COEN 6731: Distributed Software Systems

Assignment – II

Submitted to – YAN LIU

Submitted by –Rohan K (40196377)

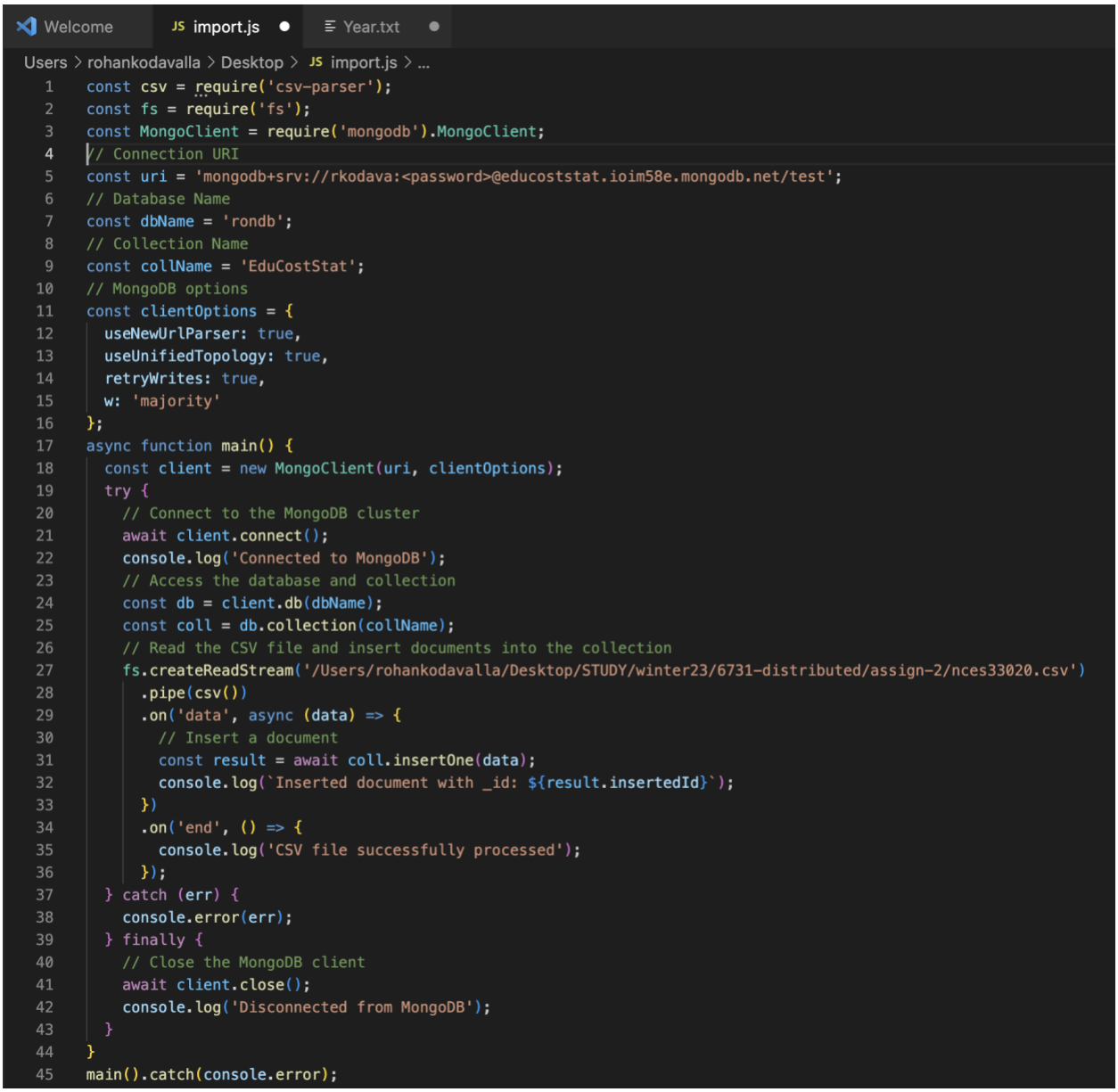
Date of submission – April 3, 2023

Task 1. MongoDB Data Storage and Operation

Task 1.1) To create a MongoDB collection named EduCostStat to store the data to the MongoDB instance running on MongoDB online cluster.

Below are the flow of operations I did to achieve this –

1. Created Mongo DB Atlas account
2. Downloaded the dataset as a .csv file.
3. Coded a program to create a collection by reading the data samples from file. The database named "rondb" and cluster "EduCostStat" is created. (snippet of code is below)



Welcome | import.js | Year.txt

Users > rohankodavalla > Desktop > import.js > ...

```
const csv = require('csv-parser');
const fs = require('fs');
const MongoClient = require('mongodb').MongoClient;
// Connection URI
const uri = 'mongodb+srv://rkodava:<password>@educoststat.ioim58e.mongodb.net/test';
// Database Name
const dbName = 'rondb';
// Collection Name
const collName = 'EduCostStat';
// MongoDB options
const clientOptions = {
  useNewUrlParser: true,
  useUnifiedTopology: true,
  retryWrites: true,
  w: 'majority'
};
async function main() {
  const client = new MongoClient(uri, clientOptions);
  try {
    // Connect to the MongoDB cluster
    await client.connect();
    console.log('Connected to MongoDB');
    // Access the database and collection
    const db = client.db(dbName);
    const coll = db.collection(collName);
    // Read the CSV file and insert documents into the collection
    fs.createReadStream('/Users/rohankodavalla/Desktop/STUDY/winter23/6731-distributed/assign-2/nces33020.csv')
      .pipe(csv())
      .on('data', async (data) => {
        // Insert a document
        const result = await coll.insertOne(data);
        console.log(`Inserted document with _id: ${result.insertedId}`);
      })
      .on('end', () => {
        console.log('CSV file successfully processed');
      });
  } catch (err) {
    console.error(err);
  } finally {
    // Close the MongoDB client
    await client.close();
    console.log('Disconnected from MongoDB');
  }
}
main().catch(console.error);
```

Below is the explanation of the code:

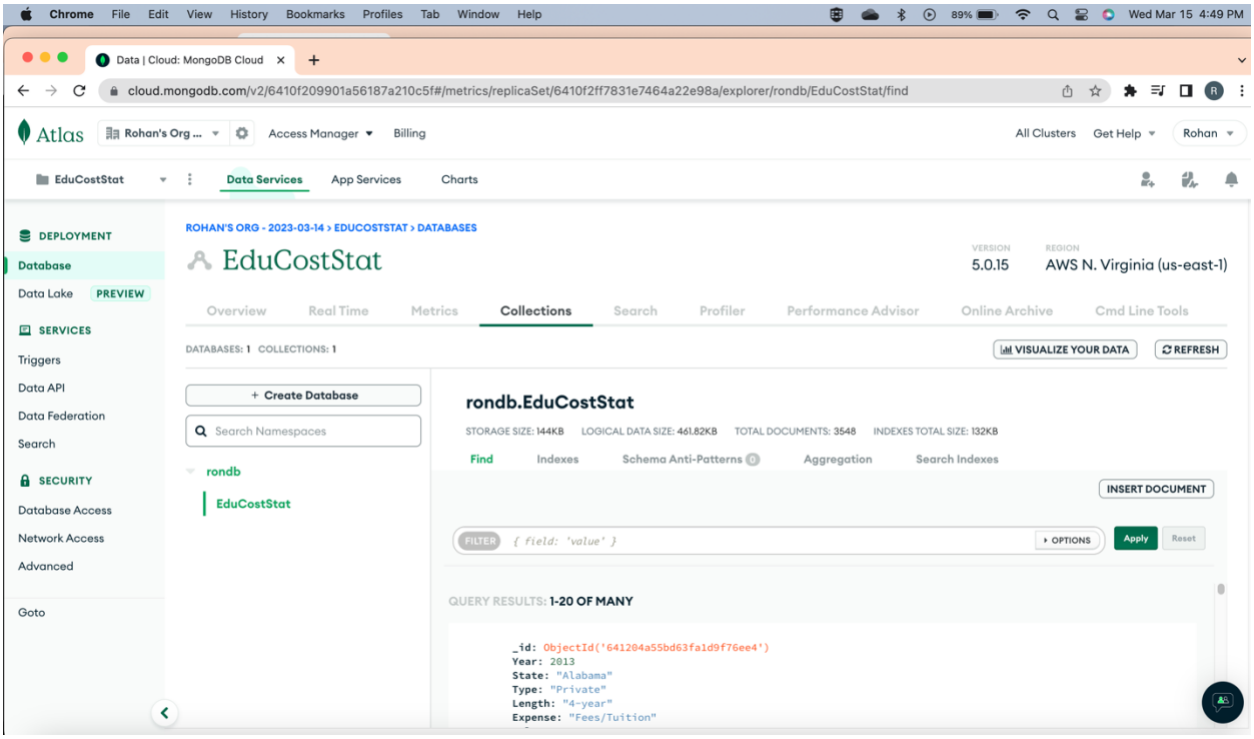
- The code is importing three dependencies: csv-parser, fs, and MongoClient from the mongodb library.
- A MongoDB connection URI is defined, which includes the username and password required to authenticate to the Atlas cluster.
- A database name and collection name are defined for the target MongoDB instance.
- MongoDB options are set, including the use of the useNewUrlParser, useUnifiedTopology, retryWrites, and w options.
- The main() function is defined as an asynchronous function.
- Within the main() function, a new instance of MongoClient is created using the URI and options defined earlier.
- An attempt is made to connect to the MongoDB cluster using the connect() method on the MongoClient instance. If successful, a message is printed to the console.
- The db and coll variables are used to reference the target database and collection.
- A CSV file is read using fs.createReadStream() and piped through csv-parser to parse each row as an object.
- For each row of data in the CSV file, a new document is inserted into the MongoDB collection using coll.insertOne(). The inserted document ID is logged to the console.
- After all rows of data have been processed, a message is logged to the console.
- If an error occurs during any of the above operations, it is logged to the console.
- The MongoDB client connection is closed using client.close(), and a message is printed to the console indicating that the connection has been closed.
- The main() function is called, and any errors encountered during its execution are logged to the console.

