users try to create a document on the same database at the same time, all these documents are created correctly?

Before I introduce last version of the demo, I tried to create large number of documents at the same time using the following code in the demo:

```
public class Cluster implements Runnable {
    @Override
    public void run() {
        try {
            createDocument();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

And create a large number of threads as the following:

```
public class Main {
    public static void main(String[] args) throws Exception {


        Cluster cluster = new Cluster();
        for(int i=0;i<100;i++) {
            Thread thread = new Thread(cluster);
            thread.start();
        }

      }

}
```

After running that code, I retrieve all documents in the database, I found each one of them has a unique _id, there was no pair of documents have the same _id, that does not mean all threads runs at the same time, but sure more than one at the same time, but one at a time is executing in the system, and other threads, or queries failed at the first time but they still retry until they are executed.

I used almost similar approach to test update a property, where if we send X number of updating property queries at the same time, the version property value after executing all queries must be incremented by X.

Clean Code Defending

As the book identified: “Making your code readable is as important as making it executable”, clean code practices care about many aspects in the code, names, functions, objects, data structures and even comments and formatting of the code.

In my code, I followed these principles and best practice to produce high quality code.

Names

Naming in my code follows some rules to be meaningful, all method names are short, descriptive, searchable, pronounceable, verb-prefixed, camel case, and identifying one job for each method.

Variables names are also follow these rules, all variables names are intention-revealing, free of typos, and informative.

Classes names are nouns, and use solution domain names (especially classes that implements some common design patterns), to be readable and understandable at least by developers.

All names in my project answer all big questions, they tell you why it exists, what they do, and how they are used.

Functions

Functions also have their rules for definition, first one, they should be as small as possible, and all functions in my project are not more than 20 lines, except for some functions, because they have some logic related for resources and exception handling, second one, functions must do one and only one thing, and they should do it well, this make them more readable and understandable, and to achieve **Single Responsibility Principle**.

Functions in my project do not have flag arguments, by dividing responsibilities, flag arguments are not needed.

Comments

Better than writing comments to explain your code, is to explain yourself in the code.
But sill comments are needed in some cases, for example, comments for explaining regular expression, domain logic, authority and highlighting.

On the other side, dumping the code with trivial comments can be considered as a noise.

I was keen to write simple self-explained code to minimize comments in my solution.

Formatting

This topic may look trivial to be mentioned, but it is really important; especially for communication between development teams.

Bad formatted code will work, dirty code will also do, but you will face troubles when time for testing, reviewing and maintenance comes.

There are some key points in terms of code formatting, we would like a source file to be like a newspaper article, if we imagine a class is an article, so articles start with headline, class name in our case, it should be short and descriptive. Then the first part should contain demonstration for the high level concepts and algorithms or dependencies in the class, like instance variables, which may be injected in the constructor as dependencies. The next parts of the article or the class are detailed specific parts, each one of them has single idea to process, which can be considered rest methods in a class.

Nearly all code is read left to right and top to bottom. Each line represents an expression or a clause, and each group of lines represents a complete thought. Those thoughts should be separated from each other with blank lines. This idea is automated in almost all modern IDEs, just be writing the closing curly brace of a class; a blank line will be inserted between each data or member field in that class.

Conceptual affinity is an important concept in terms of code formatting, which identifies that, field in a class should be physically as close as they are related.

Classes

As Functions, classes should be small, but with classes the measurement criteria are different, class size is measured using responsibilities not lines.

Cohesion measures how closely these elements work together to achieve a single, well-defined task or responsibility. By designing well-defined hierarchy, I achieved functional cohesion in my project, where all the methods or functions in a class work together to achieve a single common purpose; although number of classes are look a bit large, but this is the best approach to follow when class loss cohesion, you need to split them and make them smaller.

