Report on

Cost Analysis of Higher Education: A Distributed System using gRPC and NoSQL MongoDB Database

Submitted to
Prof. Yan Liu

COEN 6731
Distributed Software Systems

Department of Electrical and Computer Engineering
Gina Cody School of Engineering and Computer Science

By

Pranav Kumar Jha

3rd April, 2023

Introduction

The main objective of this project is to create a data operation architecture that provides efficient querying of a MongoDB database containing educational expense data. The system consists of a gRPC client that connects to a gRPC gateway, which in turn connects to five different DAO QueryTypes, each representing a specific query to the data stored in the EduCostStat collection.

The project is divided into two main tasks. In the first task, a MongoDB database is set up and populated with data samples, and Java DAO classes are developed to represent different queries to the data stored in the EduCostStat collection. The different queries include querying the cost, top 5 most expensive and economic states, top 5 states with the highest growth rate of overall expense, and aggregating region's average overall expense. The queries are saved as documents in specific collections.

In the second task, a Protocol Buff definition file is created to represent the request, response, and service for each query defined in Task 1. The Java program is then developed for each service defined in Task 2.1 as gRPC services. Finally, a gRPC client and server (or gateway) code is developed to communicate as RPC calls for the five queries defined in Task 1.

Data Operation Architecture

The diagram below represents the Data Operation Architecture with Remote Procedure Call (RPC) Communication based on the requirements given.

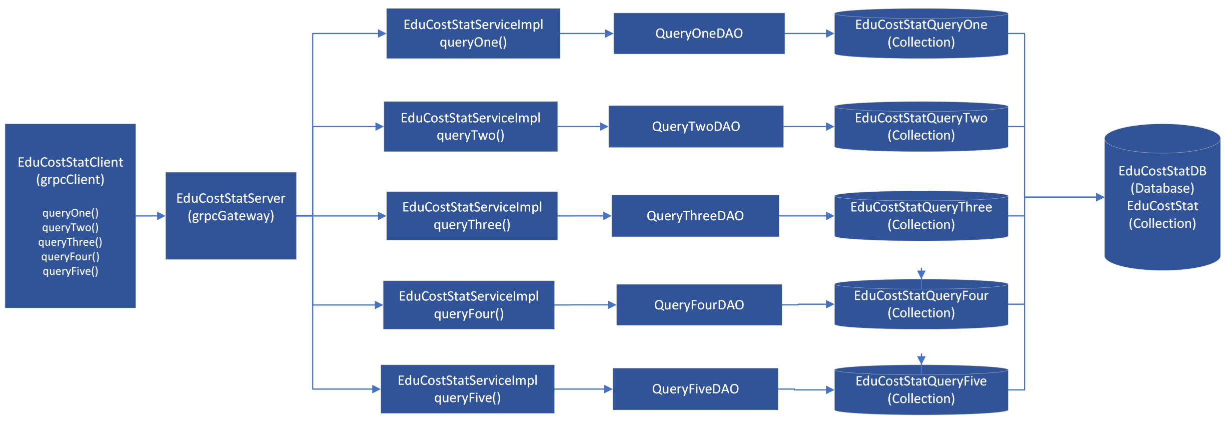

EduCostStatClient (grpcClient)
queryOne()
queryTwo()
queryThree()
queryFour()
queryFive()
EduCostStatServer (grpcGateway)
EduCostStatServiceImpl queryOne()
EduCostStatServiceImpl queryTwo()
EduCostStatServiceImpl queryThree()
EduCostStatServiceImpl queryFour()
EduCostStatServiceImpl queryFive()
QueryOneDAO
QueryTwoDAO
QueryThreeDAO
QueryFourDAO
QueryFiveDAO
EduCostStatQueryOne (Collection)
EduCostStatQueryTwo (Collection)
EduCostStatQueryThree (Collection)
EduCostStatQueryFour (Collection)
EduCostStatQueryFive (Collection)
EduCostStatDB (Database) EduCostStat (Collection)

Fig. 1: Data Operation Architecture with Remote Procedure Call (RPC) Communication

The architecture diagram shown in above figure depicts the Data Operation Architecture with Remote Procedure Call (RPC) Communication. It comprises four main components:

1. **EduCostStatClient (gRPC Client):** This gRPC client is a program that sends requests to the gRPC server (or gateway) and receives responses. It initiates the request, waits for the response and processes it.

   The EduCostStatClient class has a constructor that takes the host and port number of the server and initializes the blocking stub and asynchronous stub for making RPC calls.

   There are five methods that correspond to the five queries that can be made to the server:

   queryOne: Takes in the year, state, type, length, and expense and sends a QueryOneRequest to the server. The server returns a QueryOneResponse containing the query ID and total expense.

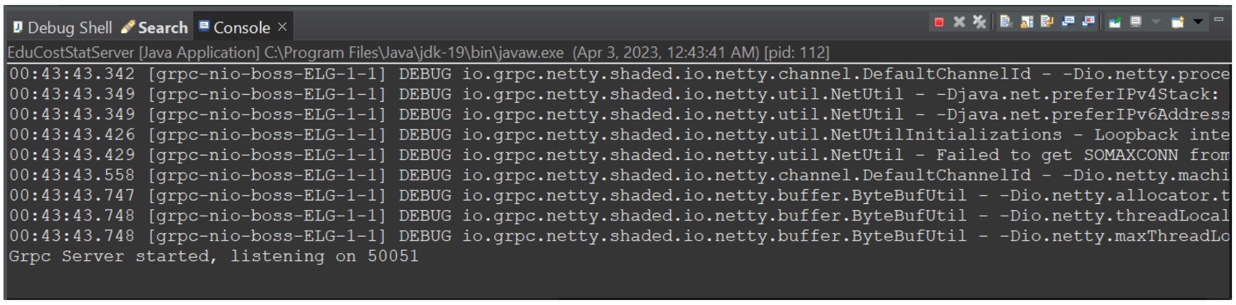
   queryTwo: Takes in the year, type, and length and sends a QueryTwoRequest to the server. The server returns a QueryTwoResponse containing a list of expensive states.

   queryThree: Takes in the year, type, and length and sends a QueryThreeRequest to the server. The server returns a QueryThreeResponse containing a list of economic states.

   queryFour: Takes in the base year, years ago, type, and length and sends a QueryFourRequest to the server. The server returns a QueryFourResponse containing a list of state growth rates.

   queryFive: Takes in the year, type, and length and sends a QueryFiveRequest to the server. The server streams a QueryFiveResponse containing a list of region costs back to the client using a StreamObserver.

2. **EduCostStat Server (Gateway):** The EduCostStat server acts as a gateway between the gRPC client and the different query types. It receives the RPC requests from the gRPC client, sends them to the corresponding query DAO, and returns the response to the gRPC client. It acts as an intermediary between the client and the different query types, abstracting away the details of the underlying database and query implementation. The server starting at port 50051 as shown below:



```
EduCostStatServer [Java Application] C:\Program Files\Java\jdk-19\bin\javaw.exe  (Apr 3, 2023, 12:43:41 AM) [pid: 112]
00:43:43.342 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.channel.DefaultChannelId - -Dio.netty.proce
00:43:43.349 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.util.NetUtil - -Djava.net.preferIPv4Stack:
00:43:43.349 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.util.NetUtil - -Djava.net.preferIPv6Address
00:43:43.426 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.util.NetUtilInitializations - Loopback inte
00:43:43.429 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.util.NetUtil - Failed to get SOMAXCONN from
00:43:43.558 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.channel.DefaultChannelId - -Dio.netty.machi
00:43:43.747 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.buffer.ByteBufUtil - -Dio.netty.allocator.t
00:43:43.748 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.buffer.ByteBufUtil - -Dio.netty.threadLocal
00:43:43.748 [grpc-nio-boss-ELG-1-1] DEBUG io.grpc.netty.shaded.io.netty.buffer.ByteBufUtil - -Dio.netty.maxThreadLo
Grpc Server started, listening on 50051
```

Fig. 2: gRPC server

The gRPC server listens on a specific port for incoming requests from gRPC clients. It has a nested class called EduCostStatServiceImpl that extends the generated EduCostStatServiceGrpc.EduCostStatServiceImplBase class. This nested class defines the implementation of the RPC methods declared in the EduCostStatService.proto file.

The EduCostStatServer class constructor initializes the gRPC server using the specified port and the implementation defined in EduCostStatServiceImpl class. It also provides methods to start and stop the server, and a blockUntilShutdown() method that blocks the current thread until the server shuts down.

3. **EduCostStatService Impl** The EduCostStatServiceImpl class defines the implementation of the five query methods defined in EduCostStatService.proto. Each method invokes the corresponding data access object (DAO) class defined in Task 1 and builds the response message using the query results obtained from the DAO methods. The response message is sent back to the client through the responseObserver parameter passed to the method.

4. **Query DAOs:** The query DAOs represent different queries to the data stored in the EduCostStat MongoDB collection. There are five query DAOs in total, queryOneDAO, QueryTwoDAO,...,QueryFiveDAO, each corresponding to one of the five different queries specified in the project requirements. Each query DAO receives the request from the EduCostStat server, executes the corresponding query on the MongoDB collection, and returns the response back to the server.

5. **EduCostStatQuery Collection** The EduCostStatQueryOne collection stores documents that represent a specific query for the cost of education given a specific year, state, type,

length, and expense. The documents include information about the query input parameters and the total cost of education based on these parameters.

The EduCostStatQueryTwo collection stores documents that represent a specific query for the top 5 most expensive states based on overall expense for a given year, type, and length. The documents include information about the year, type, and length used in the query, as well as the name of the state and its corresponding overall expense.

The EduCostStatQueryThree collection stores documents that represent a specific query for the top 5 most economic states based on overall expense for a given year, type, and length. The documents include information about the year, type, and length used in the query, as well as the name of the state and its corresponding overall expense.

The EduCostStatQueryFour collection stores documents that represent a specific query for the top 5 states with the highest growth rate of overall expense based on a range of past years, one year, three years, and five years, type, and length. The documents include information about the query input parameters, the name of the state, and its corresponding growth rate.

The EduCostStatQueryFive collection stores documents that represent a specific query for the aggregated region's average overall expense for a given year, type, and length. The documents include information about the year, type, and length used in the query, as well as the name of the region and its corresponding average overall expense.

6. **EduCostStatDB MongoDB Database:** The EduCostStat MongoDB database is where the actual data is stored. The MongoDB collection named ”EduCostStat” holds the data samples for different years, states, types, lengths, and expenses. The query DAOs interact with the MongoDB database to execute the different queries and return the corresponding results.

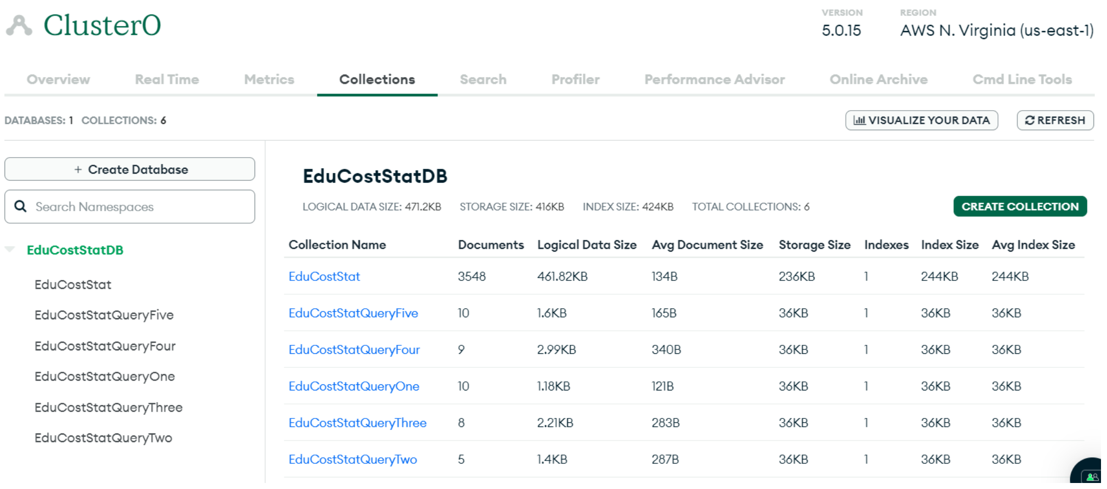
1 Task 1: MongoDB Data Storage and Operation

Task 1 involves setting up a MongoDB database and creating a collection named EduCostStat to store the data. The collection is populated by reading data samples from a file and inserting them into the collection based on specific year, state, type, length, and expense values. Additionally, Task 1 also requires the development of Java DAO classes that represent different queries to the data stored in the EduCostStat collection [1].

1.1 Task 1.1 Creating the EduCostStat Collection

Data source and format: The data is downloaded in the csv format from kaggle using the link given in the assignment description. It is saved in the ”src/main/resources/avg-cost-undergrad-by-states.csv”

MongoDB setup and connection: MongoDB setup and connection has been done in a file named ”MongoDBUtil” under the package ”org.concordia.MongoDB”. As shown in the figure below, We have Created a MongoDB online Cluster to build a Database named ”EduCostStatDB” and a collection named ”EduCostStat”.



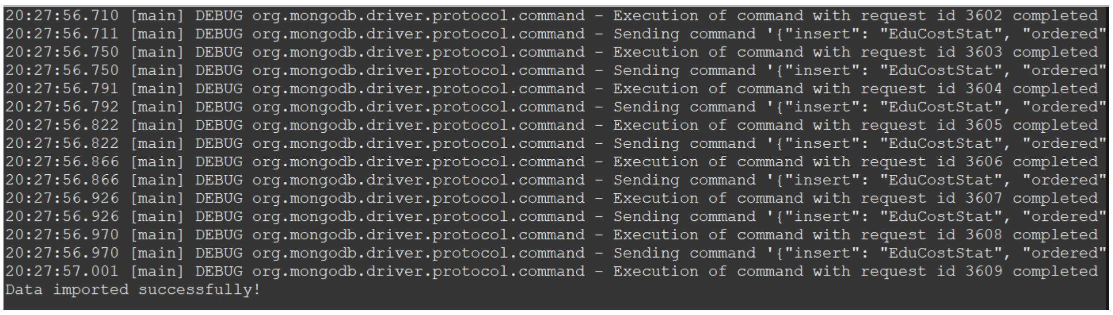
EduCostStatDB

LOGICAL DATA SIZE: 471.2KB STORAGE SIZE: 416KB INDEX SIZE: 424KB TOTAL COLLECTIONS: 6

Collection Name	Documents	Logical Data Size	Avg Document Size	Storage Size	Indexes	Index Size	Avg Index Size
EduCostStat	3548	461.82KB	134B	236KB	1	244KB	244KB
EduCostStatQueryFive	10	1.6KB	165B	36KB	1	36KB	36KB
EduCostStatQueryFour	9	2.99KB	340B	36KB	1	36KB	36KB
EduCostStatQueryOne	10	1.18KB	121B	36KB	1	36KB	36KB
EduCostStatQueryThree	8	2.21KB	283B	36KB	1	36KB	36KB
EduCostStatQueryTwo	5	1.4KB	287B	36KB	1	36KB	36KB

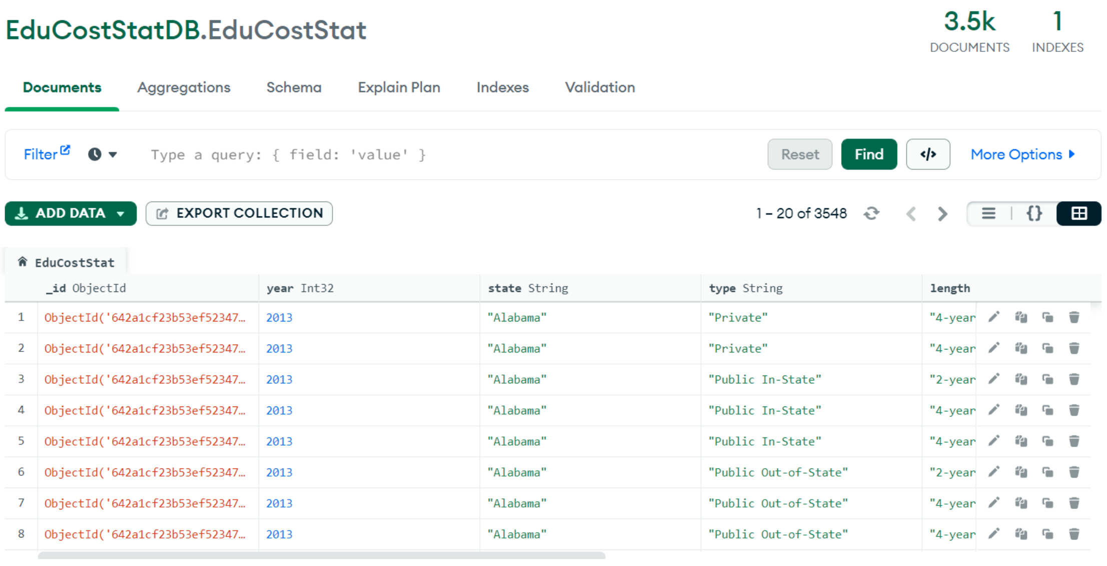
Figure 1.1: MongoDB online cluster, database and collection