4. Further, navigated to the directory where the .js file is located in my terminal and use the following command: “node import.js”.

   Below are the steps achieved by the code –

   - If the connection to the MongoDB cluster is successful, the console will log "Connected to MongoDB".
   - The code will read the specified CSV file and insert the data as documents into the specified MongoDB collection.
   - As each document is inserted, the console will log a message indicating the inserted document's ID.
   - Once all documents have been inserted, the console will log "CSV file successfully processed".
   - If there are any errors, the console will log the error message.
   - Finally, the console will log "Disconnected from MongoDB" when the connection is closed.

   The dashboard of MongoDB is below



   As shown, db name = rondb, collection = EduCostStat , with 3548 documents. The next 5 queries have been done in db = test which is a duplicate of rondb, in order to avoid any impact of huge mistakes.

Task1.2)

1) **Query the cost given specific year, state, type, length, expense; and save the query as a document in a collection named EduCostStatQueryOne :**

To develop data access objects in Java that represent different queries to the data and save the query as a document in a collection named EduCostStatQueryOne, these steps were done:

1. Added the MongoDB Java driver dependency to the project.
2. Created a Java class named EduCostStatQueryOne and define instance variables for year, state, type, length, and expense.
3. Create a constructor for the EduCostStatQueryOne class that initializes the instance variables.
4. Define a method named queryCost() in the EduCostStatQueryOne class that performs the query on the EduCostStat collection in MongoDB Atlas and saves the query result as a document in the EduCostStatQueryOne collection.
5. In the queryCost () method, the MongoDB Java driver is used to connect to the MongoDB Atlas cluster, access the EduCostStat and EduCostStatQueryOne collections, and perform the query using the instance variables.
6. Checks if the query result already exists in the EduCostStatQueryOne collection before inserting a new document to avoid duplicates and inserts it into the EduCostStatQueryOne collection.
7. Close the connection to the MongoDB Atlas cluster.

Here's the implementation of the EduCostStatQueryOne:

src > main > java > com > assign > app > J EduCostStatQueryOne.java > EduCostStatQueryOne > queryCost(int, String, String, String, String, MongoCollection<Document>, MongoCollection<Document>)

```java
package com.assign.app;

import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Filters;
import org.bson.Document;


public class EduCostStatQueryOne {

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

    Run | Debug
    public static void main(String[] args) {
        // Set up MongoDB connection
        MongoClient mongoClient = MongoClients.create(connectionString:"mongodb+srv://rkodava:Dimpu1997@educoststat.ioim58e.mongodb.net/?retryWrites=true&w=majority");
        MongoDatabase database = mongoClient.getDatabase(DB_NAME);
        MongoCollection<Document> collection = database.getCollection(COLLECTION_NAME);
        MongoCollection<Document> queryCollection = database.getCollection(QUERY_COLLECTION_NAME);

        // Call the query method
        int year = 2013;
        String state = "Alabama";
        String type = "Private";
        String length = "4-year";
        String expense = "Fees/Tuition";
        Document queryResult = queryCost(year, state, type, length, expense, collection, queryCollection);

        // Print the query result
        System.out.println(queryResult.toJson());

        // Close MongoDB connection
        mongoClient.close();
    }

    public static Document queryCost(int year, String state, String type, String length, String expense, MongoCollection<Document> collection, MongoCollection<Document> queryCollection) {
        // Check if the query already exists in the query collection
        Document queryDoc = queryCollection.find(Filters.and(
                Filters.eq(fieldName:"year", year),
                Filters.eq(fieldName:"state", state),
                Filters.eq(fieldName:"type", type),
                Filters.eq(fieldName:"length", length),
                Filters.eq(fieldName:"expense", expense)
        )).first();
        if (queryDoc != null) {
            return queryDoc;
        }

        // Query the cost from the EduCostStat collection
        Document costDoc = collection.find(Filters.and(
```

```java
        // Query the cost from the EduCostStat collection
        Document costDoc = collection.find(Filters.and(
                Filters.eq(fieldName:"year", year),
                Filters.eq(fieldName:"state", state),
                Filters.eq(fieldName:"type", type),
                Filters.eq(fieldName:"length", length)
        )).first();

        // Create a new query document with the query parameters and the query result
        Document queryResult = new Document()
                .append(key:"year", year)
                .append(key:"state", state)
                .append(key:"type", type)
                .append(key:"length", length)
                .append(key:"expense", expense)
                .append(key:"cost", costDoc.getDouble(expense));

                // Insert the new query document into the query collection
                queryCollection.insertOne(queryResult);

    return queryResult;
    }
}
```

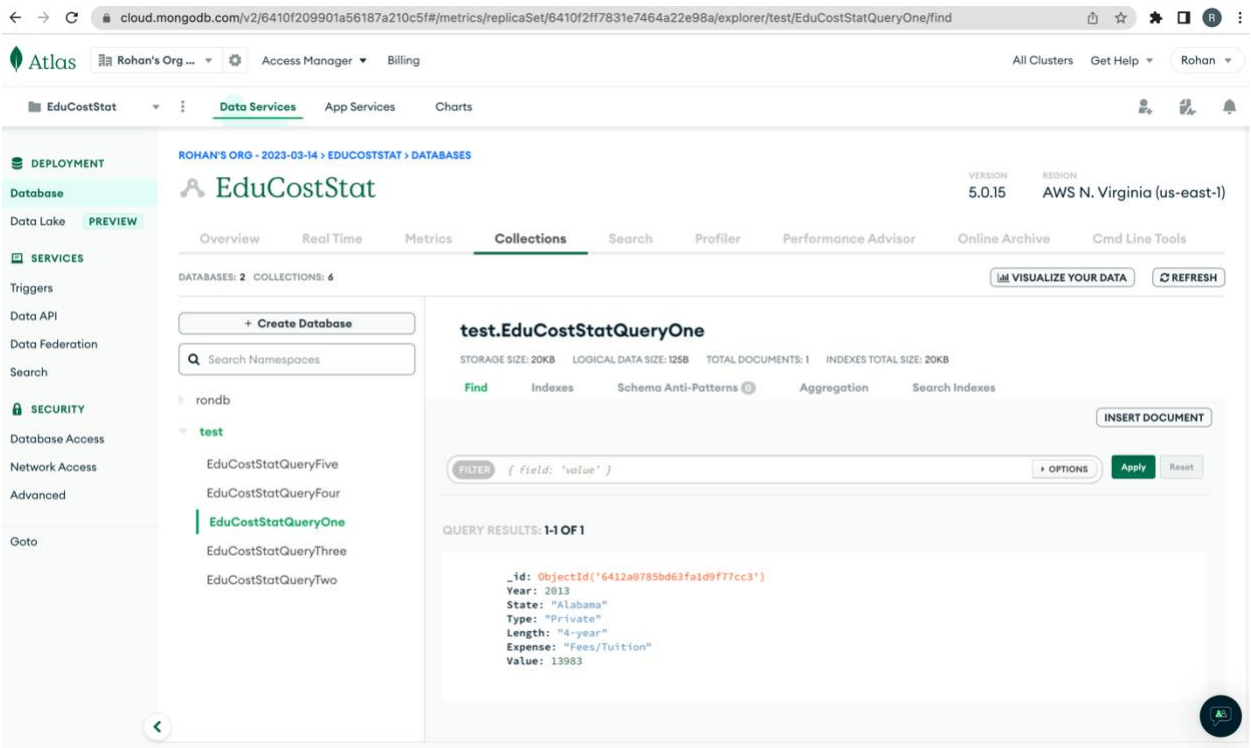
This Java code queries the education cost of a specific expense category in a given year, state, type, and length of education. The code connects to a MongoDB Atlas database and retrieves data from a collection named "EduCostStat".

The program defines a class called "EduCostStatQueryOne" with a "main" method. In the main method, the code establishes a connection to the MongoDB Atlas database, creates references to the "EduCostStat" collection and a new collection called "EduCostStatQueryOne" to store the query results.

Then the code calls a method "queryCost" which accepts the input parameters of year, state, type, length, expense, and references to the "EduCostStat" collection and "EduCostStatQueryOne" collection. The method checks if the query already exists in the "EduCostStatQueryOne" collection, and if so, returns the existing result document. If not, the method queries the cost of the specified expense category using the given input parameters from the "EduCostStat" collection, and creates a new document that contains the input parameters and the cost result. Finally, the method inserts the new document into the "EduCostStatQueryOne" collection.

In the main method, the code calls the "queryCost" method with the input parameters and retrieves the resulting document. The program prints the JSON representation of the resulting document to the console and then closes the MongoDB connection.