Effective Java Defending

Item 2: Consider a builder when faced with many constructor parameters

Request and **RequestBuilder** classes are designed using **Builder** pattern as it described in Effective Java book, instead of using telescoping constructor pattern and having multiple constructors representing all combinations of data memebers needed to create a request, builder pattern makes passing parameters to the object is more readable, also it helps to avoid sending flag values for optional parameters to satisfy compiler.

Item 5: Prefer dependency injection to hardwiring resources

My project has many classes that are depends on other classes, **Query** API class performs requests on **WriteHandler** and **ReadHandler** classes, **Broadcast** API classes also performs requests on **WriteHandler** class to manage consistency, **WriteHandler** class updates **IndexManager** status after each write operation to ensure index validity, and broadcast changes again, using **dependency injection** pattern will increase the extendibility in the project, by defining an interface reference in the dependent classes, we can inject any class implement that interface.

Spring boot, which I used to implement my project, provides an easy way for performing dependency injection by annotating references with **Autowired** annotation, and annotating creation block of the injected class (constructors or methods) with **Bean** annotation.

Item 8: Avoid finalizers and cleaner

Even though cleaners are less dangerous than finalizers, but still both of them is unpredictable, unnecessary and their execution is not guaranteed, reclaim reserved memory is done by garbage collector without any extra efforts from the developer.

Item 9: Prefer try-with-resources to try-finally

Try-with-resources can be used to instantiate all classes implement **AutoCloseable** interface, which consists of a single void-returning close method. Try-with-resources is shorter and more readable, especially in case of opening multiple resources at the same time, and it generates more useful exceptions that other exception handling techniques.

In my project, there are many methods that need to read and write on files, which are resources, I used try-with-resources in all of these methods.

Item 12: Always override toString

Default implementation for toString that is provided by **Object** class returns a string consists of the **class name** followed by an "at" sign (@) and the unsigned hexadecimal representation of the **hash code**, and this violates the general contract for toString, which says that the returned string should be "a concise but informative representation that is easy for a person to read." In all classes in my project, I overridden all toString method to retrieve informative data represents the status of the instantiated objects.

Item 15: Minimize the accessibility of classes and members

Applying this item improves the developed projects in many aspects.

It decouples the components that comprise a system, allowing them to be developed, tested, optimized, used, understood, and modified in isolation.

Also, information hiding increases software reuse because components that aren't tightly coupled often prove useful in other contexts besides the ones for which they were developed.

Java provides multiple techniques for information hiding, one of them is access modifiers.

I read about this rule in other books, but it described as principle of least privilege, I think the author of that book tries to say: Do not give someone anything that is not going to need.

Almost all data fields in my project are private, except for some fields that are needed in extended classes. Methods which represent the API of the classes are always public except if they are created to be helper method and to provide more separation of concerns in the same class.

Item 16: In public classes, use accessor methods, not public fields

Accessors are also known as getters, and mutators also known as setters, they are logically a special type of functions that are responsible for access fields of a class, rather than accessing fields directly, this improve data hiding and allow to validate assigned values for fields, and preventing editing mutable fields randomly.

Item 17: Minimize mutability

Joshua Bloch mentioned five rules to make a class immutable, I use them carefully to provide less mutability in my project, for example, for **WriteTemplate** class which provides the big image for any write query, I define broadcast method as final method to prevent extended class to override it, broadcasting process is the same for all writing query except for the broadcasted data that is combined in a hash table to be provided for requests in form of key and value. Also as I mentioned before, almost of fields are private and not all of them have mutators (setters).

Item 23: Prefer class hierarchies to tagged classes

As write queries include create, update and delete, it would be bad choice to use flags or some magic words or number and switch statements to specify which write query to execute, it would be better to use meaningful hierarchy, where each query is represented as a separate class, and override proper methods to achieve its task, this approach will add more decoupling, and the concerns will be extremely separated.