Reading data from file and inserting into collection: We are importing data to the collection using our MongoDBUtil.java class as shown in figure below.



```
20:27:56.710 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3602 completed
20:27:56.711 [main] DEBUG org.mongodb.driver.protocol.command - Sending command '{"insert": "EduCostStat", "ordered"
20:27:56.750 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3603 completed
20:27:56.750 [main] DEBUG org.mongodb.driver.protocol.command - Sending command '{"insert": "EduCostStat", "ordered"
20:27:56.791 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3604 completed
20:27:56.792 [main] DEBUG org.mongodb.driver.protocol.command - Sending command '{"insert": "EduCostStat", "ordered"
20:27:56.822 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3605 completed
20:27:56.822 [main] DEBUG org.mongodb.driver.protocol.command - Sending command '{"insert": "EduCostStat", "ordered"
20:27:56.866 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3606 completed
20:27:56.866 [main] DEBUG org.mongodb.driver.protocol.command - Sending command '{"insert": "EduCostStat", "ordered"
20:27:56.926 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3607 completed
20:27:56.926 [main] DEBUG org.mongodb.driver.protocol.command - Sending command '{"insert": "EduCostStat", "ordered"
20:27:56.970 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3608 completed
20:27:56.970 [main] DEBUG org.mongodb.driver.protocol.command - Sending command '{"insert": "EduCostStat", "ordered"
20:27:57.001 [main] DEBUG org.mongodb.driver.protocol.command - Execution of command with request id 3609 completed
Data imported successfully!
```

Figure 1.2: Data importing to the collection



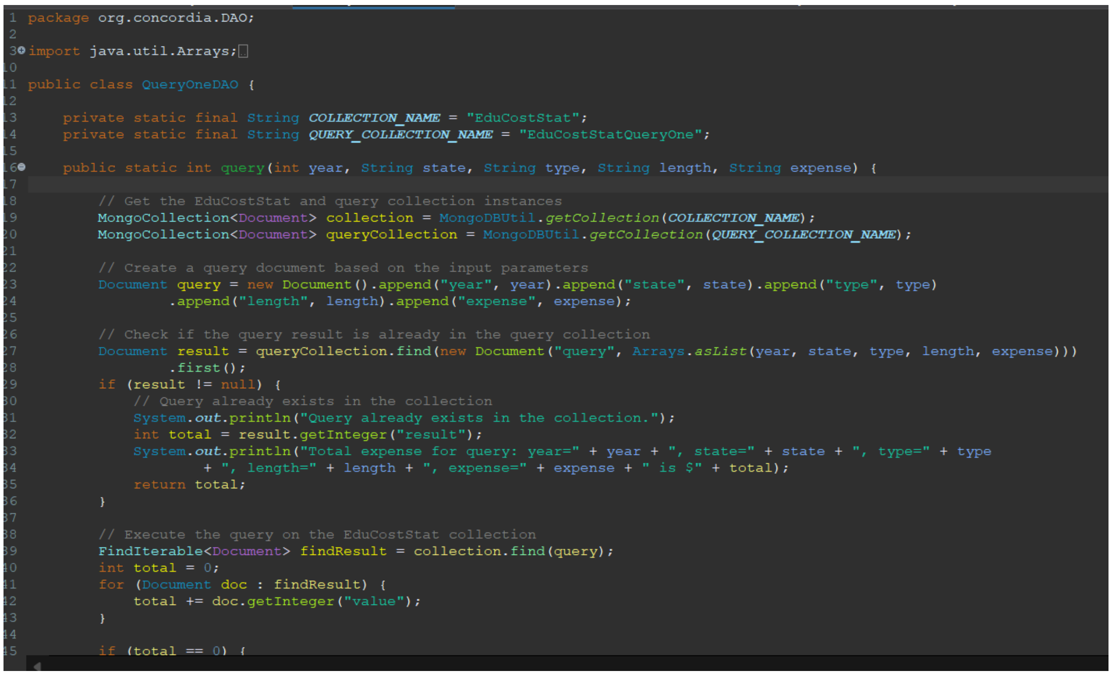
	_id ObjectId	year Int32	state String	type String	length
1	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Private"	"4-year
2	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Private"	"4-year
3	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Public In-State"	"2-year
4	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Public In-State"	"4-year
5	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Public In-State"	"4-year
6	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Public Out-of-State"	"2-year
7	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Public Out-of-State"	"4-year
8	ObjectId('642a1cf23b53ef52347…	2013	"Alabama"	"Public Out-of-State"	"4-year

Figure 1.3: Data imported to the collection

1.2 Task 1.2: Developing Data Access Objects

The DAO files for the queries 1,2,...,5 are vdescribed below:

1. **QueryOneDAO:** The QueryOneDAO class contains the implementation of the first query that retrieves the total expenses for a given year, state, type, length, and expense. The method query takes in these input parameters and executes a query on the MongoDB collection EduCostStat to retrieve documents matching the query criteria. It then calculates the total expenses by summing up the value field from the retrieved documents.

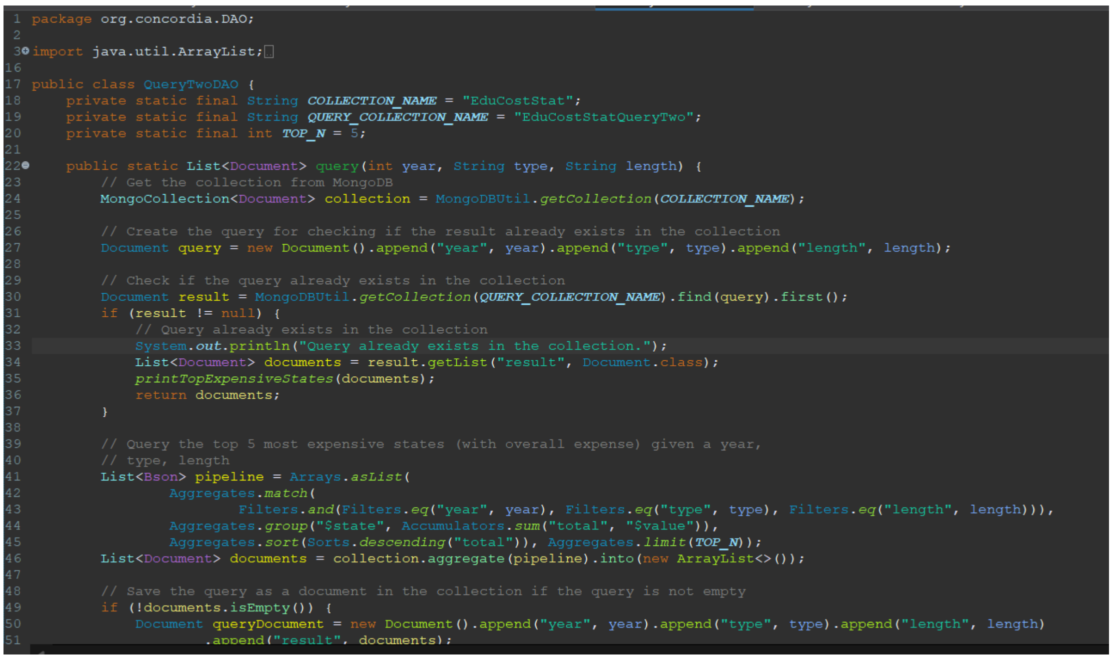


```java
package org.concordia.DAO;

import java.util.Arrays;

public class QueryOneDAO {

    private static final String COLLECTION_NAME = "EduCostStat";
    private static final String QUERY_COLLECTION_NAME = "EduCostStatQueryOne";

    public static int query(int year, String state, String type, String length, String expense) {

        // Get the EduCostStat and query collection instances
        MongoCollection<Document> collection = MongoDBUtil.getCollection(COLLECTION_NAME);
        MongoCollection<Document> queryCollection = MongoDBUtil.getCollection(QUERY_COLLECTION_NAME);

        // Create a query document based on the input parameters
        Document query = new Document().append("year", year).append("state", state).append("type", type)
                .append("length", length).append("expense", expense);

        // Check if the query result is already in the query collection
        Document result = queryCollection.find(new Document("query", Arrays.asList(year, state, type, length, expense)))
                .first();
        if (result != null) {
            // Query already exists in the collection
            System.out.println("Query already exists in the collection.");
            int total = result.getInteger("result");
            System.out.println("Total expense for query: year=" + year + ", state=" + state + ", type=" + type
                    + ", length=" + length + ", expense=" + expense + " is $" + total);
            return total;
        }

        // Execute the query on the EduCostStat collection
        FindIterable<Document> findResult = collection.find(query);
        int total = 0;
        for (Document doc : findResult) {
            total += doc.getInteger("value");
        }

        if (total == 0) {
```

Figure 1.4: QueryOneDAO

The method also checks if the query result already exists in the EduCostStatQueryOne collection by searching for a matching query field. If the result exists, it returns the total expense, and if not, it saves the query result to the collection. The MongoDBUtil class is used to establish a connection to the MongoDB server and retrieve the required collection instances. The query file is named ”QueryOneDAO”.

2. **QueryTwoDAO:** The QueryTwoDAO class contains a static method query that performs the second query requirement, which is to retrieve the top 5 most expensive states (with overall expense) given a specific year, type, and length.



```java
package org.concordia.DAO;

import java.util.ArrayList;

public class QueryTwoDAO {
    private static final String COLLECTION_NAME = "EduCostStat";
    private static final String QUERY_COLLECTION_NAME = "EduCostStatQueryTwo";
    private static final int TOP_N = 5;

    public static List<Document> query(int year, String type, String length) {
        // Get the collection from MongoDB
        MongoCollection<Document> collection = MongoDBUtil.getCollection(COLLECTION_NAME);

        // Create the query for checking if the result already exists in the collection
        Document query = new Document().append("year", year).append("type", type).append("length", length);

        // Check if the query already exists in the collection
        Document result = MongoDBUtil.getCollection(QUERY_COLLECTION_NAME).find(query).first();
        if (result != null) {
            // Query already exists in the collection
            System.out.println("Query already exists in the collection.");
            List<Document> documents = result.getList("result", Document.class);
            printTopExpensiveStates(documents);
            return documents;
        }

        // Query the top 5 most expensive states (with overall expense) given a year,
        // type, length
        List<Bson> pipeline = Arrays.asList(
                Aggregates.match(
                        Filters.and(Filters.eq("year", year), Filters.eq("type", type), Filters.eq("length", length))),
                Aggregates.group("$state", Accumulators.sum("total", "$value")),
                Aggregates.sort(Sorts.descending("total")), Aggregates.limit(TOP_N));
        List<Document> documents = collection.aggregate(pipeline).into(new ArrayList<>());

        // Save the query as a document in the collection if the query is not empty
        if (!documents.isEmpty()) {
            Document queryDocument = new Document().append("year", year).append("type", type).append("length", length)
                    .append("result", documents);
```

Figure 1.5: QueryTwoDAO

The method first checks if the query result already exists in the EduCostStatQueryTwo collection by creating a query document and using it to find a document in the collection. If a document is found, the method retrieves the query result from the result field of the document and prints the top 5 most expensive states in descending order.

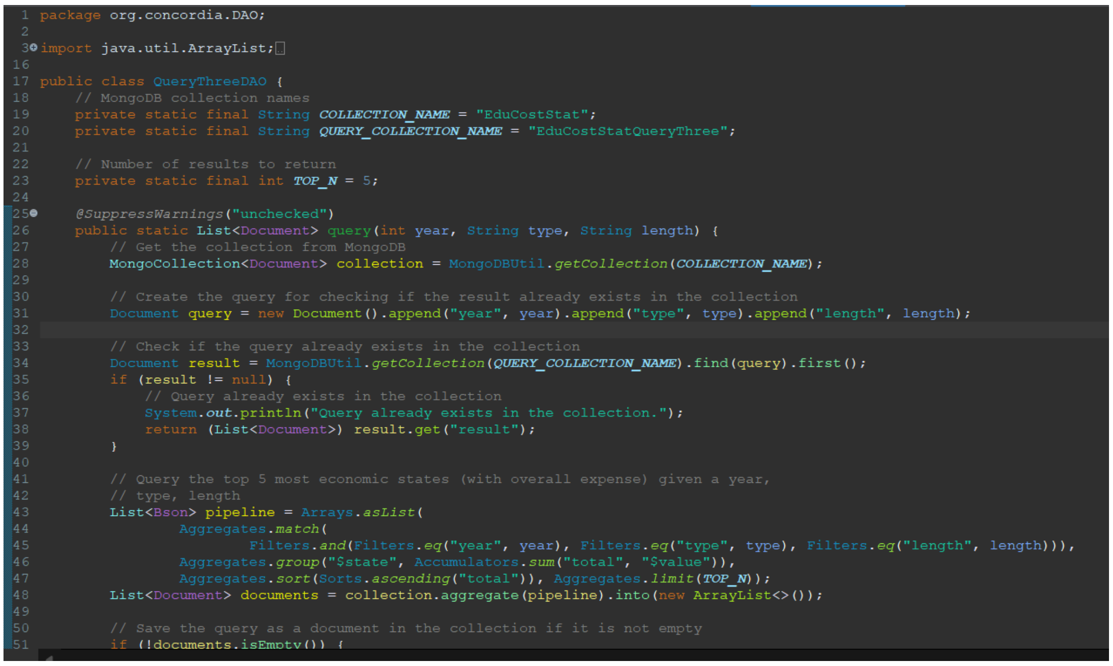
If the query result is not found in the collection, the method creates an aggregation pipeline that performs the necessary operations to retrieve the top 5 most expensive states. The pipeline first matches the documents that match the given year, type, and length criteria. Then it groups the documents by state and calculates the sum of their values. After that, the pipeline sorts the resulting documents in descending order based on their total expenses and limits the result to the top 5 documents.

The method then saves the query result as a document in the EduCostStatQueryTwo collection if the query result is not empty. Finally, the method prints the top 5 most expensive states in descending order, along with their total expenses.

The class also contains a private method printTopExpensiveStates that prints the top N most expensive states with their total expenses.

3. **QueryThreeDAO:** This is the implementation of QueryThreeDAO, which is responsible

for querying the top 5 most economic states with overall expenses given a year, type, and length. The class uses MongoDB to execute the query and save the results in a collection named EduCostStatQueryThree.

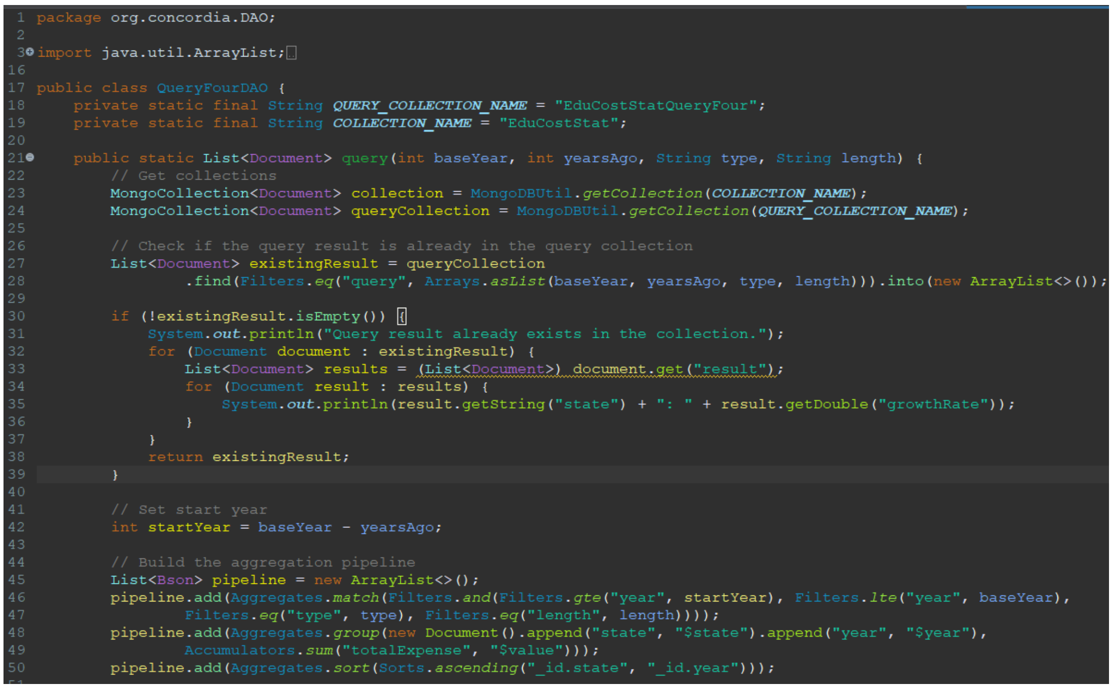


```java
package org.concordia.DAO;

import java.util.ArrayList;

public class QueryThreeDAO {
    // MongoDB collection names
    private static final String COLLECTION_NAME = "EduCostStat";
    private static final String QUERY_COLLECTION_NAME = "EduCostStatQueryThree";

    // Number of results to return
    private static final int TOP_N = 5;

    @SuppressWarnings("unchecked")
    public static List<Document> query(int year, String type, String length) {
        // Get the collection from MongoDB
        MongoCollection<Document> collection = MongoDBUtil.getCollection(COLLECTION_NAME);

        // Create the query for checking if the result already exists in the collection
        Document query = new Document().append("year", year).append("type", type).append("length", length);

        // Check if the query already exists in the collection
        Document result = MongoDBUtil.getCollection(QUERY_COLLECTION_NAME).find(query).first();
        if (result != null) {
            // Query already exists in the collection
            System.out.println("Query already exists in the collection.");
            return (List<Document>) result.get("result");
        }

        // Query the top 5 most economic states (with overall expense) given a year,
        // type, length
        List<Bson> pipeline = Arrays.asList(
                Aggregates.match(
                        Filters.and(Filters.eq("year", year), Filters.eq("type", type), Filters.eq("length", length))),
                Aggregates.group("$state", Accumulators.sum("total", "$value")),
                Aggregates.sort(Sorts.ascending("total")), Aggregates.limit(TOP_N));
        List<Document> documents = collection.aggregate(pipeline).into(new ArrayList<>());

        // Save the query as a document in the collection if it is not empty
        if (!documents.isEmpty()) {
```

Figure 1.6: QueryThreeDAO

In the query method, the class first checks if the query result already exists in the collection. If so, it returns the saved results. Otherwise, it executes the query on the EduCostStat collection using aggregation pipelines. The aggregation pipelines group the states by their name and calculate the total expense. The results are then sorted in ascending order of the total expense and limited to the top N states. If the query is not empty, the results are saved in the query collection.