The result from the above code in MongoDB is as below :



2) Query the top 5 most expensive states (with overall expense) given a year, type, length; and save the query as a document in a collection named EduCostStatQueryTwo :

Below is a Java program that queries data from a MongoDB database and returns the top 5 most expensive states for a given year, type, and length of education program. The program uses the MongoDB Java driver to connect to the database and execute the query.

Here's a brief overview of the program's structure and function:

The program starts by importing several classes from the MongoDB Java driver and the standard Java library.

The program defines three constants: DB_NAME, COLLECTION_NAME, and QUERY_COLLECTION_NAME, which are used to specify the names of the database, collection, and query collection in the MongoDB database.

The main() method sets up a connection to the MongoDB database, retrieves the specified collection and query collection, and calls the queryTop5ExpensiveStates() method to execute the query.

The queryTop5ExpensiveStates() method first checks if the query has already been executed and saved in the query collection. If so, it returns the saved query result.

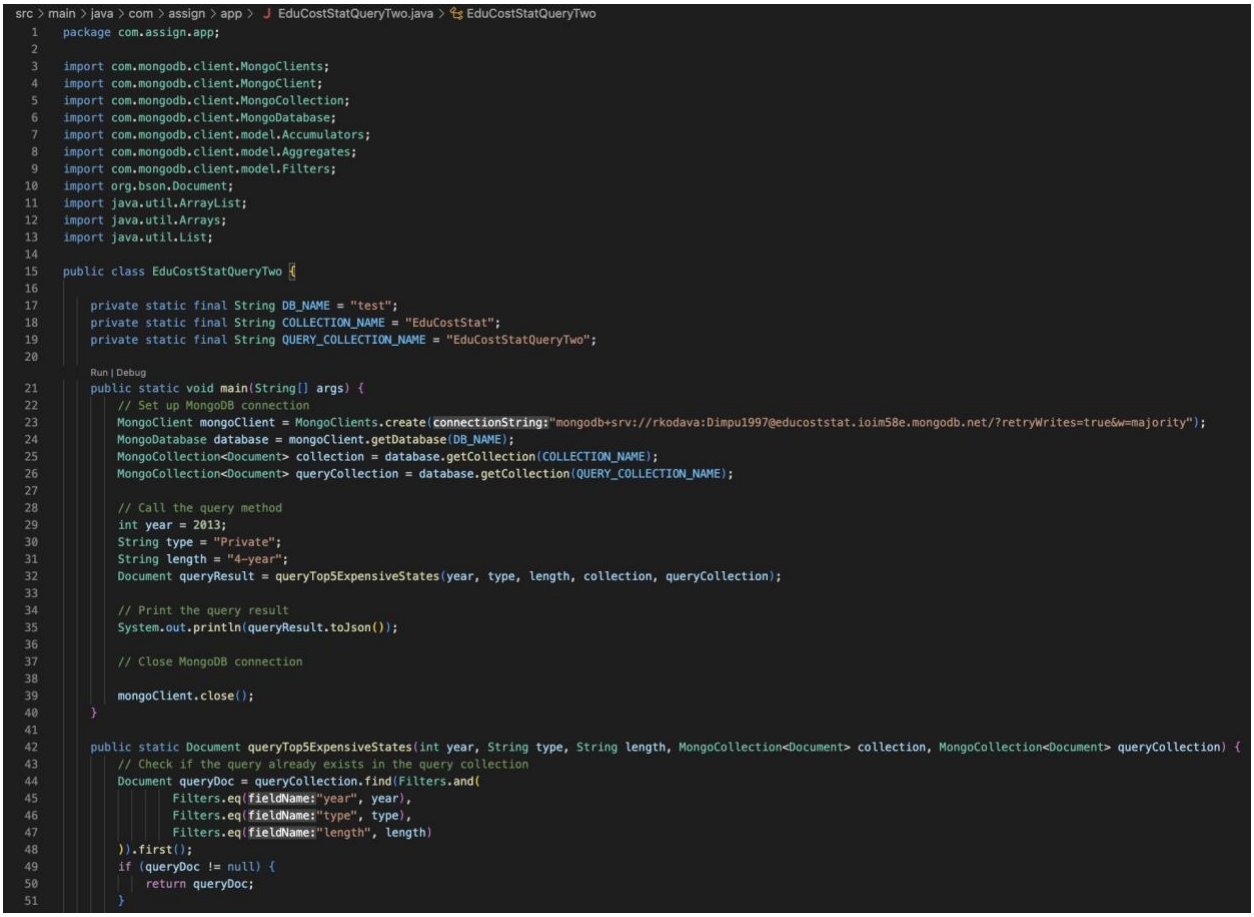
If the query has not been executed before, the method executes the query using the aggregate() method of the collection object. The query filters the data by year, type, and length, groups the data by state, calculates the total expense for each state, sorts the results in descending order of total expense, and limits the results to the top 5.

The query result is stored in a List<Object> object, which is then used to create a new query document that includes the query parameters and result.

The new query document is inserted into the query collection, and the method returns the document.

Finally, the main() method closes the MongoDB connection.

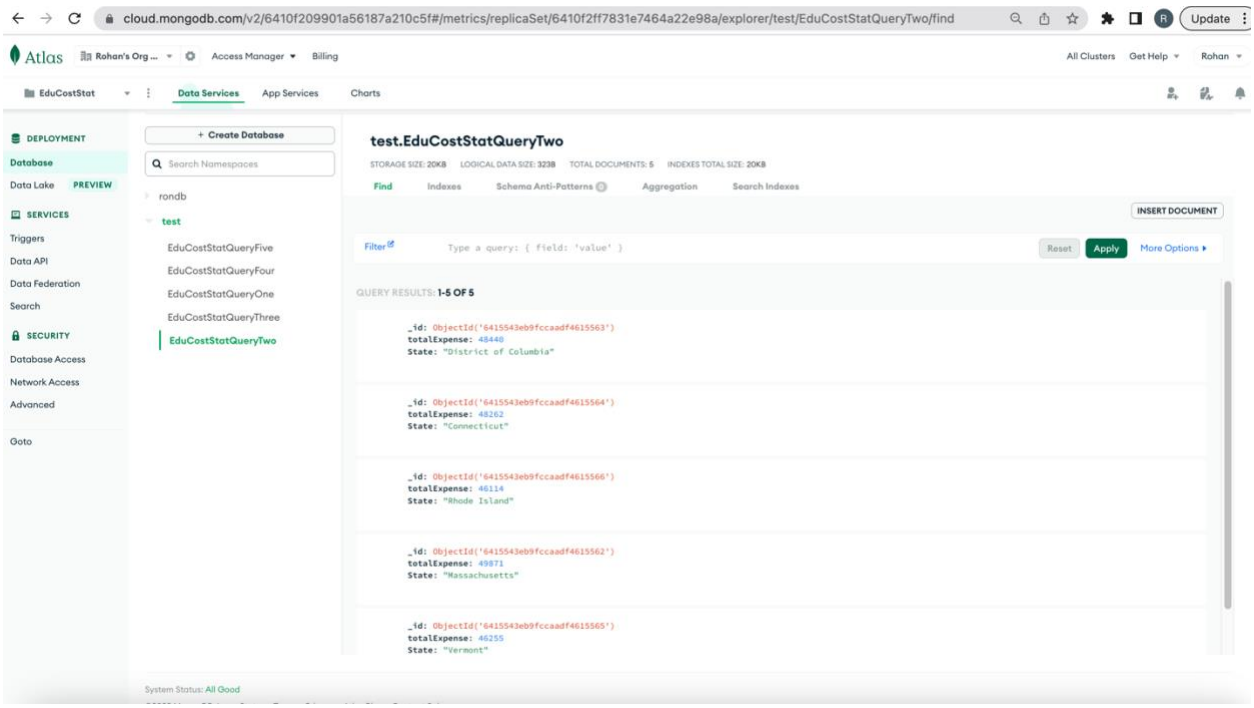
Here's an implementation of the EduCostStatQueryTwo:



```
src > main > java > com > assign > app > EduCostStatQueryTwo.java > EduCostStatQueryTwo
package com.assign.app;

import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Accumulators;
import com.mongodb.client.model.Aggregates;
import com.mongodb.client.model.Filters;
import org.bson.Document;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class EduCostStatQueryTwo {

    private static final String DB_NAME = "test";
    private static final String COLLECTION_NAME = "EduCostStat";
    private static final String QUERY_COLLECTION_NAME = "EduCostStatQueryTwo";

    Run | Debug
    public static void main(String[] args) {
        // Set up MongoDB connection
        MongoClient mongoClient = MongoClients.create(connectionString:"mongodb+srv://rkodava:Dimpu1997@educoststat.ioim58e.mongodb.net/?retryWrites=true&w=majority");
        MongoDatabase database = mongoClient.getDatabase(DB_NAME);
        MongoCollection<Document> collection = database.getCollection(COLLECTION_NAME);
        MongoCollection<Document> queryCollection = database.getCollection(QUERY_COLLECTION_NAME);

        // Call the query method
        int year = 2013;
        String type = "Private";
        String length = "4-year";
        Document queryResult = queryTop5ExpensiveStates(year, type, length, collection, queryCollection);

        // Print the query result
        System.out.println(queryResult.toJson());

        // Close MongoDB connection

        mongoClient.close();
    }

    public static Document queryTop5ExpensiveStates(int year, String type, String length, MongoCollection<Document> collection, MongoCollection<Document> queryCollection) {
        // Check if the query already exists in the query collection
        Document queryDoc = queryCollection.find(Filters.and(
                Filters.eq(fieldName:"year", year),
                Filters.eq(fieldName:"type", type),
                Filters.eq(fieldName:"length", length)
        )).first();
        if (queryDoc != null) {
            return queryDoc;
        }
```

```
        // Query the top 5 most expensive states
        List<Object> queryResult = collection.aggregate(Arrays.asList(
            Aggregates.match(
                    Filters.and(
                            Filters.eq(fieldName:"year", year),
                            Filters.eq(fieldName:"type", type),
                            Filters.eq(fieldName:"length", length)
                    )
            ),
            Aggregates.group(
                    id:"$state",
                    Accumulators.sum(fieldName:"totalExpense", expression:"$Value")
            ),
            Aggregates.sort(
                    new Document(key:"totalExpense", -1)
            ),
            Aggregates.limit(limit:5)
    )).into(new ArrayList<>());

        // Create a new query document with the query parameters and the query result
        Document queryDocToInsert = new Document()
                .append(key:"year", year)
                .append(key:"type", type)
                .append(key:"length", length)
                .append(key:"result", queryResult);

        // Insert the query document into the query collection
        queryCollection.insertOne(queryDocToInsert);

        return queryDocToInsert;
    }
}
```