I use the same approach, I defined an abstract class called **WriteHandler**, containing an abstract method for each method in the tagged class whose behavior depends on the tag value(**isAffinity**, **isUpTpDate**, **delegateToAffinity**

and execute methods), and I implemented the method that its behavior does not depend on the tag value inside that class, which is broadcast method, all of these methods are called in perform method which was implemented in **WriteHandler** class to gain the privileges of **Template** design pattern.

I used the same approach for data fields, Request and IndexManager.

Item 25: Limit source files to a single top-level class

Each source file has only one class, to avoid any compilation errors producing by multiple definitions of the same class by preventing re-using the same name for multiple classes, and this is applied for all source files in my project.

SOLID Principles Defending

Single Responsibility Principle

Each component in any software should have only one thing to do; this will make the component maintainable, testable and readable. In my project, each method does what its name is and only its name is, for this principle I had to define some private methods, and smaller methods, and descriptively named methods to minimize the job of each method.

Open Closed Principle

Software components should be closed for modifications, but open for extension.

I applied this principle in the logic that connect Request class with WriteTemplate class, data needed for methods in WriteTemplate class are not passed as parameters, they are injected in Request objects, in addition to clean code terms where the best number of parameters for a method is zero, if a developer want to override a method of WriteTemplate methods, there is no need to modify the methods' signatures for adding more or parameters, the developer just needs to inject these parameters in injected Request's object, and accessing them from methods in request object that is inherited from WriteTemplate and use them anywhere.

Liskuv Substitution Principle

Objects should be replaceable with their subtypes without affecting the correctness of the program.

In all writing handlers that are extended WriteTemplate, any one of these subtypes are replaceable with the super class, all of them inherited the default implementation for methods in the super class and able to override it to apply their own logic, so the super class is replaceable with all its children.

Interface Segregation Principle

No Client should be forced to depend on methods it doesn't use.

I used Removable interface for all classes that are able to remove any local resource, this interface has only one method's signature called **remove,** and receive a path for a resource to be deleted; to avoid putting such methods in WriteTemplate class, where some of its subtypes do not remove any resource.

Dependency Inversion Principle

High level modules shouldn't depend on low level modules, both should depend on abstractions and Abstractions shouldn't depend on details, details should depend on abstractions.

Dependency inversion can be achieved by following these practices:

1. Dependency Injection:
   Rather than hardwiring of dependencies, they can be created outside the classes and injected in constructors or mutators; this allows for the substitution of different implementations of the same interface or abstraction without modifying the class that uses them.
2. Use of Interfaces or Abstract Classes:
   When injected dependencies implement the same interface or extends concrete types of abstract classes, replacing them will be easier, because they agreed on the same contract.
3. Inversion of Control Containers:
   Spring automates the process of injecting dependencies using some annotations. It manages the creation and injection of objects, promoting loose coupling between components; so using spring speeded up the development process, by annotating dependency as a **Bean** and annotating its reference as **Autowired**, spring will inject the dependency to the dependent after Bean is created in run-time

Design Patterns

Template Method Pattern