4. **QueryFourDAO:** The QueryFourDAO class in this Java package defines a method that queries a MongoDB database for the top 5 states with the highest growth rate in educational expenses over a given number of years ago.



```java
package org.concordia.DAO;

import java.util.ArrayList;

public class QueryFourDAO {
    private static final String QUERY_COLLECTION_NAME = "EduCostStatQueryFour";
    private static final String COLLECTION_NAME = "EduCostStat";

    public static List<Document> query(int baseYear, int yearsAgo, String type, String length) {
        // Get collections
        MongoCollection<Document> collection = MongoDBUtil.getCollection(COLLECTION_NAME);
        MongoCollection<Document> queryCollection = MongoDBUtil.getCollection(QUERY_COLLECTION_NAME);

        // Check if the query result is already in the query collection
        List<Document> existingResult = queryCollection
                .find(Filters.eq("query", Arrays.asList(baseYear, yearsAgo, type, length))).into(new ArrayList<>());

        if (!existingResult.isEmpty()) {
            System.out.println("Query result already exists in the collection.");
            for (Document document : existingResult) {
                List<Document> results = (List<Document>) document.get("result");
                for (Document result : results) {
                    System.out.println(result.getString("state") + ": " + result.getDouble("growthRate"));
                }
            }
            return existingResult;
        }

        // Set start year
        int startYear = baseYear - yearsAgo;

        // Build the aggregation pipeline
        List<Bson> pipeline = new ArrayList<>();
        pipeline.add(Aggregates.match(Filters.and(Filters.gte("year", startYear), Filters.lte("year", baseYear),
                Filters.eq("type", type), Filters.eq("length", length))));
        pipeline.add(Aggregates.group(new Document().append("state", "$state").append("year", "$year"),
                Accumulators.sum("totalExpense", "$value")));
        pipeline.add(Aggregates.sort(Sorts.ascending("_id.state", "_id.year")));
```

Figure 1.7: QueryFourDAO

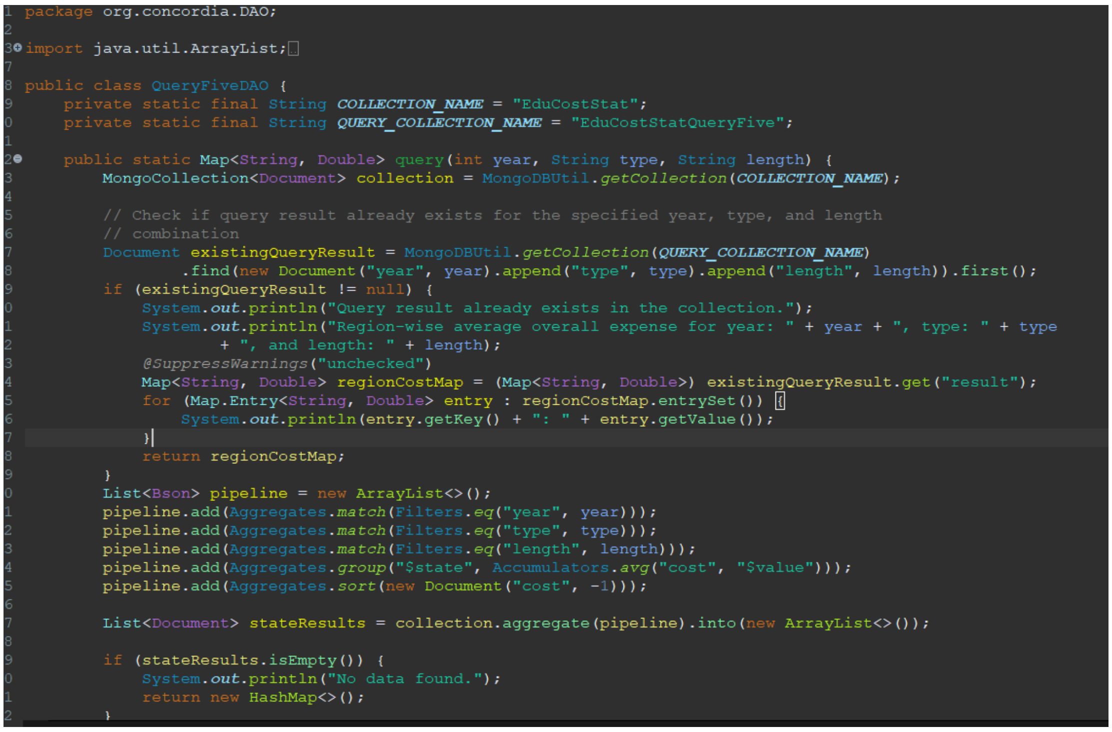
The method takes in four input parameters: the base year (as an integer), the number of years ago to start the comparison (also an integer), the type of education institution (as a string), and the length of education program (as a string).

First, the method checks if the query result already exists in the query collection. If it does, the existing result is returned and printed to the console. Otherwise, an aggregation pipeline is built to filter documents in the EduCostStat collection based on the input parameters [2]. Then, the documents are grouped by state and year, and the total educational expenses are accumulated for each state and year combination.

The list of state-year expenses is then traversed to calculate the growth rate for each state. The growth rate is calculated by comparing the educational expenses between the base year and the specified number of years ago, and then dividing by the starting expenses and multiplying by 100 to get a percentage. The growth rates are then sorted in descending order and the top 5 are returned and printed to the console. Finally, the query result is saved to the query collection for future use.

5. **QueryFiveDAO:** This is the implementation of Query 5 for the Education Cost Statistics application. The goal of this query is to determine the region-wise average overall expense

for a given year, type, and length of education. The query uses the StateRegionMap class to map each state to its corresponding region.



```java
package org.concordia.DAO;

import java.util.ArrayList;

public class QueryFiveDAO {
    private static final String COLLECTION_NAME = "EduCostStat";
    private static final String QUERY_COLLECTION_NAME = "EduCostStatQueryFive";

    public static Map<String, Double> query(int year, String type, String length) {
        MongoCollection<Document> collection = MongoDBUtil.getCollection(COLLECTION_NAME);

        // Check if query result already exists for the specified year, type, and length
        // combination
        Document existingQueryResult = MongoDBUtil.getCollection(QUERY_COLLECTION_NAME)
                .find(new Document("year", year).append("type", type).append("length", length)).first();
        if (existingQueryResult != null) {
            System.out.println("Query result already exists in the collection.");
            System.out.println("Region-wise average overall expense for year: " + year + ", type: " + type
                    + ", and length: " + length);
            @SuppressWarnings("unchecked")
            Map<String, Double> regionCostMap = (Map<String, Double>) existingQueryResult.get("result");
            for (Map.Entry<String, Double> entry : regionCostMap.entrySet()) {
                System.out.println(entry.getKey() + ": " + entry.getValue());
            }
            return regionCostMap;
        }
        List<Bson> pipeline = new ArrayList<>();
        pipeline.add(Aggregates.match(Filters.eq("year", year)));
        pipeline.add(Aggregates.match(Filters.eq("type", type)));
        pipeline.add(Aggregates.match(Filters.eq("length", length)));
        pipeline.add(Aggregates.group("$state", Accumulators.avg("cost", "$value")));
        pipeline.add(Aggregates.sort(new Document("cost", -1)));

        List<Document> stateResults = collection.aggregate(pipeline).into(new ArrayList<>());

        if (stateResults.isEmpty()) {
            System.out.println("No data found.");
            return new HashMap<>();
        }
```

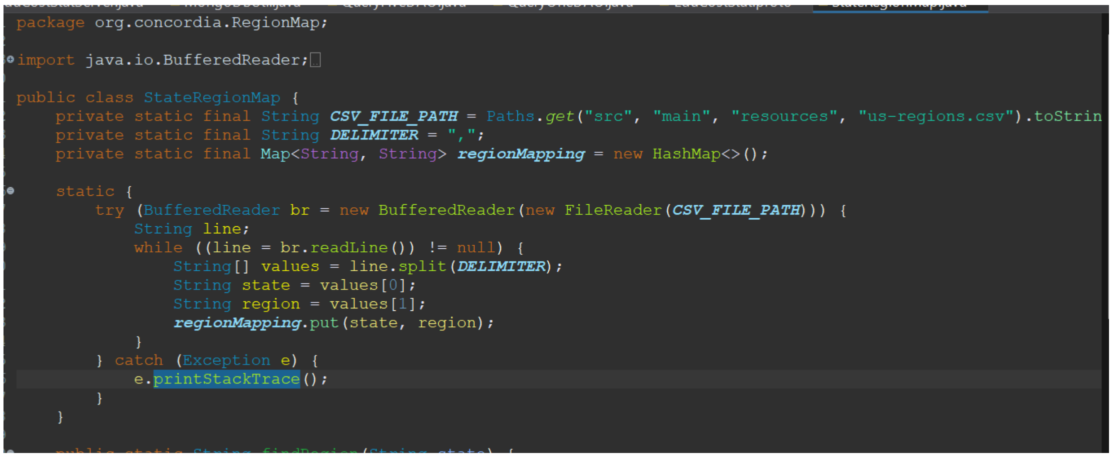
Figure 1.8: QueryFiveDAO

The implementation starts by checking if the query result already exists in the collection for the given year, type, and length. If the result exists, it is returned from the collection. Otherwise, the query is executed using the specified parameters.

The query involves filtering the collection to only include data for the given year, type, and length. Then, the data is grouped by state and the average cost is calculated for each state. The state results are then aggregated by region to calculate the region-wise average overall expense.

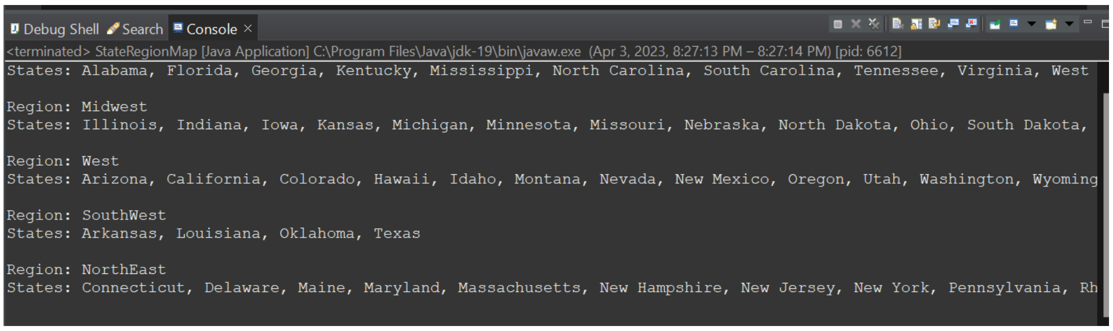
If no data is found for the given parameters, an empty map is returned. Otherwise, the region-wise average overall expense is returned and stored in the collection for future use.

To support the mapping of each state to its corresponding region in Query 5, a CSV file named ”us-regions.csv” has been created. The implementation includes a method to read this file and categorize the US states region-wise as shown below.



```java
package org.concordia.RegionMap;

import java.io.BufferedReader;

public class StateRegionMap {
    private static final String CSV_FILE_PATH = Paths.get("src", "main", "resources", "us-regions.csv").toStrin
    private static final String DELIMITER = ",";
    private static final Map<String, String> regionMapping = new HashMap<>();

    static {
        try (BufferedReader br = new BufferedReader(new FileReader(CSV_FILE_PATH))) {
            String line;
            while ((line = br.readLine()) != null) {
                String[] values = line.split(DELIMITER);
                String state = values[0];
                String region = values[1];
                regionMapping.put(state, region);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static String findRegion(String state) {
```

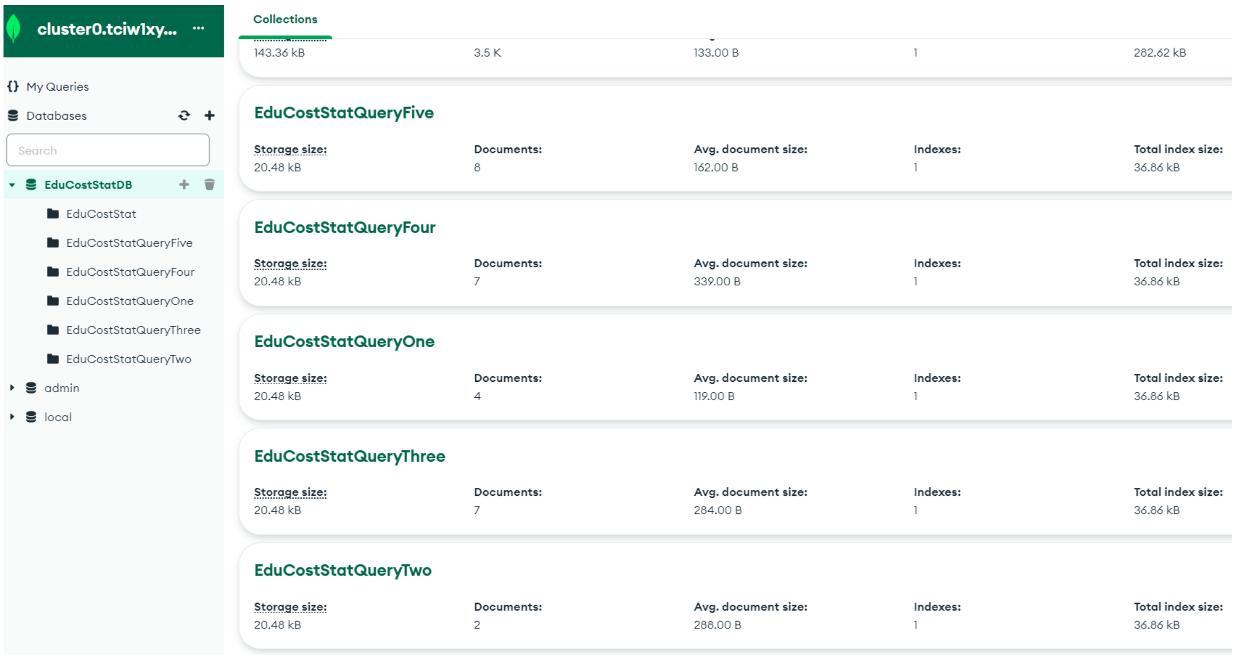
Figure 1.9: Java class ”StateRegionMap”



```
<terminated> StateRegionMap [Java Application] C:\Program Files\Java\jdk-19\bin\javaw.exe (Apr 3, 2023, 8:27:13 PM – 8:27:14 PM) [pid: 6612]
States: Alabama, Florida, Georgia, Kentucky, Mississippi, North Carolina, South Carolina, Tennessee, Virginia, West

Region: Midwest
States: Illinois, Indiana, Iowa, Kansas, Michigan, Minnesota, Missouri, Nebraska, North Dakota, Ohio, South Dakota,

Region: West
States: Arizona, California, Colorado, Hawaii, Idaho, Montana, Nevada, New Mexico, Oregon, Utah, Washington, Wyoming

Region: SouthWest
States: Arkansas, Louisiana, Oklahoma, Texas

Region: NorthEast
States: Connecticut, Delaware, Maine, Maryland, Massachusetts, New Hampshire, New Jersey, New York, Pennsylvania, Rh
```

Figure 1.10: States categorized in Regions

The figure below shows the collections saved for each queries on the ”EduCostStatDB” database.