The result from the above code for given a year, type, length equals to 2013, Private, 4-year in MongoDB is as below :



From the result above the totalExpense for the states are also shown.

3) **<u>Query the top 5 most economic states (with overall expense) given a year, type, length; and save the query as a document in a collection named EduCostStatQueryThree</u>** :

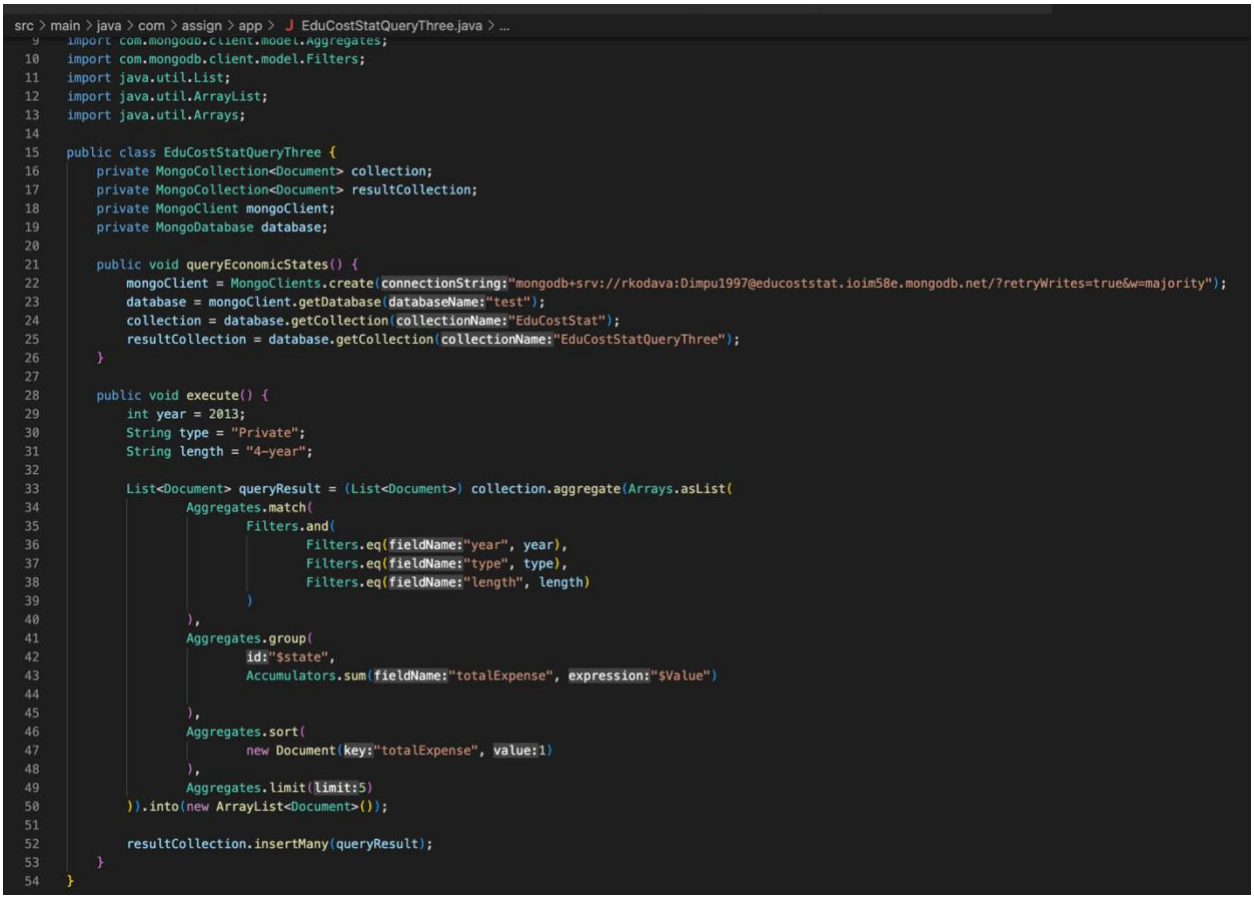
In this Java program, a query on a MongoDB database is performed using the aggregation pipeline. Specifically, it queries the "EduCostStat" collection to find the top 5 most economical states based on education expenses for a given year, type, and length.

The program first defines the values for the year, type, and length that it wants to query. It then creates a MongoDB client and connects to a database named "test" on the MongoDB Atlas cloud service. It selects the "EduCostStat" collection from the database and uses the aggregation pipeline to perform the query.

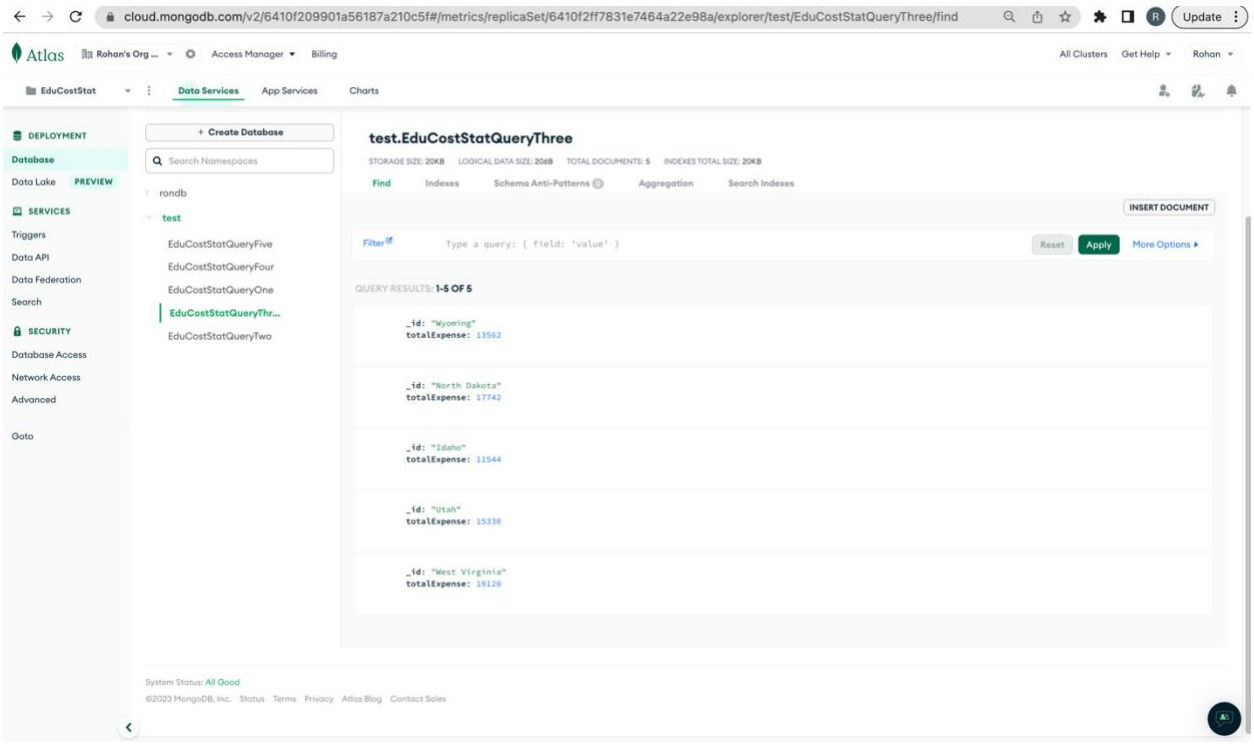
The aggregation pipeline consists of several stages, each defined using the Aggregates class from the MongoDB Java driver. The first stage uses the Filters class to match documents in the collection that have the specified year, type, and length. The second stage groups the matching documents by state and calculates the total expense for each state using the Accumulators class. The third stage sorts the resulting documents in ascending order of total expense. The fourth and final stage limits the output to the top 5 documents. The program then saves the query result as a

list of documents and inserts it into a new collection named "EduCostStatQueryThree" in the same MongoDB database.

Here's the implementation of the EduCostStatQueryThree:



The result from the above code for given a year, type, length equals to 2013, Private, 4-year in MongoDB is as below :



From the result above the totalExpense for the states are also shown.