As mentioned literally in Head First Deign Pattern: “The Template Method defines the steps of an algorithm and allows subclasses to provide the implementation for one or more steps”. And this is completely matches our case in writing queries, where there is a step for checking affinity, delegation another node for some queries, checking version matching, then execute the query and broadcast that query for consistency guarantees, but each query has its affinity definition, version matching, and different place for writing and effecting the disk storage. I designed an abstract class called WriteHandler, which has a Template Method called perform that implements the following algorithm:

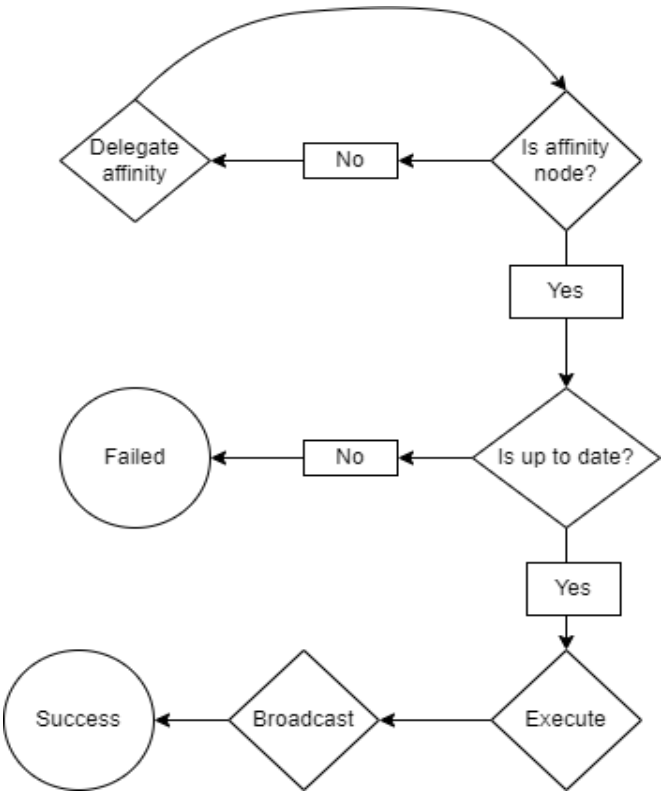

Figure 9: Template method perform algorithm

Singleton pattern

IndexManager class designed to be singleton, on each replica or node, there will be only one manager for indexing management, because this class carry all indexes in the replica, its size could be large, and because of the same index

should be read in **ReadHandler** and updated after each writing query, make this class as singleton helps to ensure that read and write done on the same object.

Builder Pattern

Builder pattern is a creational pattern; it aims to separate the process of constructing a complex object from its representation.

It is one the best alternatives for **Telescoping Constructor** pattern, when a class has number of fields that some of them are optional.

Telescoping Constructor pattern can provide different constructor allows to inject the needed parameters for fields, but you should create number of constructors that is almost equal to the number of fields combinations, **Builder** pattern solve this dilemma by creating mutators for all fields, but with a small different from traditional mutators.
Mutators in Builder patterns return a reference points for all these fields in a wrapper object that will be injected to a concrete type of the builder class in build method.

I used this pattern with **Request** and **RequestBuilder** classes, which are abstract class and concrete class, respectively. I used the approach described in **Effective Java Book – Item2**, both of class has its own builder internally, abstract class has an abstract builder, concrete class has a concrete builder, also, as the concrete class extended the abstract class, the builder of the concrete class extended the builder of the abstract class. Builder of the abstract class is static to be accessible in concrete class, and builder inside concrete class is also static to make it an enclosing class to eliminate the needing of outer class object instantiation.

Almost all fields in these class are logically optional, not all requests need to be broadcasted, and not all need to be redirected, not all have the same list of parameters, not all use the same request method, it can be eliminated because if it is **GET** method, because it is the default method.

DevOps Practices

Clustered system has multiple identical nodes, operating such system in real-environment needs multiple physical machines or servers, where each machine represents an isolated node, such infrastructure costs a lot.

For simulation purposes, I used **Docker** software that can ship each node and run it as isolated virtual machine, by creating a **Dockerfile** for each part of the cluster, this Dockerfile will generates an image, which will be used to run the system as a container, container is a running image, encapsulates all dependencies of a software, and operates this software and make operable on any machine without worrying about missing dependencies or compatibility issues.
I used **Maven** build tool to generate a **jar** file (stands for Java Archive), which is a file format based on the popular ZIP file format and is used for aggregating many files into one to ship the project, and copied that jar file into a specific directory inside the container to use it to run the software.

I also used **Docker-compose** file, which is a tool for defining and running multiple containers simultaneously, defining network for containers communications, volumes for permanent storage, exposed ports for each container, environments variables and running order for containers.

In Docker-compose, each container is represented as a service, that has all necessary information to start this service.

Using containerization and dockerization techniques, running such micro-service system is too easy, and can be done using a few commands.