143.36 kB	3.5 K	133.00 B	1	282.62 kB

EduCostStatQueryFive

Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	8	162.00 B	1	36.86 kB

EduCostStatQueryFour

Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	7	339.00 B	1	36.86 kB

EduCostStatQueryOne

Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	4	119.00 B	1	36.86 kB

EduCostStatQueryThree

Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	7	284.00 B	1	36.86 kB

EduCostStatQueryTwo

Storage size:	Documents:	Avg. document size:	Indexes:	Total index size:
20.48 kB	2	288.00 B	1	36.86 kB

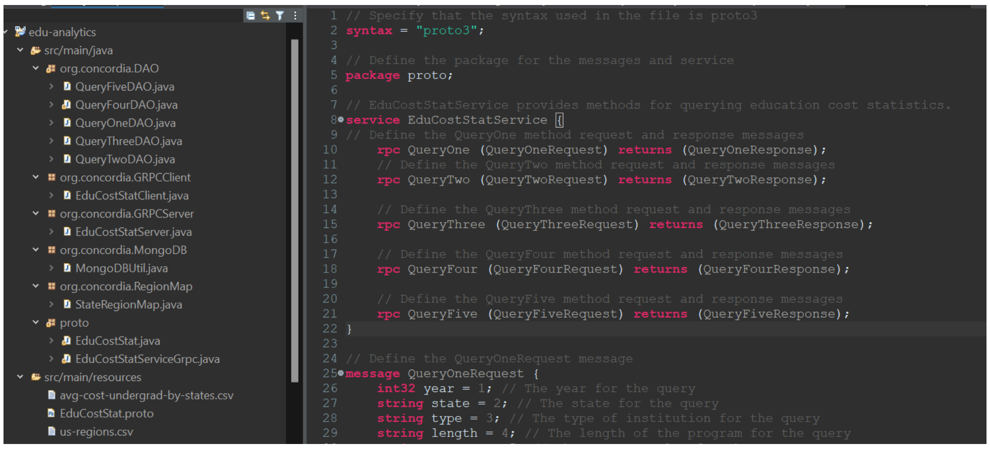
Figure 1.11: DAO Query collections

2 Task 2: Data Communication Interface Definition and Service Implementation

2.1 Task 2.1: Protocol Buff Definition File Request and Response messages Service definitions

The ProtoBuff definition file is created to represent the request, response and service for each query in Task 1 [3].

Request and response rpc services for each query

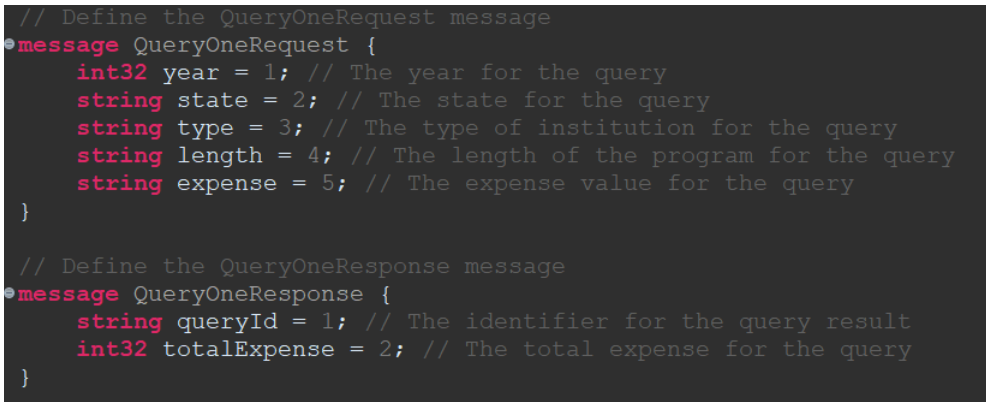


```
edu-analytics
  src/main/java
    org.concordia.DAO
      QueryFiveDAO.java
      QueryFourDAO.java
      QueryOneDAO.java
      QueryThreeDAO.java
      QueryTwoDAO.java
    org.concordia.GRPCClient
      EduCostStatClient.java
    org.concordia.GRPCServer
      EduCostStatServer.java
    org.concordia.MongoDB
      MongoDBUtil.java
    org.concordia.RegionMap
      StateRegionMap.java
    proto
      EduCostStat.java
      EduCostStatServiceGrpc.java
  src/main/resources
    avg-cost-undergrad-by-states.csv
    EduCostStat.proto
    us-regions.csv
```

```
// Specify that the syntax used in the file is proto3
syntax = "proto3";

// Define the package for the messages and service
package proto;

// EduCostStatService provides methods for querying education cost statistics.
service EduCostStatService {
// Define the QueryOne method request and response messages
    rpc QueryOne (QueryOneRequest) returns (QueryOneResponse);
    // Define the QueryTwo method request and response messages
    rpc QueryTwo (QueryTwoRequest) returns (QueryTwoResponse);

    // Define the QueryThree method request and response messages
    rpc QueryThree (QueryThreeRequest) returns (QueryThreeResponse);

    // Define the QueryFour method request and response messages
    rpc QueryFour (QueryFourRequest) returns (QueryFourResponse);

    // Define the QueryFive method request and response messages
    rpc QueryFive (QueryFiveRequest) returns (QueryFiveResponse);
}

// Define the QueryOneRequest message
message QueryOneRequest {
    int32 year = 1; // The year for the query
    string state = 2; // The state for the query
    string type = 3; // The type of institution for the query
    string length = 4; // The length of the program for the query
```

Figure 2.1: rpc services for reuqest and response for each query

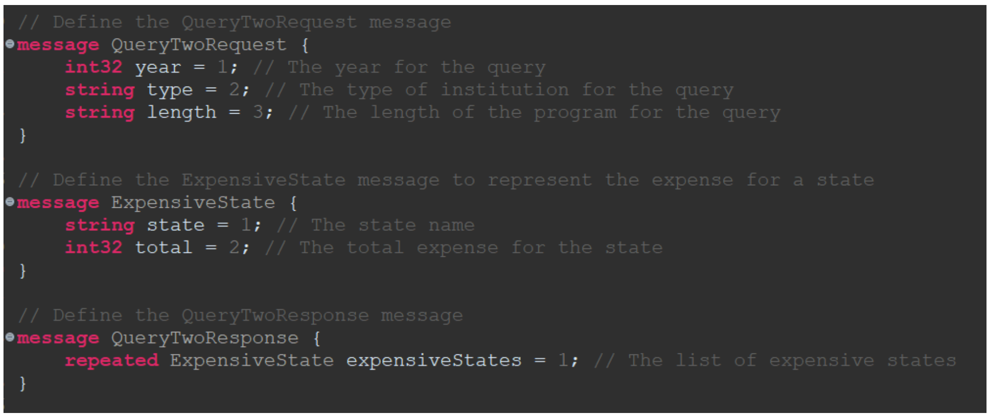
Messages for queryOne



```
// Define the QueryOneRequest message
message QueryOneRequest {
    int32 year = 1; // The year for the query
    string state = 2; // The state for the query
    string type = 3; // The type of institution for the query
    string length = 4; // The length of the program for the query
    string expense = 5; // The expense value for the query
}

// Define the QueryOneResponse message
message QueryOneResponse {
    string queryId = 1; // The identifier for the query result
    int32 totalExpense = 2; // The total expense for the query
}
```

Figure 2.2: Query one messages

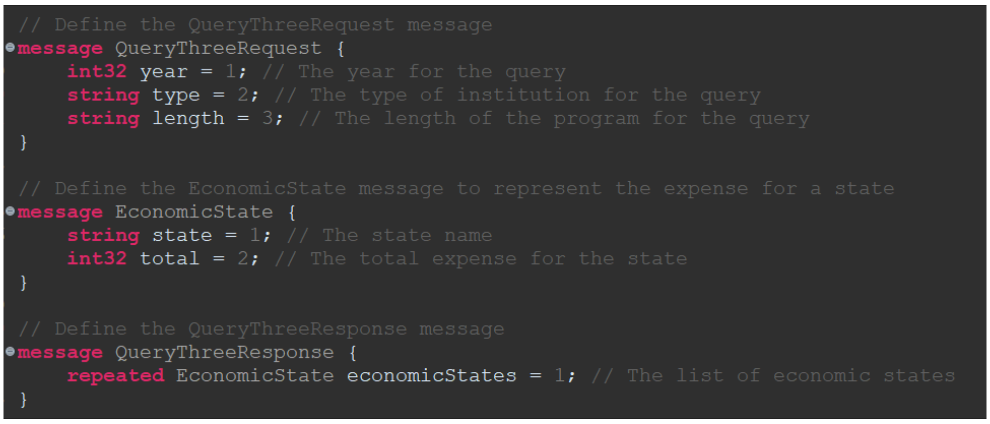
Messages for queryTwo



```
// Define the QueryTwoRequest message
message QueryTwoRequest {
    int32 year = 1; // The year for the query
    string type = 2; // The type of institution for the query
    string length = 3; // The length of the program for the query
}

// Define the ExpensiveState message to represent the expense for a state
message ExpensiveState {
    string state = 1; // The state name
    int32 total = 2; // The total expense for the state
}

// Define the QueryTwoResponse message
message QueryTwoResponse {
    repeated ExpensiveState expensiveStates = 1; // The list of expensive states
}
```

Figure 2.3: Query two messages

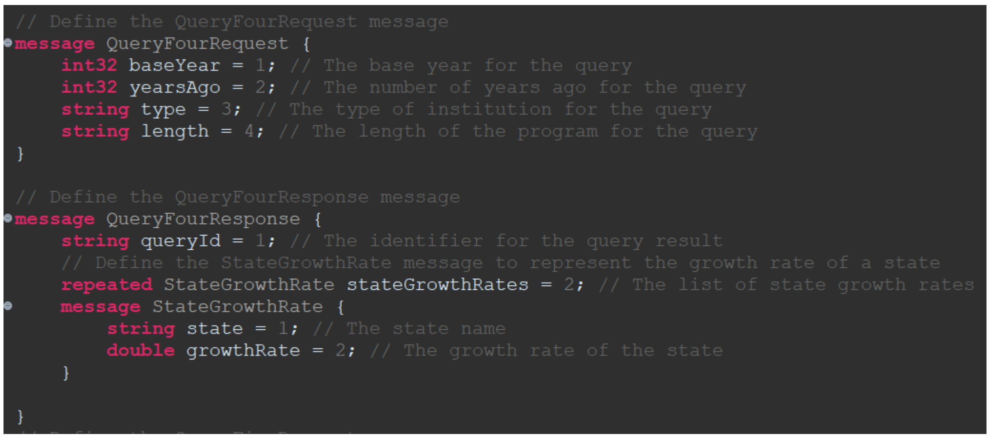
Messages for queryThree



```
// Define the QueryThreeRequest message
message QueryThreeRequest {
    int32 year = 1; // The year for the query
    string type = 2; // The type of institution for the query
    string length = 3; // The length of the program for the query
}

// Define the EconomicState message to represent the expense for a state
message EconomicState {
    string state = 1; // The state name
    int32 total = 2; // The total expense for the state
}

// Define the QueryThreeResponse message
message QueryThreeResponse {
    repeated EconomicState economicStates = 1; // The list of economic states
}
```

Figure 2.4: Query three messages

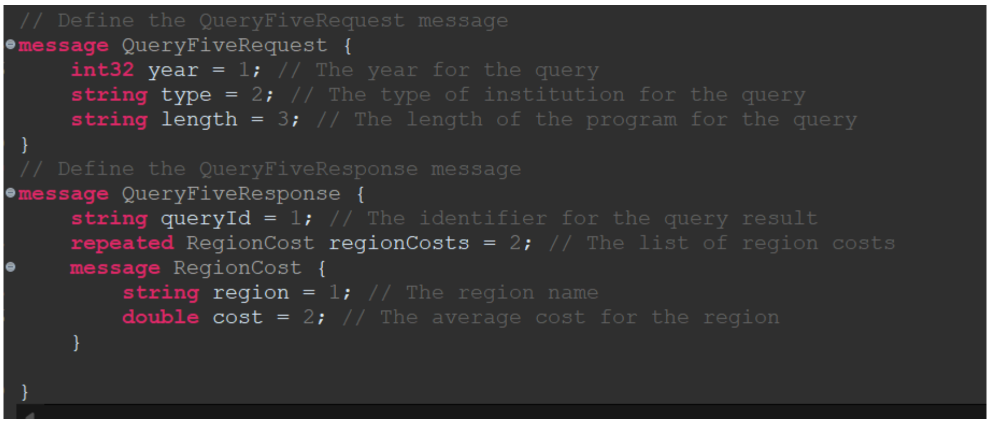
Messages for queryFour



```
// Define the QueryFourRequest message
message QueryFourRequest {
    int32 baseYear = 1; // The base year for the query
    int32 yearsAgo = 2; // The number of years ago for the query
    string type = 3; // The type of institution for the query
    string length = 4; // The length of the program for the query
}

// Define the QueryFourResponse message
message QueryFourResponse {
    string queryId = 1; // The identifier for the query result
    // Define the StateGrowthRate message to represent the growth rate of a state
    repeated StateGrowthRate stateGrowthRates = 2; // The list of state growth rates
    message StateGrowthRate {
        string state = 1; // The state name
        double growthRate = 2; // The growth rate of the state
    }

}
```

Figure 2.5: Query four messages

Messages for queryFive



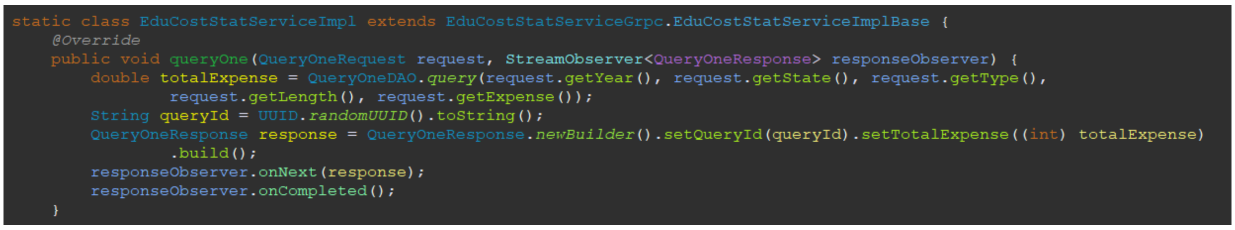
```
// Define the QueryFiveRequest message
message QueryFiveRequest {
    int32 year = 1; // The year for the query
    string type = 2; // The type of institution for the query
    string length = 3; // The length of the program for the query
}
// Define the QueryFiveResponse message
message QueryFiveResponse {
    string queryId = 1; // The identifier for the query result
    repeated RegionCost regionCosts = 2; // The list of region costs
    message RegionCost {
        string region = 1; // The region name
        double cost = 2; // The average cost for the region
    }

}
```

Figure 2.6: Query five messages

2.2 Task 2.2: Java Program for gRPC Services

QueryOneService: The class EduCostStatServiceImpl inside the ”EduCostStatServer.java” under ”org.concordia.GRPCServer” package is the implementation of the gRPC service defined in EduCostStatService.proto. It contains five query services: queryOne, queryTwo, queryThree, queryFour, and queryFive.

QueryOneService:



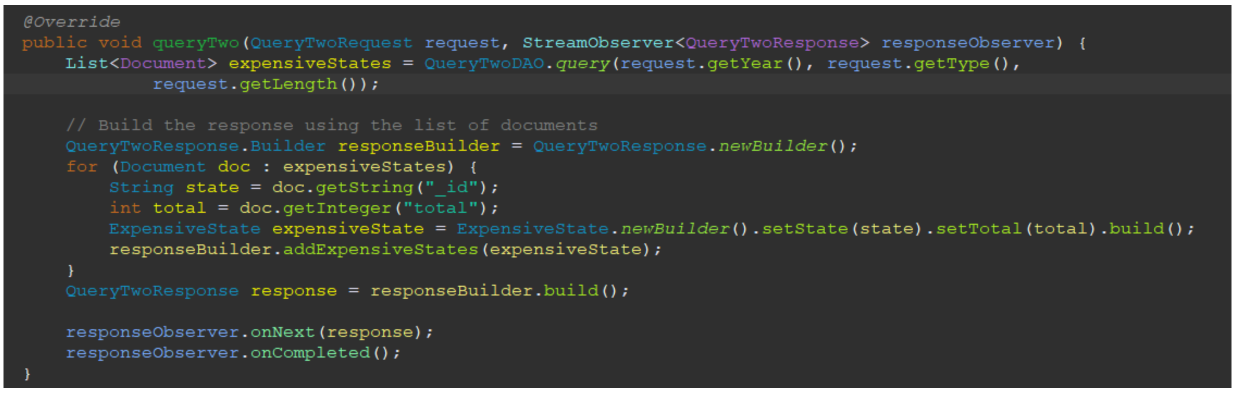
```
static class EduCostStatServiceImpl extends EduCostStatServiceGrpc.EduCostStatServiceImplBase {
    @Override
    public void queryOne(QueryOneRequest request, StreamObserver<QueryOneResponse> responseObserver) {
        double totalExpense = QueryOneDAO.query(request.getYear(), request.getState(), request.getType(),
                request.getLength(), request.getExpense());
        String queryId = UUID.randomUUID().toString();
        QueryOneResponse response = QueryOneResponse.newBuilder().setQueryId(queryId).setTotalExpense((int) totalExpense)
                .build();
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    }
```

Figure 2.7: queryOne service implementation

This code snippet represents the implementation of the queryOne method in the EduCostStatServiceImpl class. This method is responsible for querying the database collection to retrieve the total expense based on the given query parameters such as year, state, type, length, and expense. It uses the QueryOneDAO class to perform the actual database query operation and calculates the total expense based on the results returned. Once the total expense is calculated, the method creates a unique query ID using the
UUID.randomUUID().toString() method and creates a QueryOneResponse object using the queryId and totalExpense values. The responseObserver object is then used to send the response back to

the client using the onNext() method and complete the response using the onCompleted() method.