4) <u>Query the top 5 states of the highest growth rate of overall expense given a range of past years, one year, three years and five years (using the latest year as the base) , type and length; and save the query as a document in a collection named EduCostStatQueryFour :</u>

This program connects to a MongoDB database and executes a query to find the top 5 states with the highest growth rate of education expenses over a period of time. The program has a main method that sets up the MongoDB connection and calls the queryTop5HighestGrowthRateStates method with the required parameters.

The queryTop5HighestGrowthRateStates method takes in several parameters such as the latest year, past years, type of education institution, length of education, the main collection to query, and the query collection to store the result. The method checks if the query already exists in the query collection using the query parameters.

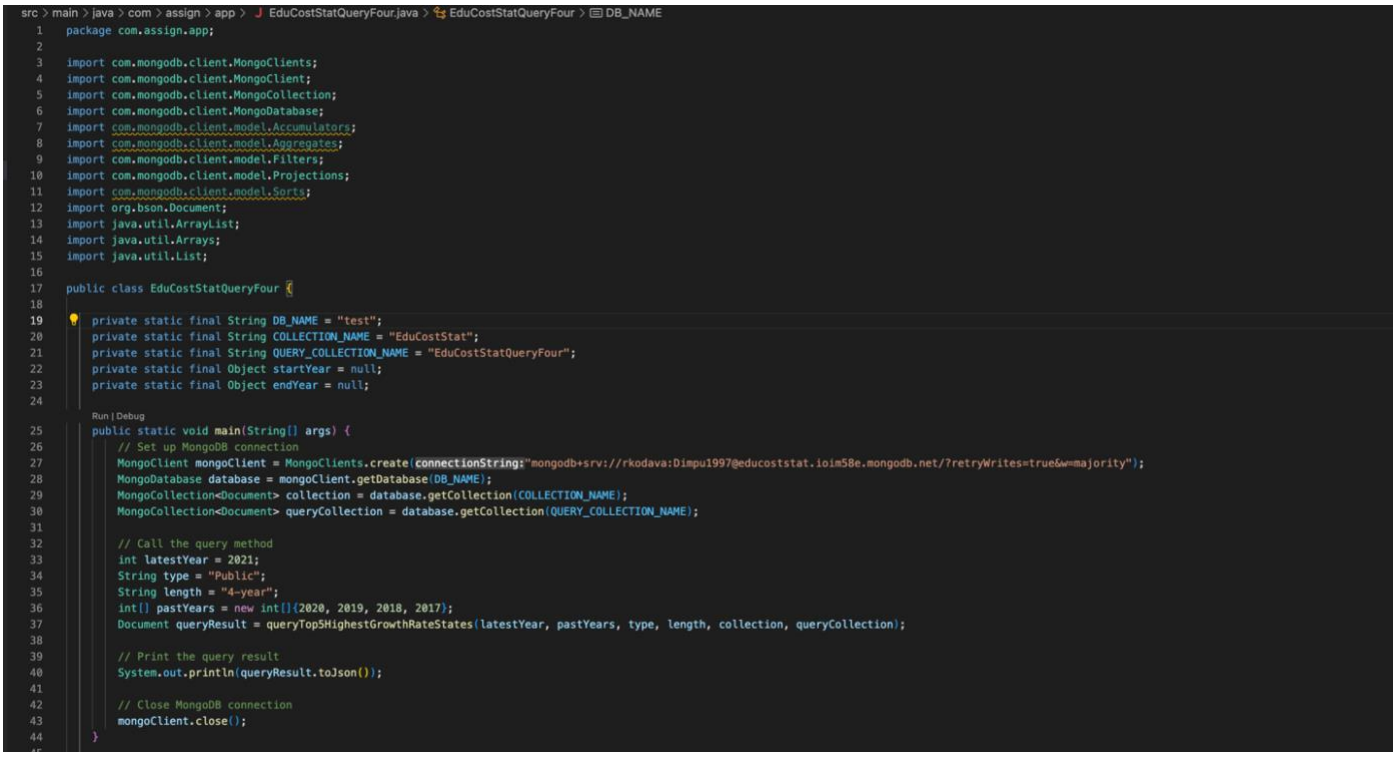
If it exists, the method returns the result from the query document. Otherwise, it calculates the expenses for the latest year and past years using the main collection and projects the results to

only include the state and the value of expenses. It then calculates the growth rate for each state by comparing the average expenses of each year and storing the growth rates in an array.

Next, the method extracts the top 5 states with the highest growth rate by iterating through the growth rate array and adding the states to a new document. The method also checks if the growth rate is NA and skips adding that state if it is. Finally, the method creates a new query document with the query parameters and the query result and inserts it into the query collection. The method then returns the query document.

The program also imports several classes from the com.mongodb.client package such as MongoClient, MongoDatabase, and MongoCollection to connect to the database and query the collections. It also imports the Document class from the org.bson package to work with BSON documents in MongoDB. Additionally, it imports classes from the com.mongodb.client.model package to use filters, projections, and sorts in the queries.

Here's the implementation of the EduCostStatQueryFour:

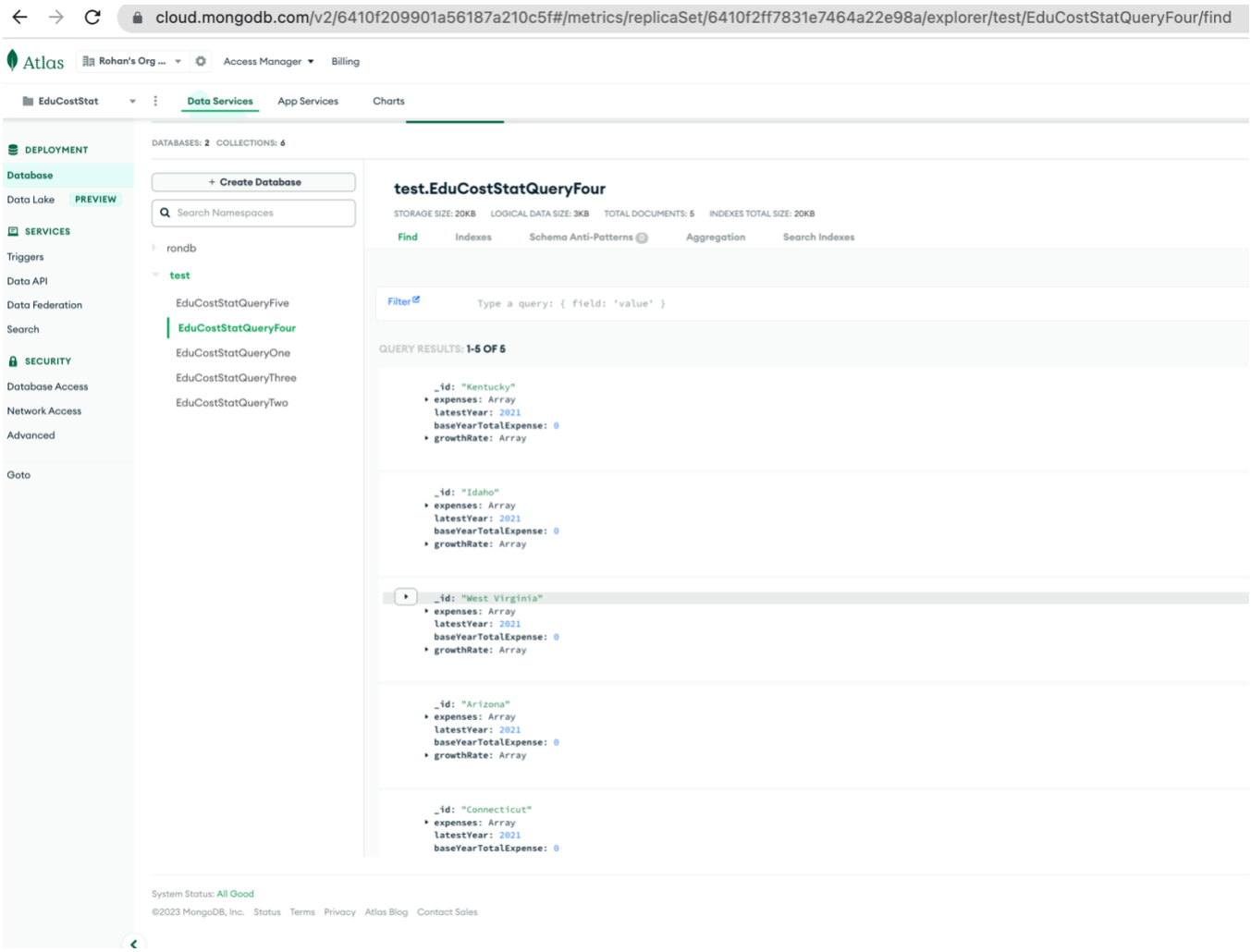


src > main > java > com > assign > app > EduCostStatQueryFour.java > EduCostStatQueryFour > DB_NAME