QueryTwoService:



```
@Override
public void queryTwo(QueryTwoRequest request, StreamObserver<QueryTwoResponse> responseObserver) {
    List<Document> expensiveStates = QueryTwoDAO.query(request.getYear(), request.getType(),
            request.getLength());

    // Build the response using the list of documents
    QueryTwoResponse.Builder responseBuilder = QueryTwoResponse.newBuilder();
    for (Document doc : expensiveStates) {
        String state = doc.getString("_id");
        int total = doc.getInteger("total");
        ExpensiveState expensiveState = ExpensiveState.newBuilder().setState(state).setTotal(total).build();
        responseBuilder.addExpensiveStates(expensiveState);
    }
    QueryTwoResponse response = responseBuilder.build();

    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

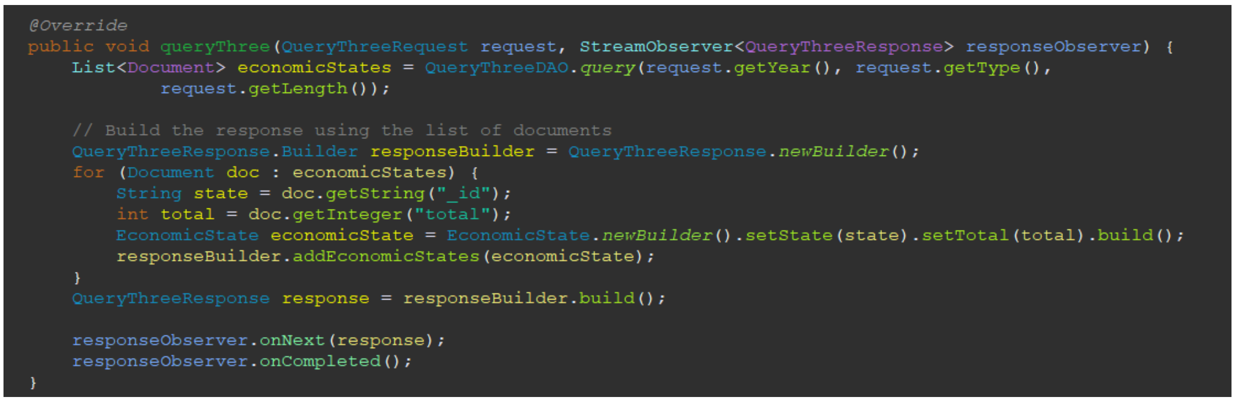
Figure 2.8: queryTwo service implementation

This code implements the gRPC service method queryTwo, which handles the second query type mentioned in the system architecture. It takes in a QueryTwoRequest object, which contains the year, type, and length parameters for the query.

It then calls the QueryTwoDAO.query() method to retrieve a list of documents representing the top 5 most expensive states, based on the given parameters. It builds the response by iterating through the list of documents and creating an ExpensiveState object for each document.

Finally, it builds a QueryTwoResponse object using the QueryTwoResponse.Builder class, adding each ExpensiveState object to the response. It sends the response to the client using responseObserver.onNext() and then signals the end of the response using responseObserver.onCompleted().

QueryThreeService:



```
@Override
public void queryThree(QueryThreeRequest request, StreamObserver<QueryThreeResponse> responseObserver) {
    List<Document> economicStates = QueryThreeDAO.query(request.getYear(), request.getType(),
            request.getLength());

    // Build the response using the list of documents
    QueryThreeResponse.Builder responseBuilder = QueryThreeResponse.newBuilder();
    for (Document doc : economicStates) {
        String state = doc.getString("_id");
        int total = doc.getInteger("total");
        EconomicState economicState = EconomicState.newBuilder().setState(state).setTotal(total).build();
        responseBuilder.addEconomicStates(economicState);
    }
    QueryThreeResponse response = responseBuilder.build();

    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

Figure 2.9: queryThree service implementation

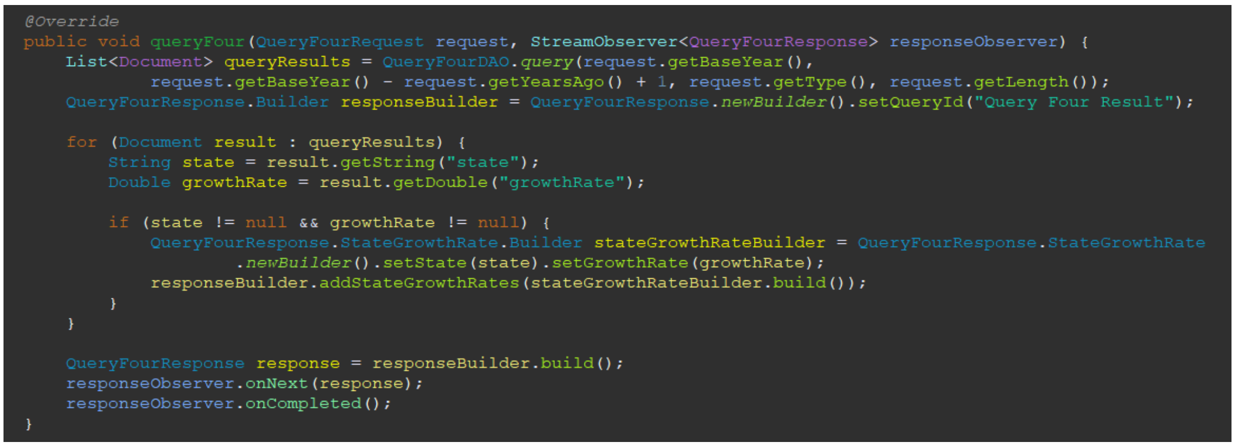
The queryThree method is a server-side implementation of the queryThree gRPC service defined in the .proto file. It takes a QueryThreeRequest object as input and returns a QueryThreeResponse object as output.

The method first calls the QueryThreeDAO.query() method to retrieve the top 5 most economic states (with overall expense) given a year, type, and length. It then builds a response using the list of documents returned by the query() method.

For each document in the list, the method extracts the state and total fields, and creates an EconomicState object with those values. It then adds the EconomicState object to a QueryThreeResponse object using the addEconomicStates() method.

Finally, the method builds the complete QueryThreeResponse object using the build() method of the QueryThreeResponse.Builder class, and sends the response to the client using the onNext() method of the StreamObserver object. The onCompleted() method is then called to signal the end of the RPC call.

QueryFourService:



```java
@Override
public void queryFour(QueryFourRequest request, StreamObserver<QueryFourResponse> responseObserver) {
    List<Document> queryResults = QueryFourDAO.query(request.getBaseYear(),
            request.getBaseYear() - request.getYearsAgo() + 1, request.getType(), request.getLength());
    QueryFourResponse.Builder responseBuilder = QueryFourResponse.newBuilder().setQueryId("Query Four Result");

    for (Document result : queryResults) {
        String state = result.getString("state");
        Double growthRate = result.getDouble("growthRate");

        if (state != null && growthRate != null) {
            QueryFourResponse.StateGrowthRate.Builder stateGrowthRateBuilder = QueryFourResponse.StateGrowthRate
                    .newBuilder().setState(state).setGrowthRate(growthRate);
            responseBuilder.addStateGrowthRates(stateGrowthRateBuilder.build());
        }
    }

    QueryFourResponse response = responseBuilder.build();
    responseObserver.onNext(response);
    responseObserver.onCompleted();
}
```

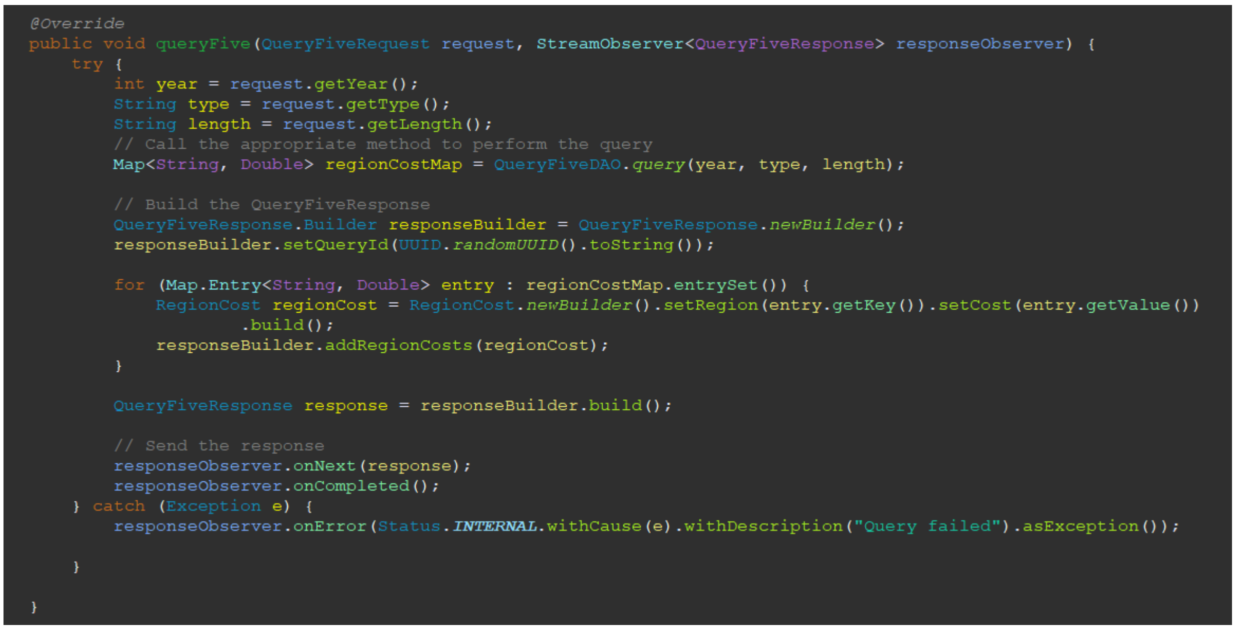
Figure 2.10: queryFour service implementation

The queryFour method of the EduCostStatServiceImpl class implements the fourth query defined in the system. This method receives a QueryFourRequest message from the gRPC client, which contains information about the base year, the number of years ago, the type, and length of education for which the growth rate of expenses should be calculated. The method calls the QueryFourDAO class to perform the query, passing the parameters of the request. The result of the query is a list of Document objects, which contain the growth rate of expenses for each state that matches the query.

The method then creates a QueryFourResponse.Builder object and sets its query ID as ”Query Four Result”. It iterates through the list of Document objects and extracts the state and growth rate information. For each pair of state and growth rate, it creates a QueryFourResponse.StateGrowthRate.Builder object and sets its state and growth rate attributes. Finally, it adds the StateGrowthRate object to the QueryFourResponse.Builder object.

After all StateGrowthRate objects have been added to the response, the method builds the final QueryFourResponse object using the QueryFourResponse.Builder object and sends it to the gRPC client using the responseObserver object.

QueryFiveService:



```java
@Override
public void queryFive(QueryFiveRequest request, StreamObserver<QueryFiveResponse> responseObserver) {
    try {
        int year = request.getYear();
        String type = request.getType();
        String length = request.getLength();
        // Call the appropriate method to perform the query
        Map<String, Double> regionCostMap = QueryFiveDAO.query(year, type, length);

        // Build the QueryFiveResponse
        QueryFiveResponse.Builder responseBuilder = QueryFiveResponse.newBuilder();
        responseBuilder.setQueryId(UUID.randomUUID().toString());

        for (Map.Entry<String, Double> entry : regionCostMap.entrySet()) {
            RegionCost regionCost = RegionCost.newBuilder().setRegion(entry.getKey()).setCost(entry.getValue())
                    .build();
            responseBuilder.addRegionCosts(regionCost);
        }

        QueryFiveResponse response = responseBuilder.build();

        // Send the response
        responseObserver.onNext(response);
        responseObserver.onCompleted();
    } catch (Exception e) {
        responseObserver.onError(Status.INTERNAL.withCause(e).withDescription("Query failed").asException());

    }
}
```

Figure 2.11: queryFive service implementation

The queryFive method in the EduCostStatServiceImpl class is responsible for handling the fifth query, which aggregates region’s average overall expense for a given year, type, and length. The method calls the appropriate method to perform the query in the QueryFiveDAO class and gets the result as a Map of region names and their average expenses.

The method then builds a QueryFiveResponse using the responseBuilder and adds RegionCost objects to the builder for each entry in the regionCostMap. Each RegionCost object contains the region name and its average expense. Finally, the method sends the response to the client using responseObserver.onNext() and responseObserver.onCompleted() methods.

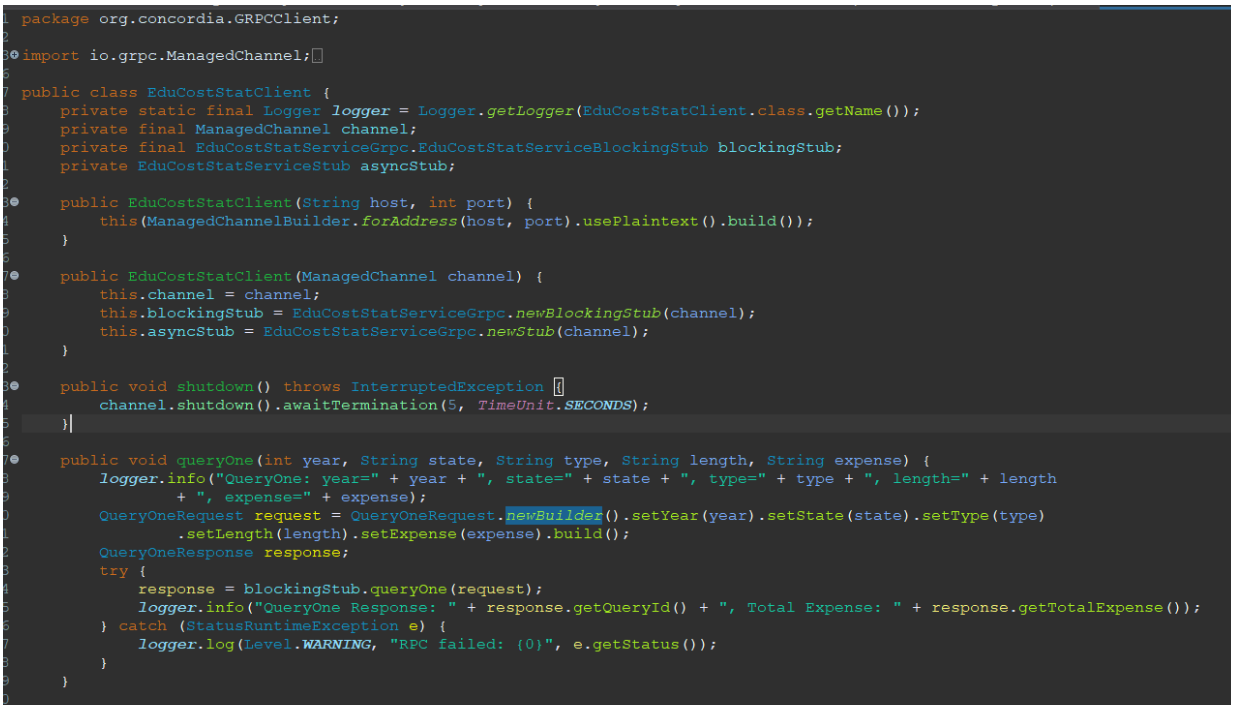
If an exception occurs during the query, the method catches it and sends an error response to the client using the responseObserver.onError() method. The error message includes a description of the error and the status of the gRPC response.

Task 2.3: gRPC Client and Server Code

EduCostStatClient: The client code is responsible for sending requests to the gRPC server to retrieve data from the database. It is developed using Java and uses the Protocol Buff definition file to communicate with the gRPC server using RPC calls. The client code consists of methods that correspond to each of the five queries defined in the Protocol Buff definition file.

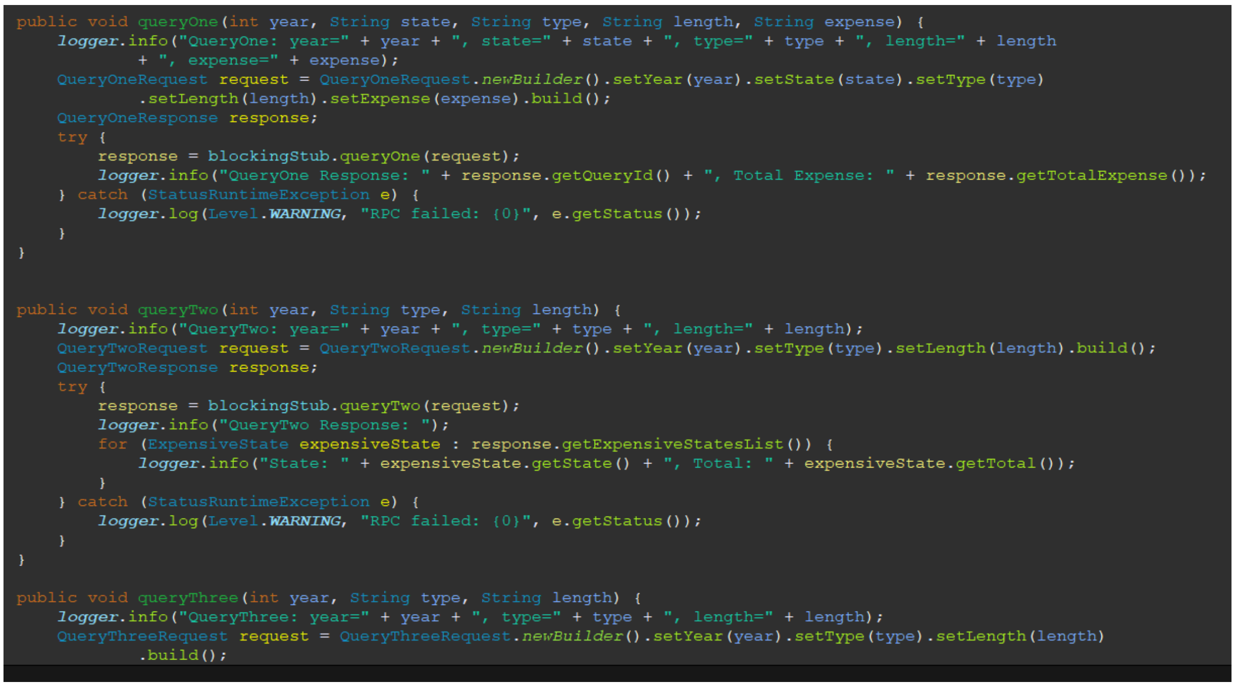
When a method is called, a request is created with the appropriate parameters and sent to the gRPC server. Once the response is received, the data is extracted and returned to the caller. The client code handles errors that may occur during the RPC call and logs them.

The client code can be run as a standalone Java application, and it provides a command-line interface for users to input parameters for each query. It allows the user to specify the year, state, type, length, and expense for query one, and the year, type, and length for queries two to five.



```java
package org.concordia.GRPCClient;

import io.grpc.ManagedChannel;

public class EduCostStatClient {
    private static final Logger logger = Logger.getLogger(EduCostStatClient.class.getName());
    private final ManagedChannel channel;
    private final EduCostStatServiceGrpc.EduCostStatServiceBlockingStub blockingStub;
    private EduCostStatServiceStub asyncStub;

    public EduCostStatClient(String host, int port) {
        this(ManagedChannelBuilder.forAddress(host, port).usePlaintext().build());
    }

    public EduCostStatClient(ManagedChannel channel) {
        this.channel = channel;
        this.blockingStub = EduCostStatServiceGrpc.newBlockingStub(channel);
        this.asyncStub = EduCostStatServiceGrpc.newStub(channel);
    }

    public void shutdown() throws InterruptedException {
        channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
    }

    public void queryOne(int year, String state, String type, String length, String expense) {
        logger.info("QueryOne: year=" + year + ", state=" + state + ", type=" + type + ", length=" + length
                + ", expense=" + expense);
        QueryOneRequest request = QueryOneRequest.newBuilder().setYear(year).setState(state).setType(type)
                .setLength(length).setExpense(expense).build();
        QueryOneResponse response;
        try {
            response = blockingStub.queryOne(request);
            logger.info("QueryOne Response: " + response.getQueryId() + ", Total Expense: " + response.getTotalExpense());
        } catch (StatusRuntimeException e) {
            logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
        }
    }
```

Figure 2.12: Client code



```java
public void queryOne(int year, String state, String type, String length, String expense) {
    logger.info("QueryOne: year=" + year + ", state=" + state + ", type=" + type + ", length=" + length
            + ", expense=" + expense);
    QueryOneRequest request = QueryOneRequest.newBuilder().setYear(year).setState(state).setType(type)
            .setLength(length).setExpense(expense).build();
    QueryOneResponse response;
    try {
        response = blockingStub.queryOne(request);
        logger.info("QueryOne Response: " + response.getQueryId() + ", Total Expense: " + response.getTotalExpense());
    } catch (StatusRuntimeException e) {
        logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    }
}


public void queryTwo(int year, String type, String length) {
    logger.info("QueryTwo: year=" + year + ", type=" + type + ", length=" + length);
    QueryTwoRequest request = QueryTwoRequest.newBuilder().setYear(year).setType(type).setLength(length).build();
    QueryTwoResponse response;
    try {
        response = blockingStub.queryTwo(request);
        logger.info("QueryTwo Response: ");
        for (ExpensiveState expensiveState : response.getExpensiveStatesList()) {
            logger.info("State: " + expensiveState.getState() + ", Total: " + expensiveState.getTotal());
        }
    } catch (StatusRuntimeException e) {
        logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
    }
}

public void queryThree(int year, String type, String length) {
    logger.info("QueryThree: year=" + year + ", type=" + type + ", length=" + length);
    QueryThreeRequest request = QueryThreeRequest.newBuilder().setYear(year).setType(type).setLength(length)
            .build();
```

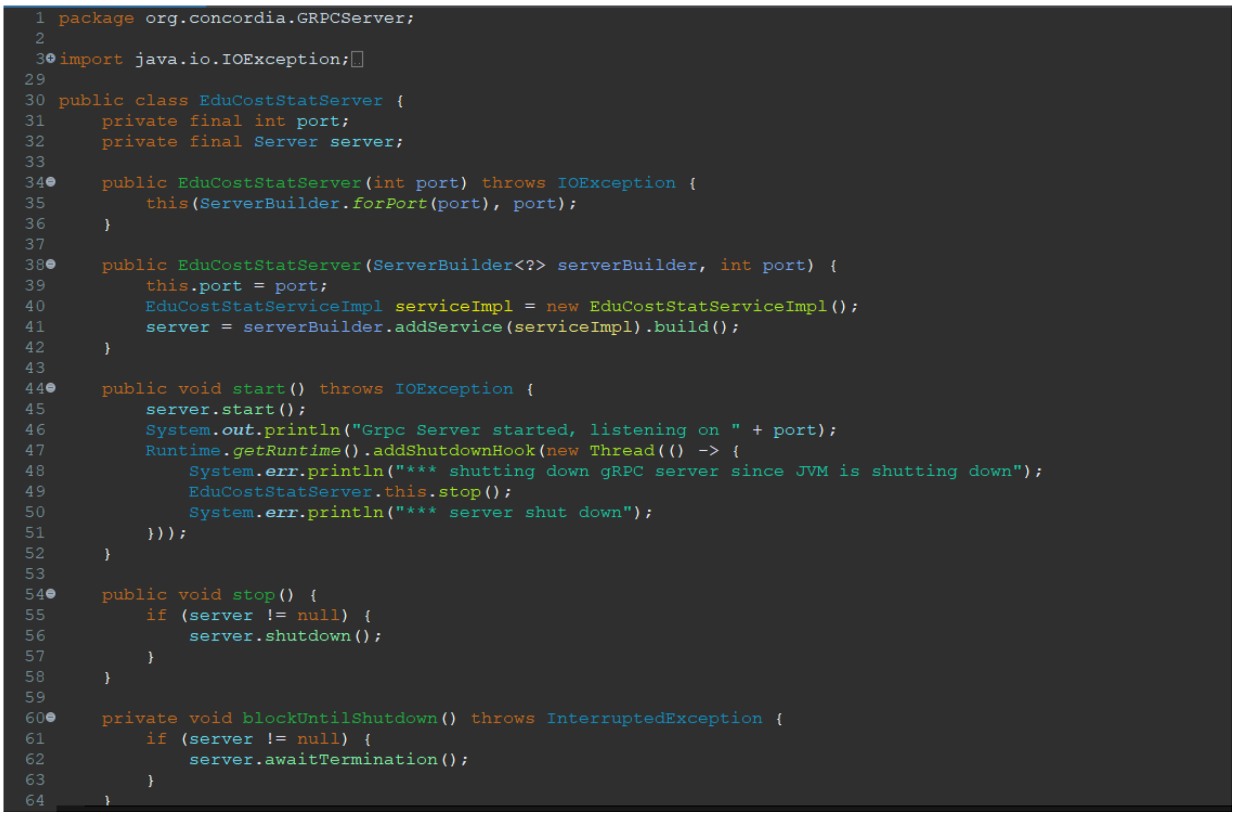
Figure 2.13: Client code

EduCostStatServer: The server code is responsible for handling the incoming gRPC requests from the client and processing them according to the defined services. It starts a gRPC server on a specified port and listens for incoming requests.

The EduCostStatServer class is responsible for starting and stopping the gRPC server, and it creates an instance of the EduCostStatServiceImpl class to handle incoming requests. The EduCostStatServer class starts the gRPC server by calling the start() method, which in turn starts the server and listens on the specified port. When the server is shut down, either manually or due to an error, the stop() method is called to shut down the server.

The EduCostStatServiceImpl class contains the actual implementation of the gRPC services, which are defined in the Protocol Buffers file. It contains the queryOne(), queryTwo(), query-Three(), queryFour(), and queryFive() methods, each corresponding to one of the five defined services.

Each service method receives a request message from the client and returns a response message. The response message is built using the data returned from the corresponding DAO class. The response message is then sent back to the client through the responseObserver object.



```java
package org.concordia.GRPCServer;

import java.io.IOException;

public class EduCostStatServer {
    private final int port;
    private final Server server;

    public EduCostStatServer(int port) throws IOException {
        this(ServerBuilder.forPort(port), port);
    }

    public EduCostStatServer(ServerBuilder<?> serverBuilder, int port) {
        this.port = port;
        EduCostStatServiceImpl serviceImpl = new EduCostStatServiceImpl();
        server = serverBuilder.addService(serviceImpl).build();
    }

    public void start() throws IOException {
        server.start();
        System.out.println("Grpc Server started, listening on " + port);
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.err.println("*** shutting down gRPC server since JVM is shutting down");
            EduCostStatServer.this.stop();
            System.err.println("*** server shut down");
        }));
    }

    public void stop() {
        if (server != null) {
            server.shutdown();
        }
    }

    private void blockUntilShutdown() throws InterruptedException {
        if (server != null) {
            server.awaitTermination();
        }
    }
```

Figure 2.14: Server code

Communication flow:

The client sends a gRPC request to the server. The server receives the request and processes it. The server sends a gRPC response back to the client. The client receives the response and displays it.

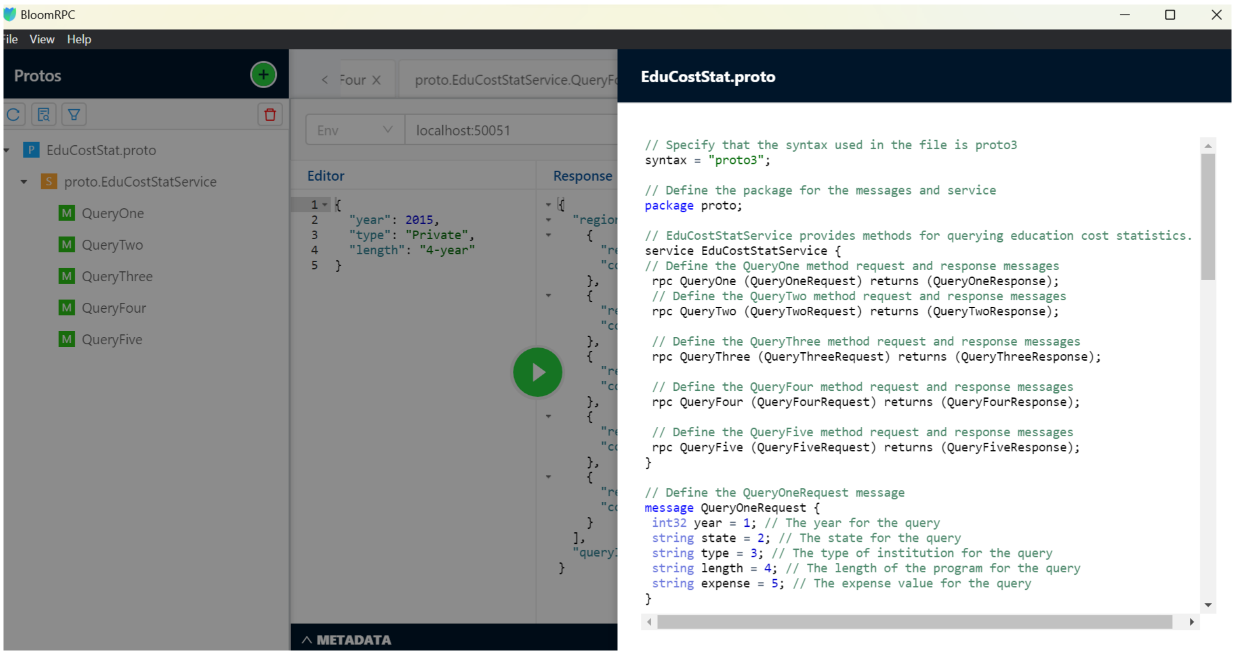
Implementation details:

The EduCostStatServer is implemented using the gRPC Java framework. It provides a set of gRPC services that can be used to perform various queries on education cost statistics. The server listens on the port 50051 and accepts incoming requests. It then processes the request and returns the response back to the client. The server uses DAO classes to interact with the MongoDB database to perform the necessary queries. The EduCostStatClient is also implemented using the gRPC Java framework. It sends gRPC requests to the server and receives responses. The client provides methods that can be used to perform each query by calling the appropriate gRPC service method. The client uses a ManagedChannel to communicate with the server. The channel provides a low-level abstraction for establishing and managing connections to the server. To perform the queries asynchronously, the client uses the gRPC stub's async method and provides a StreamObserver to handle the response. The StreamObserver receives onNext, onError, and onCompleted events as the response is processed by the server.

3 Client gRPC Service Testing using Bloom RPC

Bloom RPC [4] has been employed for testing the gRPC services for all the queries implemented. Firstly, the proto file was uploaded to Bloom RPC and then its services were utilized to execute and test the queries. The results obtained from Bloom RPC were found to be in accordance with the requirements for each query. Moreover, the performance of the system was observed to be satisfactory during the testing phase.

Proto file import

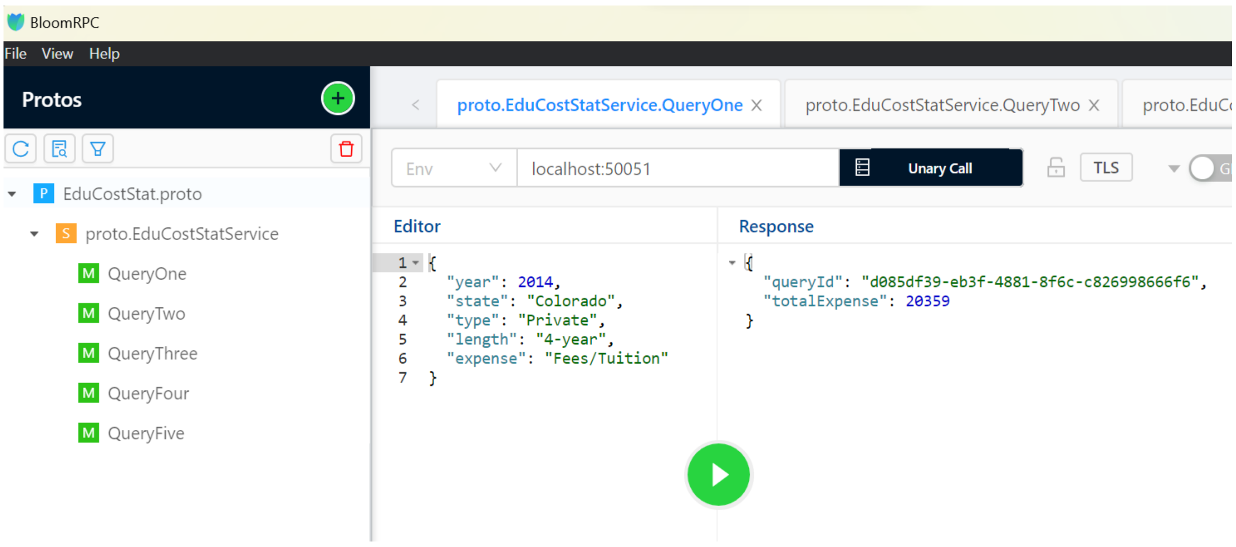


```
{
  "year": 2015,
  "type": "Private",
  "length": "4-year"
}
```

```
// Specify that the syntax used in the file is proto3
syntax = "proto3";

// Define the package for the messages and service
package proto;

// EduCostStatService provides methods for querying education cost statistics.
service EduCostStatService {
// Define the QueryOne method request and response messages
 rpc QueryOne (QueryOneRequest) returns (QueryOneResponse);
 // Define the QueryTwo method request and response messages
 rpc QueryTwo (QueryTwoRequest) returns (QueryTwoResponse);

 // Define the QueryThree method request and response messages
 rpc QueryThree (QueryThreeRequest) returns (QueryThreeResponse);

 // Define the QueryFour method request and response messages
 rpc QueryFour (QueryFourRequest) returns (QueryFourResponse);

 // Define the QueryFive method request and response messages
 rpc QueryFive (QueryFiveRequest) returns (QueryFiveResponse);
}

// Define the QueryOneRequest message
message QueryOneRequest {
 int32 year = 1; // The year for the query
 string state = 2; // The state for the query
 string type = 3; // The type of institution for the query
 string length = 4; // The length of the program for the query
 string expense = 5; // The expense value for the query
}
```

Figure 3.1: Importing the proto file to the Bloom RPC

Testing QueryOne thorugh gRPC: The implementation of the first query has been successful in returning the total expense for the specified combination of "year", "state", "type", "length", and "Fees/Tuition" as 20359. This indicates that the system is able to accurately retrieve and process the required data from the EduCostStat collection in the MongoDB database.