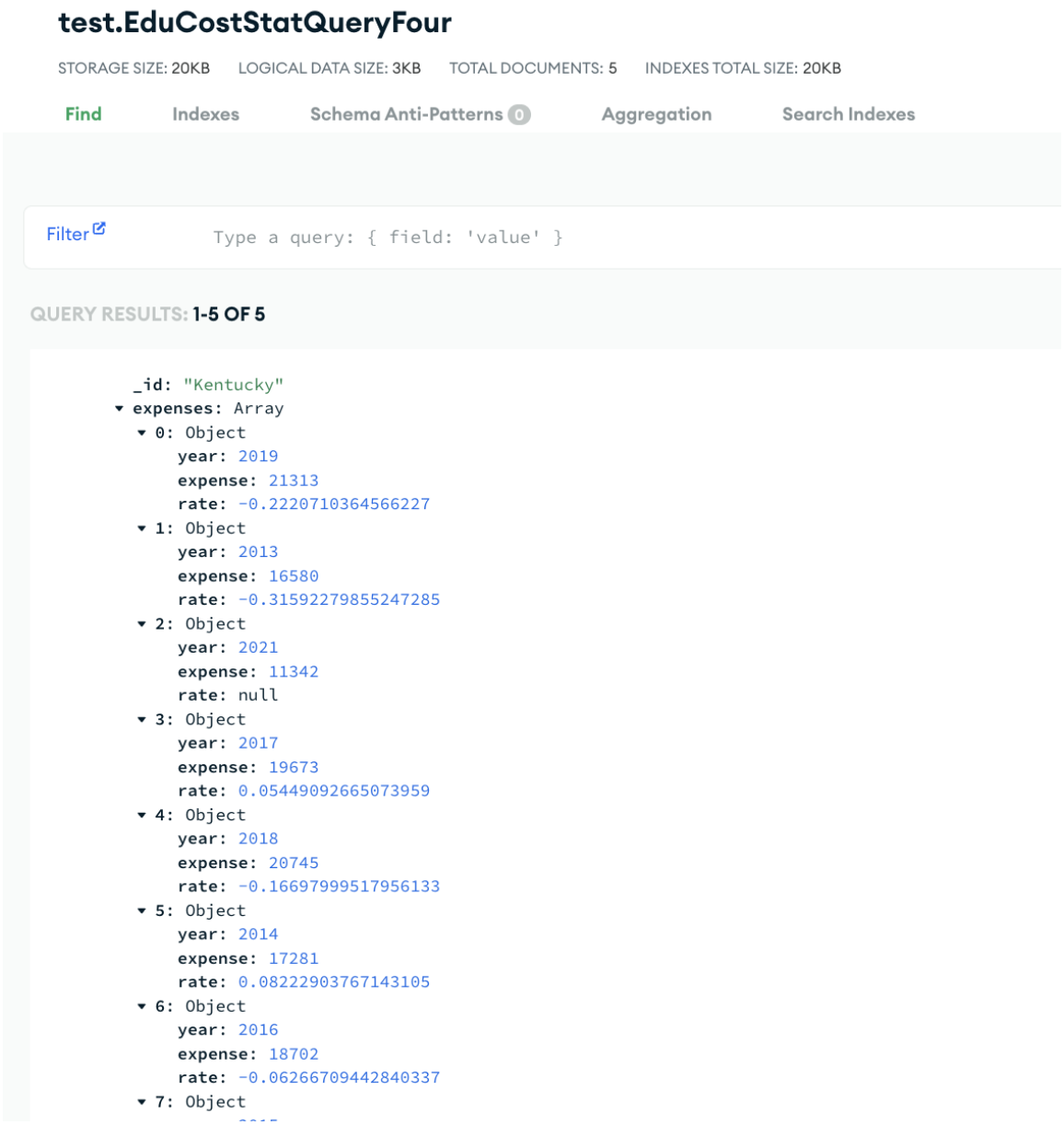
```java
public static Document queryTop5HighestGrowthRateStates(int latestYear, int[] pastYears, String type, String length, MongoCollection<Document> collection, MongoCollection<Document> queryCollection) {
    // Check if the query already exists in the query collection
    Document queryDoc = queryCollection.find(Filters.and(
            Filters.eq(fieldName:"latestYear", latestYear),
            Filters.eq(fieldName:"pastYears", Arrays.toString(pastYears)),
            Filters.eq(fieldName:"type", type),
            Filters.eq(fieldName:"length", length)
    )).first();
    if (queryDoc != null) {
        return queryDoc;
    }

    // Calculate the expenses for the latest year and past years
    List<Document> expenses = new ArrayList<>();
    for (int i = 0; i < pastYears.length; i++) {
        int year = pastYears[i];
        Document expense = collection.find(Filters.and(
                Filters.eq(fieldName:"year", year),
                Filters.eq(fieldName:"type", type),
                Filters.eq(fieldName:"length", length)
        )).projection(Projections.fields(Projections.excludeId(), Projections.include(...fieldNames:"state"), Projections.include(...fieldNames:"Value"))).first();
        if (expense != null) {
            expenses.add(expense);
        }
    }
    Document latestExpense = collection.find(Filters.and(
            Filters.eq(fieldName:"year", latestYear),
            Filters.eq(fieldName:"type", type),
            Filters.eq(fieldName:"length", length)
    )).projection(Projections.fields(Projections.excludeId(), Projections.include(...fieldNames:"state"), Projections.include(...fieldNames:"Value"))).first();
    if (latestExpense != null) {
        expenses.add(latestExpense);
    }

    // Calculate the growth rate for each state
    List<Double> growthRates = new ArrayList<>();
    List<Document> growthRateResult = new ArrayList<>();
    for (int i = 0; i < growthRateResult.size(); i++) {
        Document stateDoc = (Document) growthRateResult.get(i);
        String state = stateDoc.getString(key:"_id");
        Double avgExpense = stateDoc.getDouble(key:"avgExpense");
        Double prevAvgExpense = 0.0;
        if (i > 0) {
            prevAvgExpense = ((Document) growthRateResult.get(i - 1)).getDouble(key:"avgExpense");
        }
        Double growthRate = ((avgExpense - prevAvgExpense) / prevAvgExpense) * 100;
        growthRates.add(growthRate);
    }
```

```java
        // Get the top 5 states with the highest growth rate
    List<Document> top5GrowthStates = new ArrayList<>();
    for (int i = 0; i < growthRates.size(); i++) {
        Double growthRate = growthRates.get(i);
        if (Double.isNaN(growthRate)) {
            continue;
        }
        String state = ((Document) growthRateResult.get(i)).getString(key:"_id");
        Document doc = new Document(key:"state", state)
                .append(key:"growthRate", growthRate);
        top5GrowthStates.add(doc);
        if (top5GrowthStates.size() == 5) {
            break;
        }
    }

    // Create a new query document with the query parameters and the query result
    Document queryDocToInsert = new Document()
            .append(key:"startYear", startYear)
            .append(key:"endYear", endYear)
            .append(key:"type", type)
            .append(key:"length", length)
            .append(key:"result", top5GrowthStates);

    // Insert the query document into the query collection
    queryCollection.insertOne(queryDocToInsert);

    return queryDocToInsert;
}
}
```

The result from the above code for given latest year, past years, type of education institution, length of education equals to 2021, {2020, 2019, 2018, 2017}, Public , 4-year in MongoDB is as below –



It is also possible to see the individual growth rate for each year in each state as below -



test.EduCostStatQueryFour

STORAGE SIZE: 20KB LOGICAL DATA SIZE: 3KB TOTAL DOCUMENTS: 5 INDEXES TOTAL SIZE: 20KB

Find Indexes Schema Anti-Patterns 0 Aggregation Search Indexes

Filter Type a query: { field: 'value' }


```
        expense: 18702
        rate: -0.06266709442840337
      ▾ 7: Object
        year: 2015
        expense: 17530
        rate: 0.2435253850541928
      ▾ 8: Object
        year: 2020
        expense: 21799
        rate: null
    latestYear: 2021
    baseYearTotalExpense: 0
  ▾ growthRate: Array
    ▾ 0: Object
        year: 2017
        rate: 0.024026833279414787
    ▾ 1: Object
        year: 2018
        rate: 0.08769577609907349
    ▾ 2: Object
        year: 2019
        rate: 0.09042914531289471
    ▾ 3: Object
        year: 2020
        rate: 0.2435253850541928
```

```
  _id: "Idaho"
▸ expenses: Array
  latestYear: 2021
  baseYearTotalExpense: 0
▸ growthRate: Array
```

5) Aggregate region‘s average overall expense for a given year, type and length; and save the query as a document in a collection named EduCostStatQueryFive :

This code queries MongoDB for average education expenses in different regions of the United States.

The code first sets up a MongoDB connection using the MongoClient class from the MongoDB Java driver. It specifies the name of the database and collection to be used for the query, as well as the name of a separate collection to store the results of the query for future reference.

The program then calls the queryRegionAverageExpense method with the desired parameters for the query: year, type, length, and an array of regions. The method checks if the query has already been executed by searching the query collection for a document that matches the specified parameters. If such a document exists, it returns the query results from that document.