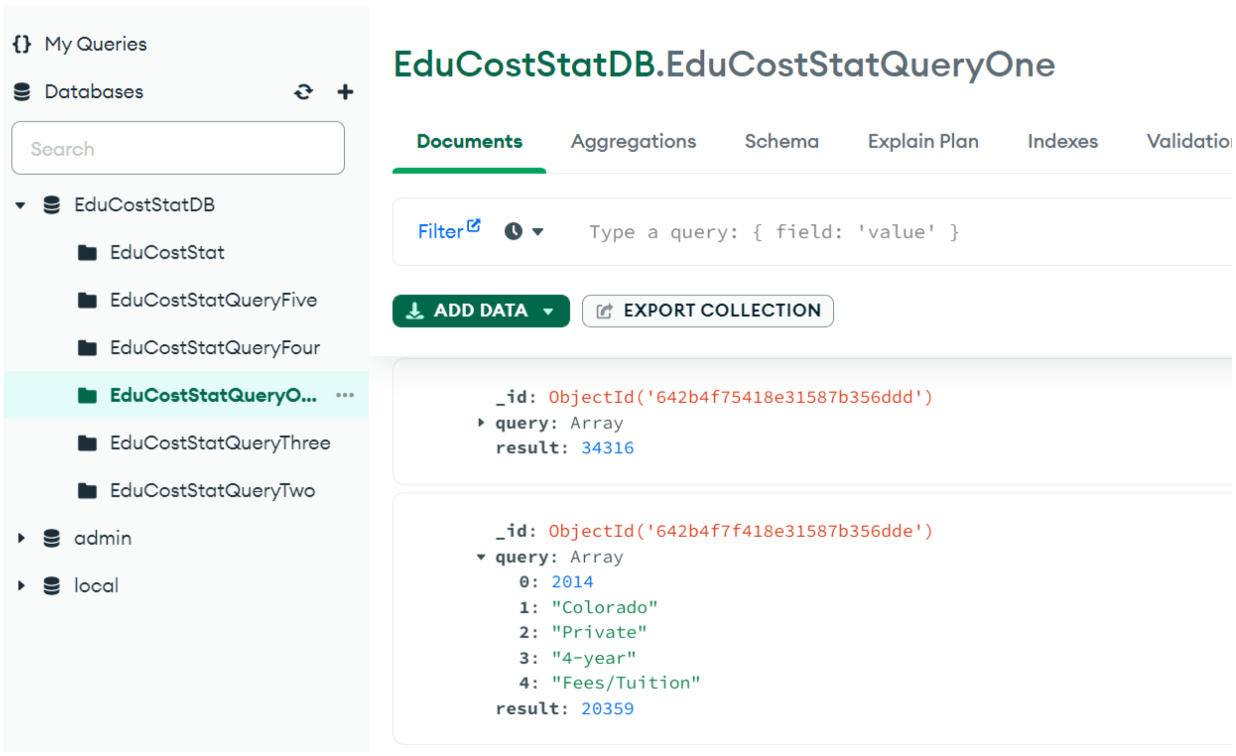


```
{
  "year": 2014,
  "state": "Colorado",
  "type": "Private",
  "length": "4-year",
  "expense": "Fees/Tuition"
}
```

```
{
  "queryId": "d085df39-eb3f-4881-8f6c-c826998666f6",
  "totalExpense": 20359
}
```

Figure 3.2: Client test Query One

Also, the data is saved in the ”EduCostStaQueryOne” collection as shown below.

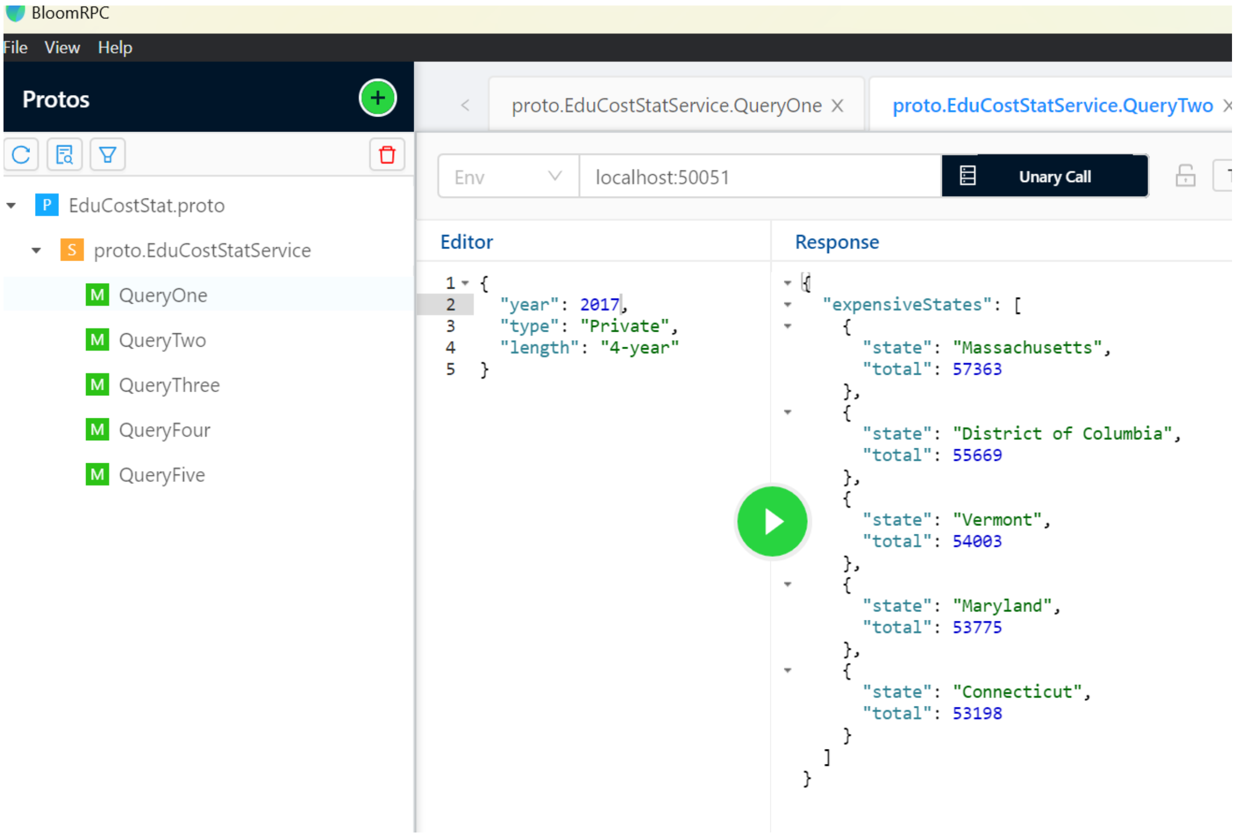


```
_id: ObjectId('642b4f75418e31587b356ddd')
query: Array
result: 34316
```

```
_id: ObjectId('642b4f7f418e31587b356dde')
query: Array
  0: 2014
  1: "Colorado"
  2: "Private"
  3: "4-year"
  4: "Fees/Tuition"
result: 20359
```

Figure 3.3: Query data saved in the collection

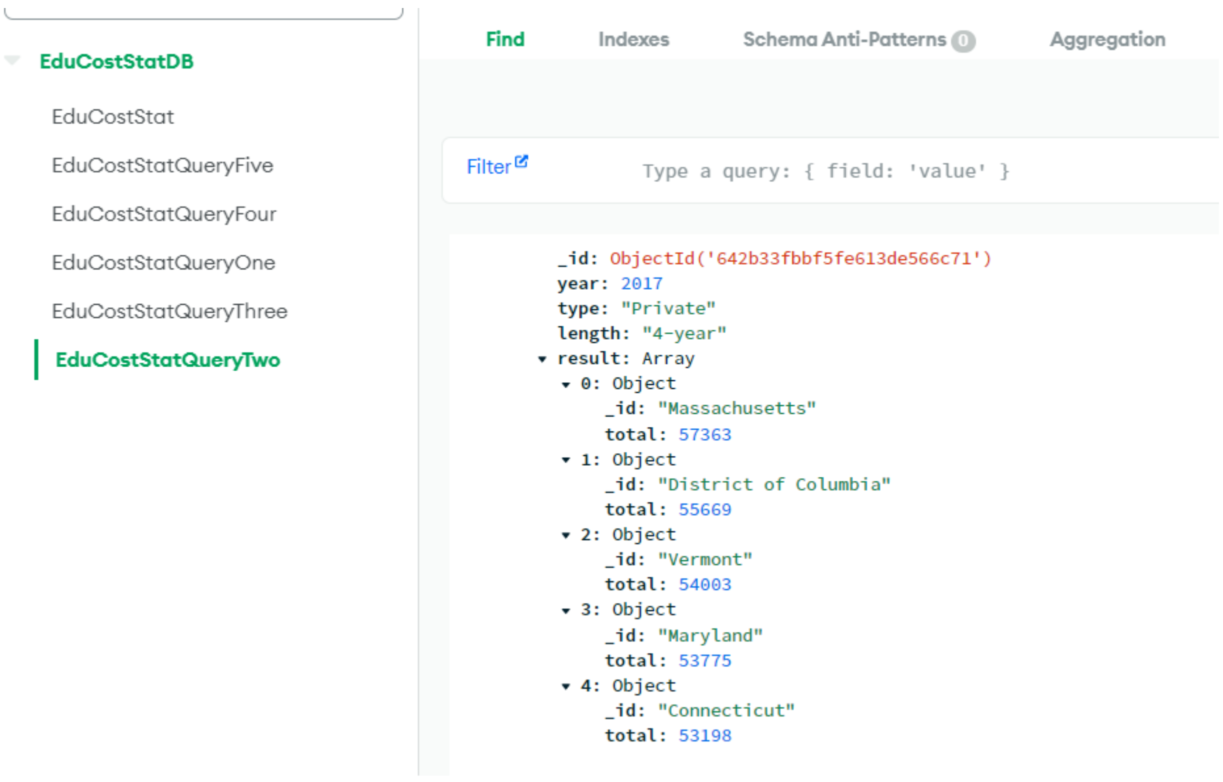
Testing QueryTwo thorugh gRPC:



```
{
  "year": 2017,
  "type": "Private",
  "length": "4-year"
}
```

```
{
  "expensiveStates": [
    {
      "state": "Massachusetts",
      "total": 57363
    },
    {
      "state": "District of Columbia",
      "total": 55669
    },
    {
      "state": "Vermont",
      "total": 54003
    },
    {
      "state": "Maryland",
      "total": 53775
    },
    {
      "state": "Connecticut",
      "total": 53198
    }
  ]
}
```

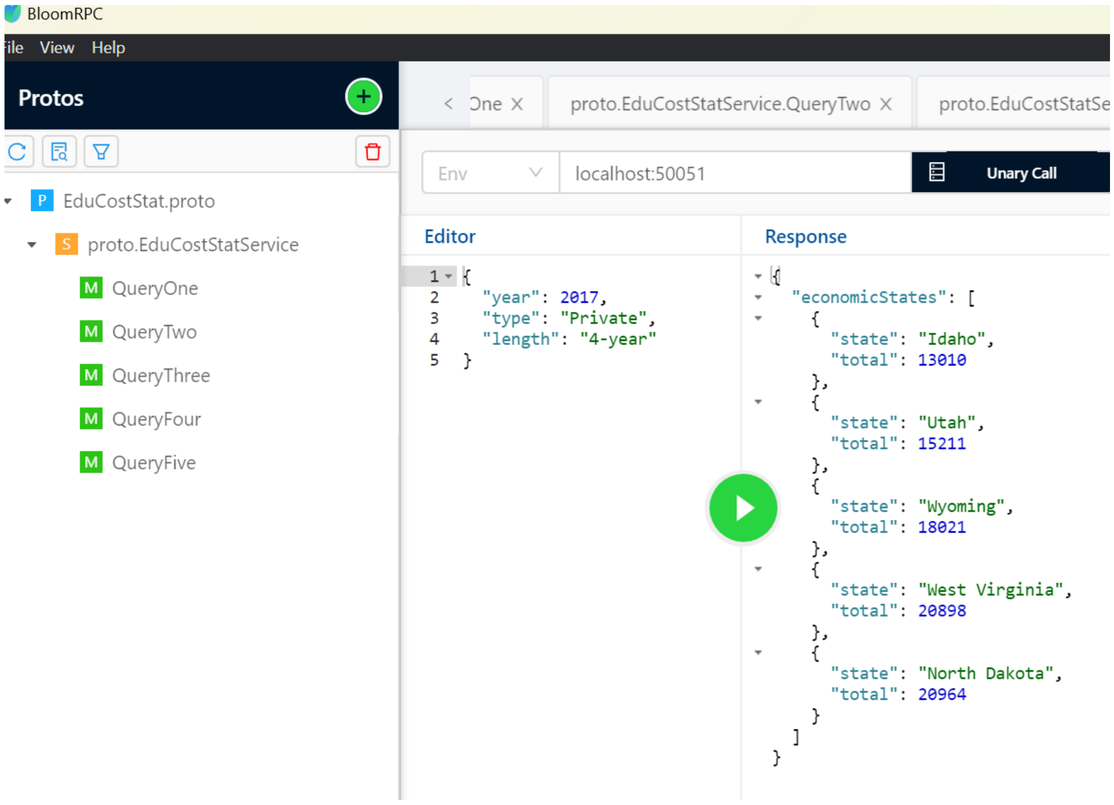
Figure 3.4: Client test Query One



```
_id: ObjectId('642b33fbbf5fe613de566c71')
year: 2017
type: "Private"
length: "4-year"
▾ result: Array
  ▾ 0: Object
      _id: "Massachusetts"
      total: 57363
  ▾ 1: Object
      _id: "District of Columbia"
      total: 55669
  ▾ 2: Object
      _id: "Vermont"
      total: 54003
  ▾ 3: Object
      _id: "Maryland"
      total: 53775
  ▾ 4: Object
      _id: "Connecticut"
      total: 53198
```

Figure 3.5: Query data saved in the collection

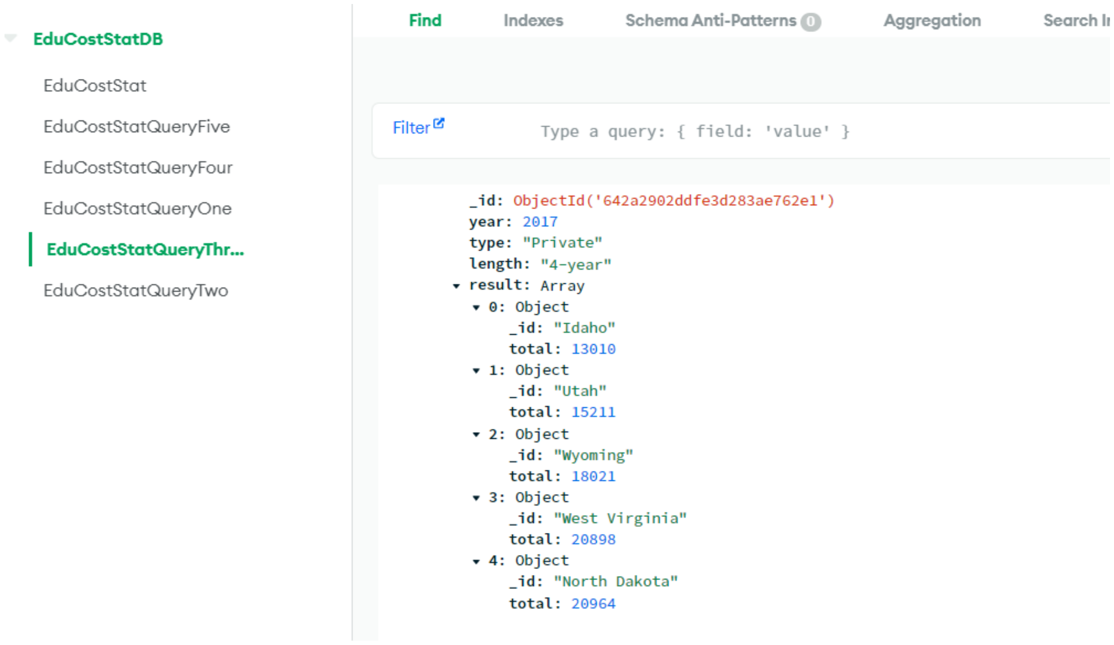
Testing QueryThree thorugh gRPC:



```
{
  "year": 2017,
  "type": "Private",
  "length": "4-year"
}
```

```
{
  "economicStates": [
    {
      "state": "Idaho",
      "total": 13010
    },
    {
      "state": "Utah",
      "total": 15211
    },
    {
      "state": "Wyoming",
      "total": 18021
    },
    {
      "state": "West Virginia",
      "total": 20898
    },
    {
      "state": "North Dakota",
      "total": 20964
    }
  ]
}
```

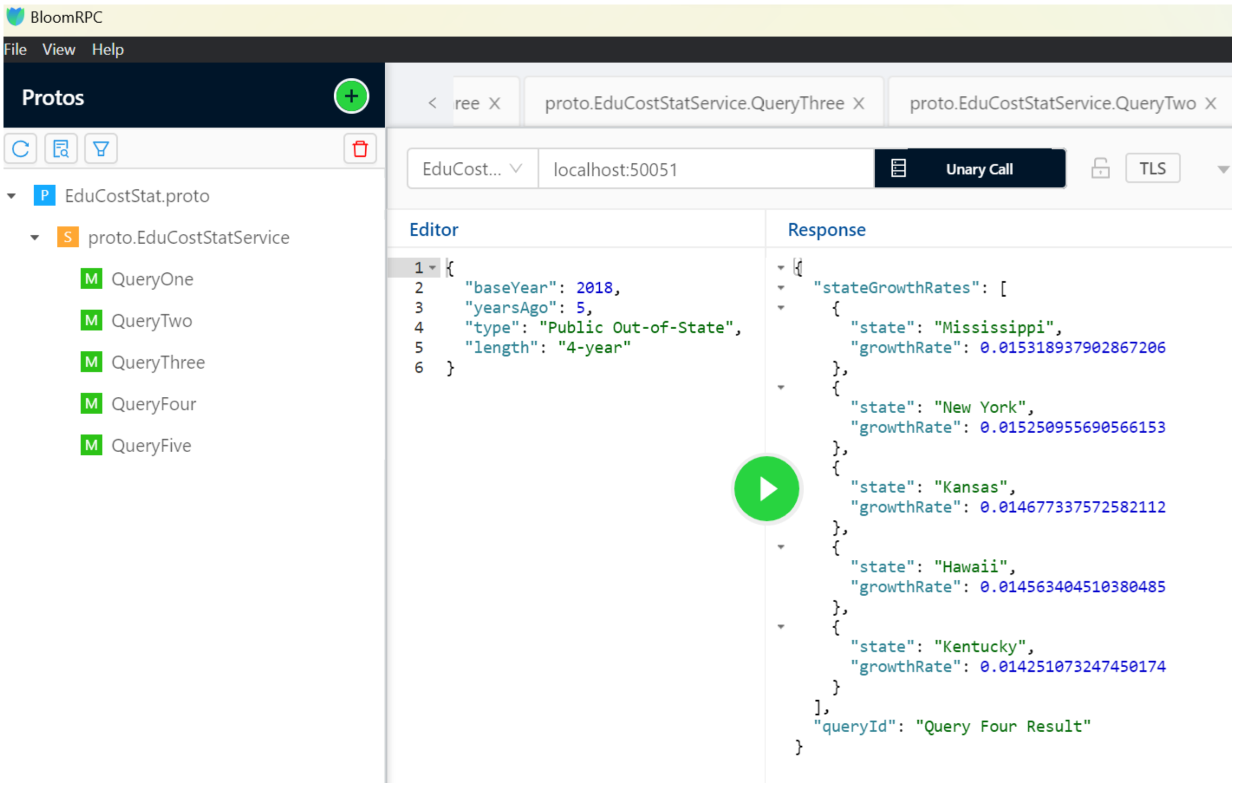
Figure 3.6: Client test Query Three



```
_id: ObjectId('642a2902ddfe3d283ae762e1')
year: 2017
type: "Private"
length: "4-year"
result: Array
  0: Object
    _id: "Idaho"
    total: 13010
  1: Object
    _id: "Utah"
    total: 15211
  2: Object
    _id: "Wyoming"
    total: 18021
  3: Object
    _id: "West Virginia"
    total: 20898
  4: Object
    _id: "North Dakota"
    total: 20964
```

Figure 3.7: Query data saved in the collection

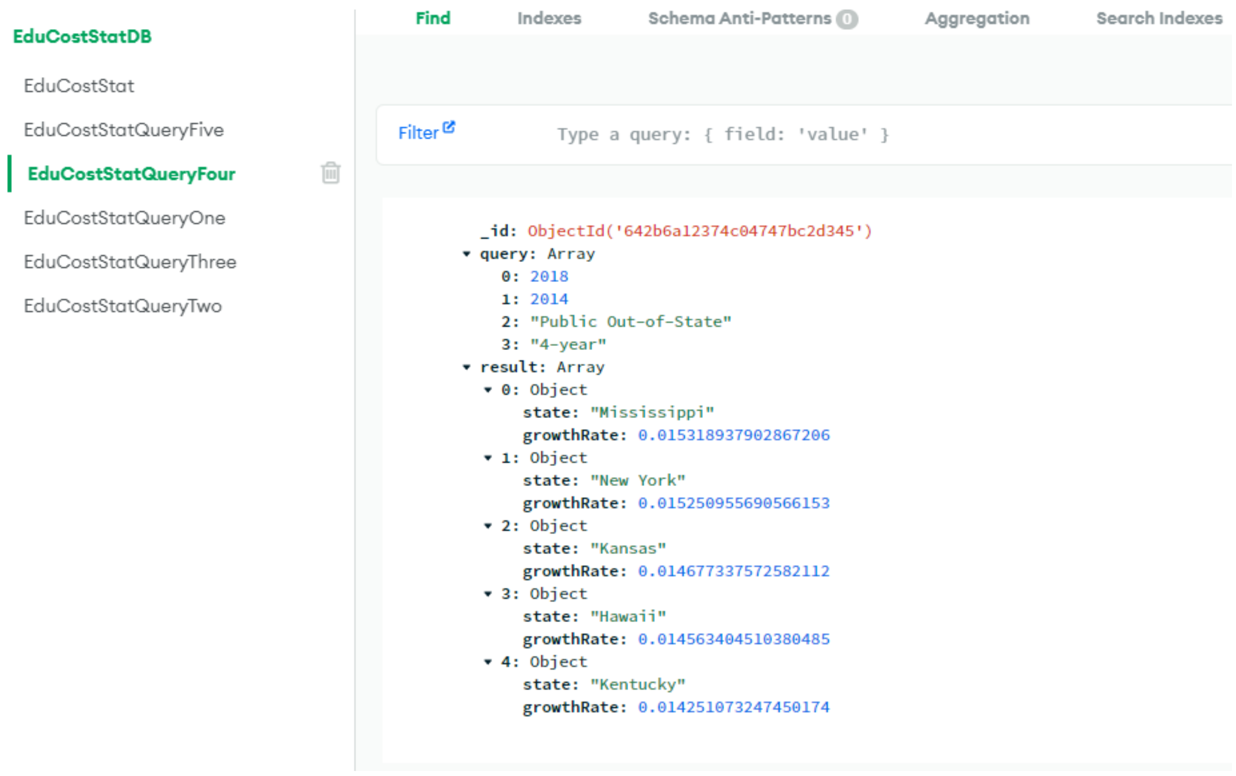
Testing QueryFour thorugh gRPC:



```
{
  "baseYear": 2018,
  "yearsAgo": 5,
  "type": "Public Out-of-State",
  "length": "4-year"
}
```

```
{
  "stateGrowthRates": [
    {
      "state": "Mississippi",
      "growthRate": 0.015318937902867206
    },
    {
      "state": "New York",
      "growthRate": 0.015250955690566153
    },
    {
      "state": "Kansas",
      "growthRate": 0.014677337572582112
    },
    {
      "state": "Hawaii",
      "growthRate": 0.014563404510380485
    },
    {
      "state": "Kentucky",
      "growthRate": 0.014251073247450174
    }
  ],
  "queryId": "Query Four Result"
}
```

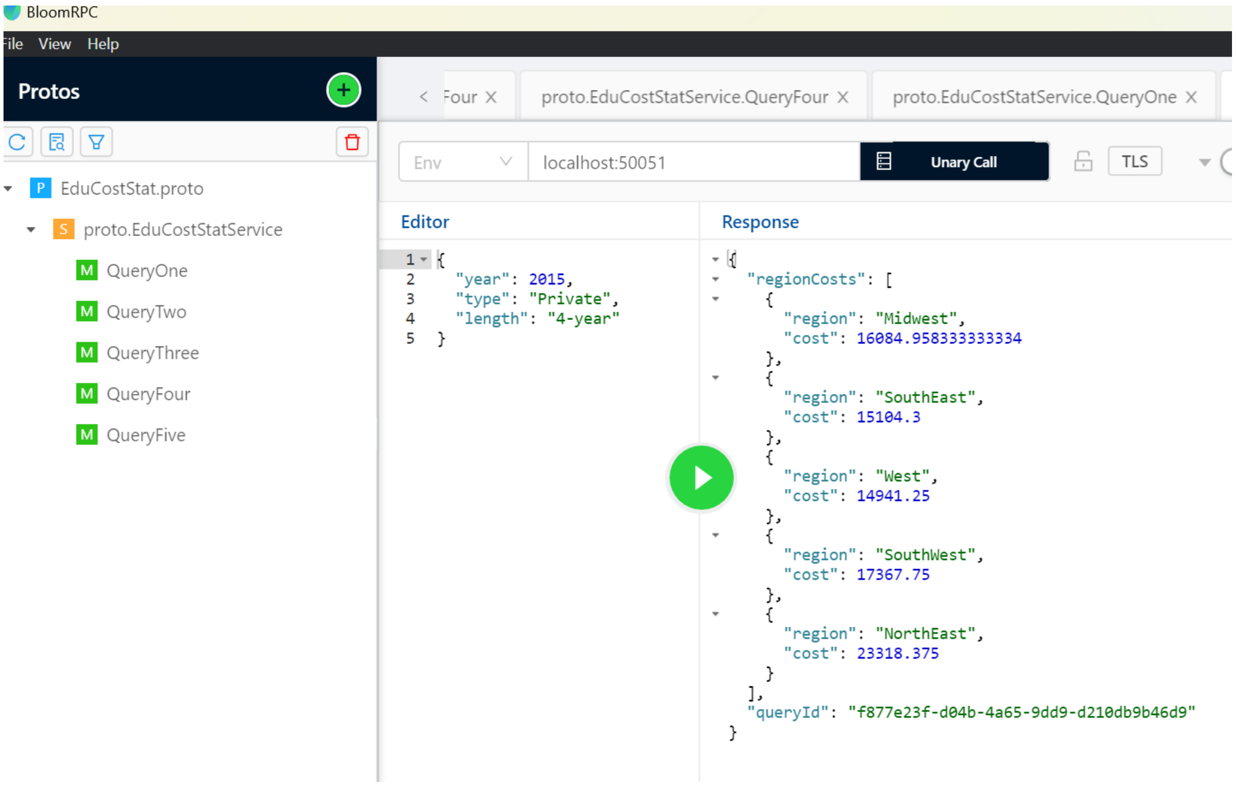
Figure 3.8: Client test Query Four



```
_id: ObjectId('642b6a12374c04747bc2d345')
query: Array
  0: 2018
  1: 2014
  2: "Public Out-of-State"
  3: "4-year"
result: Array
  0: Object
    state: "Mississippi"
    growthRate: 0.015318937902867206
  1: Object
    state: "New York"
    growthRate: 0.015250955690566153
  2: Object
    state: "Kansas"
    growthRate: 0.014677337572582112
  3: Object
    state: "Hawaii"
    growthRate: 0.014563404510380485
  4: Object
    state: "Kentucky"
    growthRate: 0.014251073247450174
```

Figure 3.9: Query data saved in the collection

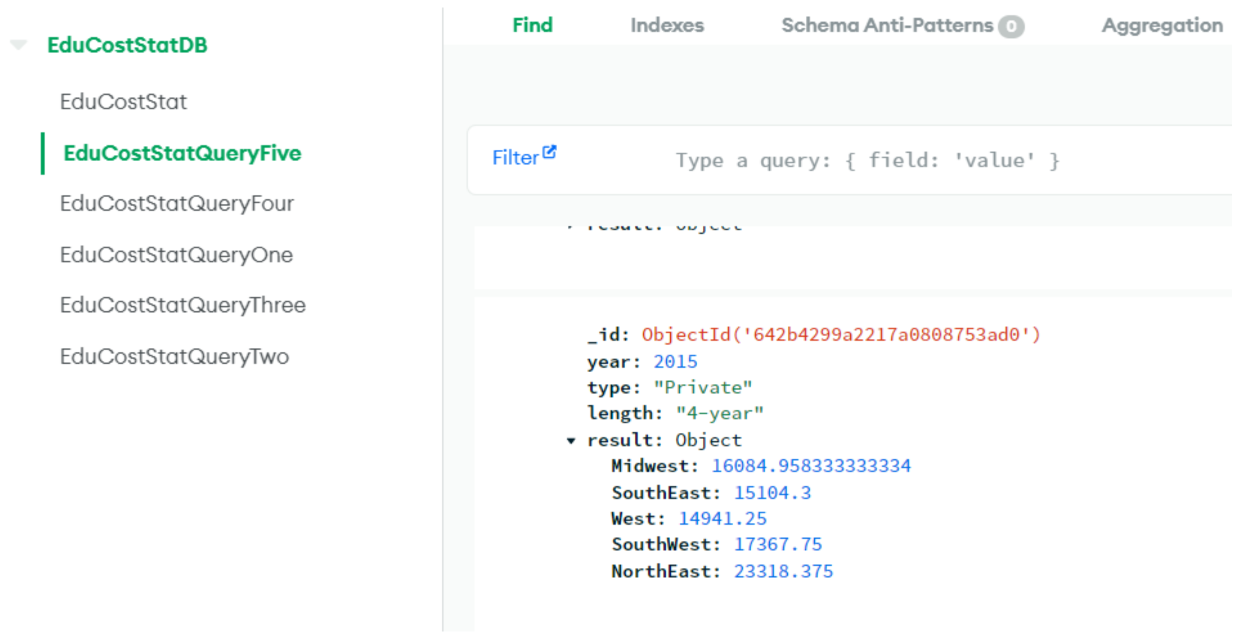
Testing QueryFive thorugh gRPC:



```
{
  "year": 2015,
  "type": "Private",
  "length": "4-year"
}
```

```
{
  "regionCosts": [
    {
      "region": "Midwest",
      "cost": 16084.958333333334
    },
    {
      "region": "SouthEast",
      "cost": 15104.3
    },
    {
      "region": "West",
      "cost": 14941.25
    },
    {
      "region": "SouthWest",
      "cost": 17367.75
    },
    {
      "region": "NorthEast",
      "cost": 23318.375
    }
  ],
  "queryId": "f877e23f-d04b-4a65-9dd9-d210db9b46d9"
}
```

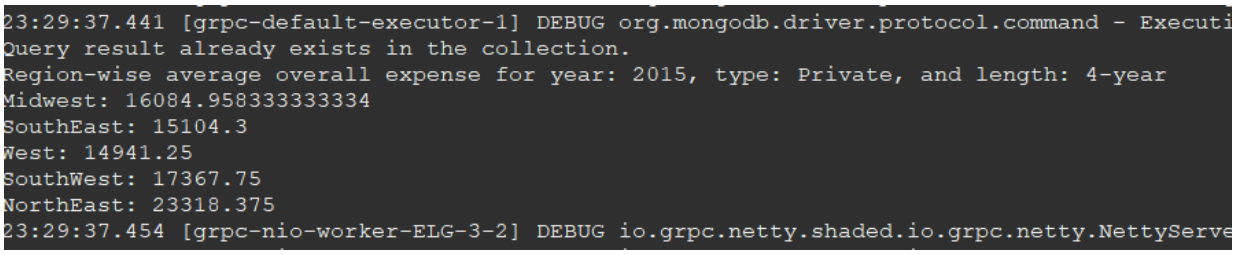
Figure 3.10: Client test Query Five



```
_id: ObjectId('642b4299a2217a0808753ad0')
year: 2015
type: "Private"
length: "4-year"
▾ result: Object
    Midwest: 16084.958333333334
    SouthEast: 15104.3
    West: 14941.25
    SouthWest: 17367.75
    NorthEast: 23318.375
```

Figure 3.11: Query data saved in the collection

When the data already exists in the collection, the program will print the message ”Query result already exists in the collection.” The query will not be executed and the existing result will be returned without being saved in the collection.



```
23:29:37.441 [grpc-default-executor-1] DEBUG org.mongodb.driver.protocol.command - Executi
Query result already exists in the collection.
Region-wise average overall expense for year: 2015, type: Private, and length: 4-year
Midwest: 16084.958333333334
SouthEast: 15104.3
West: 14941.25
SouthWest: 17367.75
NorthEast: 23318.375
23:29:37.454 [grpc-nio-worker-ELG-3-2] DEBUG io.grpc.netty.shaded.io.grpc.netty.NettyServe
```

Figure 3.12: Query data already exist in the collection

4 Conclusion

In conclusion, the Education Cost Statistics application provides an efficient way to analyze the cost of education in the United States. The five implemented queries cover a wide range of analysis, from determining the highest and lowest expense states to finding the region-wise average overall expense.

Each query is implemented using MongoDB and Java, with the data stored in a MongoDB database. The queries use various aggregation pipelines to filter and aggregate the data based on the specified parameters.

Overall, the application provides a useful tool for analyzing the cost of education in the United States and can be extended to include more queries and analysis in the future.

References

[1] MongoDB. (2021) Mongodb documentation. [Online]. Available: https://docs.mongodb.com/

[2] Google, “Aggregation pipeline - java,” https://mongodb.github.io/mongo-java-driver/3.12/driver/tutorials/aggregation/, 2021.

[3] Google, “Protocol Buffers,” https://developers.google.com/protocol-buffers, 2022.

[4] J. Li. “Bloom rpc,” ,. [Online]. Available: https://github.com/bloomrpc/bloomrpc/releases