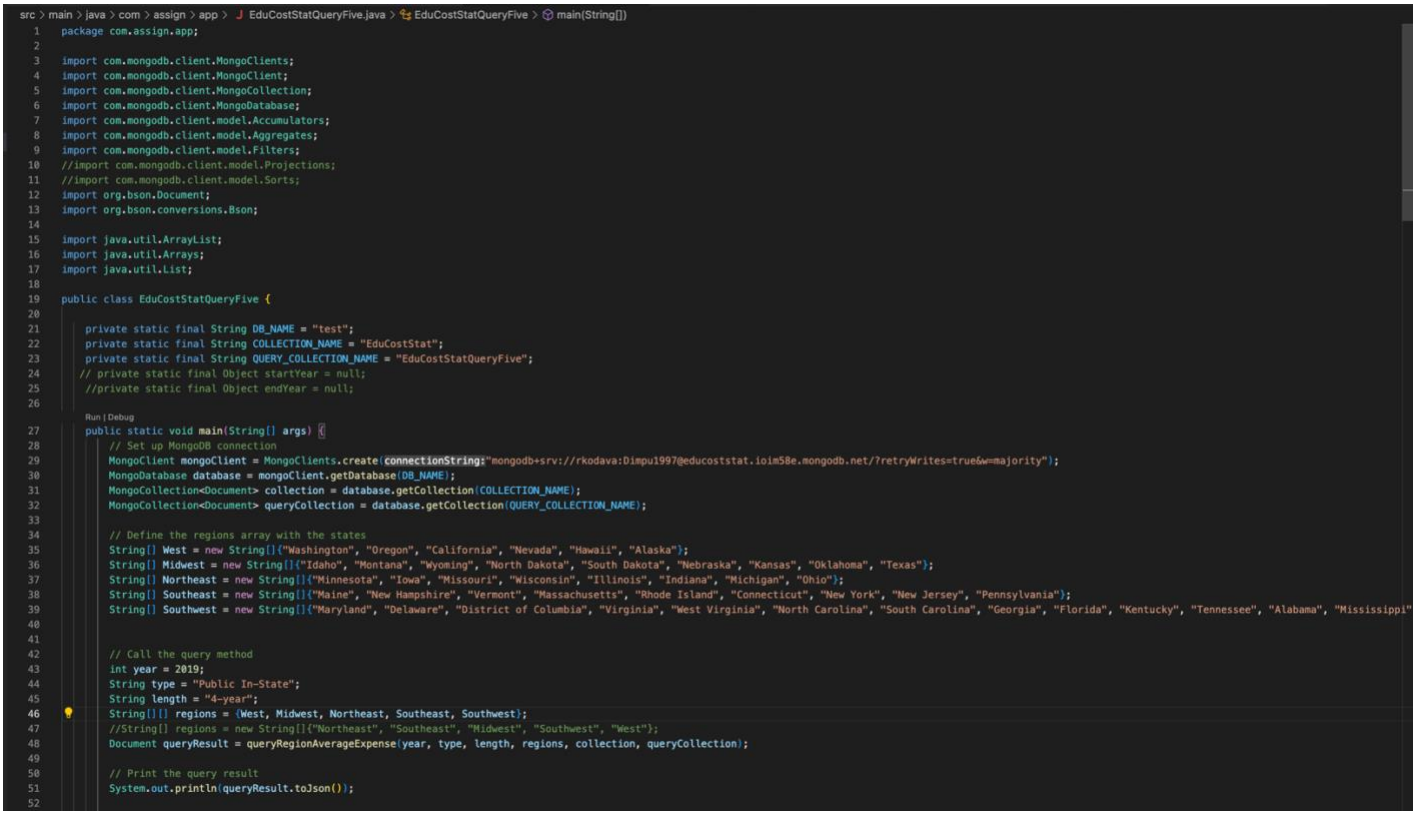
For regions, the 5 five regions - West, Midwest, Northeast, Southeast, Southwest are defined before the query is called.

If the query has not been executed before, the method constructs an aggregation pipeline using the Aggregates class from the MongoDB Java driver. The pipeline consists of two stages: a match stage that filters the documents based on the specified parameters, and a group stage that calculates the average education expenses for each region. The pipeline is executed using the aggregate method of the MongoCollection class, and the results are stored in a List of Document objects.

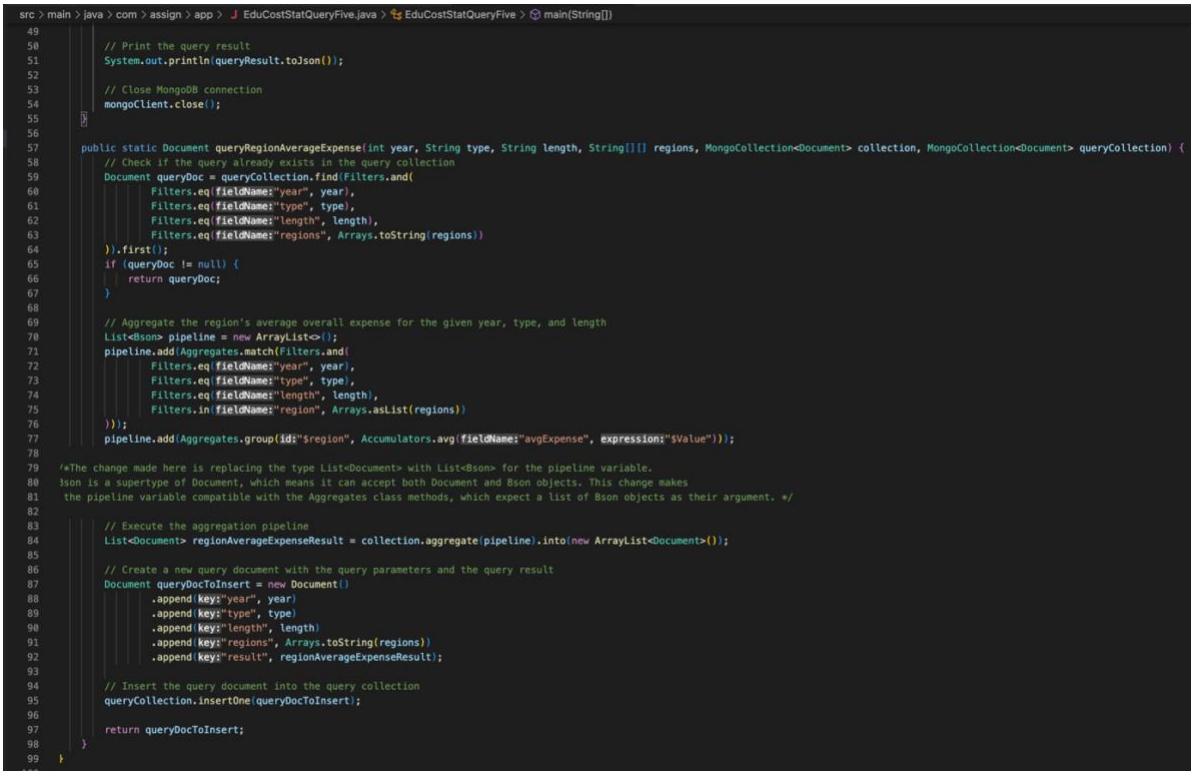
The method then creates a new query document containing the query parameters and the query results, and inserts it into the query collection using the insertOne method of the MongoCollection class.

Finally, the main method of the program calls the queryRegionAverageExpense method and prints the results to the console. The MongoDB connection is then closed using the close method of the MongoClient class.

Here's the implementation of the EduCostStatQueryFive:

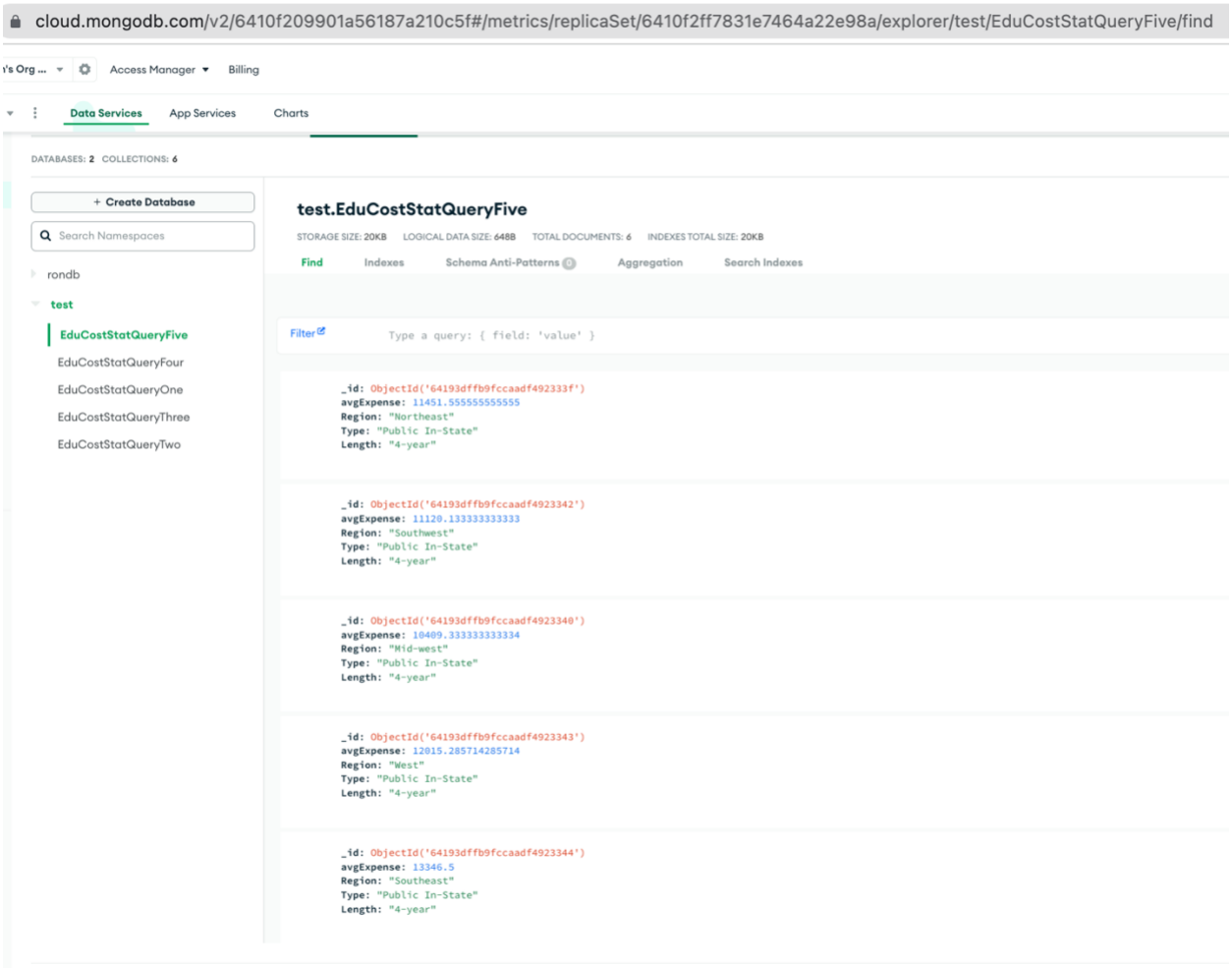


```
src > main > java > com > assign > app > J EduCostStatQueryFive.java > EduCostStatQueryFive > main(String[])
package com.assign.app;

import com.mongodb.client.MongoClients;
import com.mongodb.client.MongoClient;
import com.mongodb.client.MongoCollection;
import com.mongodb.client.MongoDatabase;
import com.mongodb.client.model.Accumulators;
import com.mongodb.client.model.Aggregates;
import com.mongodb.client.model.Filters;
//import com.mongodb.client.model.Projections;
//import com.mongodb.client.model.Sorts;
import org.bson.Document;
import org.bson.conversions.Bson;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class EduCostStatQueryFive {

    private static final String DB_NAME = "test";
    private static final String COLLECTION_NAME = "EduCostStat";
    private static final String QUERY_COLLECTION_NAME = "EduCostStatQueryFive";
   // private static final Object startYear = null;
    //private static final Object endYear = null;

    Run | Debug
    public static void main(String[] args) {
        // Set up MongoDB connection
        MongoClient mongoClient = MongoClients.create(connectionString:"mongodb+srv://rkodava:Dimpu1997@educoststat.ioim5Be.mongodb.net/?retryWrites=true&w=majority");
        MongoDatabase database = mongoClient.getDatabase(DB_NAME);
        MongoCollection<Document> collection = database.getCollection(COLLECTION_NAME);
        MongoCollection<Document> queryCollection = database.getCollection(QUERY_COLLECTION_NAME);

        // Define the regions array with the states
        String[] West = new String[]{"Washington", "Oregon", "California", "Nevada", "Hawaii", "Alaska"};
        String[] Midwest = new String[]{"Idaho", "Montana", "Wyoming", "North Dakota", "South Dakota", "Nebraska", "Kansas", "Oklahoma", "Texas"};
        String[] Northeast = new String[]{"Minnesota", "Iowa", "Missouri", "Wisconsin", "Illinois", "Indiana", "Michigan", "Ohio"};
        String[] Southeast = new String[]{"Maine", "New Hampshire", "Vermont", "Massachusetts", "Rhode Island", "Connecticut", "New York", "New Jersey", "Pennsylvania"};
        String[] Southwest = new String[]{"Maryland", "Delaware", "District of Columbia", "Virginia", "West Virginia", "North Carolina", "South Carolina", "Georgia", "Florida", "Kentucky", "Tennessee", "Alabama", "Mississippi"


        // Call the query method
        int year = 2019;
        String type = "Public In-State";
        String length = "4-year";
        String[][] regions = {West, Midwest, Northeast, Southeast, Southwest};
        //String[] regions = new String[]{"Northeast", "Southeast", "Midwest", "Southwest", "West"};
        Document queryResult = queryRegionAverageExpense(year, type, length, regions, collection, queryCollection);

        // Print the query result
        System.out.println(queryResult.toJson());
```



```
src > main > java > com > assign > app > J EduCostStatQueryFive.java > EduCostStatQueryFive > main(String[])
        // Print the query result
        System.out.println(queryResult.toJson());

        // Close MongoDB connection
        mongoClient.close();
    }

    public static Document queryRegionAverageExpense(int year, String type, String length, String[][] regions, MongoCollection<Document> collection, MongoCollection<Document> queryCollection) {
        // Check if the query already exists in the query collection
        Document queryDoc = queryCollection.find(Filters.and(
                Filters.eq(fieldName:"year", year),
                Filters.eq(fieldName:"type", type),
                Filters.eq(fieldName:"length", length),
                Filters.eq(fieldName:"regions", Arrays.toString(regions))
        )).first();
        if (queryDoc != null) {
            return queryDoc;
        }

        // Aggregate the region's average overall expense for the given year, type, and length
        List<Bson> pipeline = new ArrayList<>();
        pipeline.add(Aggregates.match(Filters.and(
                Filters.eq(fieldName:"year", year),
                Filters.eq(fieldName:"type", type),
                Filters.eq(fieldName:"length", length),
                Filters.in(fieldName:"region", Arrays.asList(regions))
        )));
        pipeline.add(Aggregates.group(id:"$region", Accumulators.avg(fieldName:"avgExpense", expression:"$Value")));

    /*The change made here is replacing the type List<Document> with List<Bson> for the pipeline variable.
    Bson is a supertype of Document, which means it can accept both Document and Bson objects. This change makes
     the pipeline variable compatible with the Aggregates class methods, which expect a list of Bson objects as their argument. */

        // Execute the aggregation pipeline
        List<Document> regionAverageExpenseResult = collection.aggregate(pipeline).into(new ArrayList<Document>());

        // Create a new query document with the query parameters and the query result
        Document queryDocToInsert = new Document()
                .append(key:"year", year)
                .append(key:"type", type)
                .append(key:"length", length)
                .append(key:"regions", Arrays.toString(regions))
                .append(key:"result", regionAverageExpenseResult);

        // Insert the query document into the query collection
        queryCollection.insertOne(queryDocToInsert);

        return queryDocToInsert;
    }
}
```

The result from the above code for given year, type, length equals to 2019, Public In-State, 4-year is as below:



TASK-2

Task 2.1) Define a ProtocolBuff definition file to represent the request, response and service for each query in Task 1.

ProtocolBuff definition files to represent the request, response and service for query-1,2,3,4,5 are as below :

Query-1

```
syntax = "proto3";

package edu_cost_stat;

message QueryOneRequest {
    int32 year = 1;
    string state = 2;
    string type = 3;
    string length = 4;
    int32 expense = 5;
}

message QueryOneResponse {
    int32 year = 1;
    string state = 2;
    string type = 3;
    string length = 4;
    int32 expense = 5;
    double cost = 6;
}

service EduCostStatQueryOne {
    rpc queryCost(QueryOneRequest) returns (QueryOneResponse);
}
```

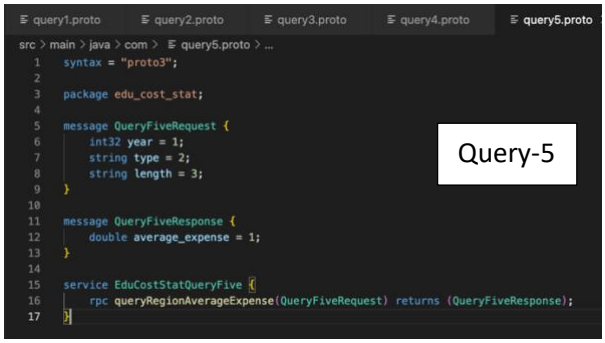
Query-4

```
syntax = "proto3";

package edu_cost_stat;

message QueryFourRequest {
    repeated int32 years = 1;
    int32 year = 2;
    string type = 3;
    string length = 4;
}

message QueryFourResponse {
    repeated StateGrowthPair state_growth_pairs = 1;

    message StateGrowthPair {
        string state = 1;
        double growth_rate = 2;
    }
}

service EduCostStatQueryFour {
    rpc queryTop5HighestGrowthRateStates(QueryFourRequest) returns (QueryFourResponse);
}
```

Query-2

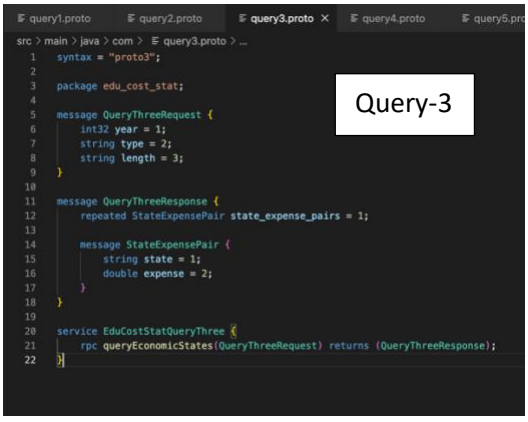
```
syntax = "proto3";

package edu_cost_stat;

message QueryTwoRequest {
    int32 year = 1;
    string type = 2;
    string length = 3;
}

message QueryTwoResponse {
    repeated StateExpensePair state_expense_pairs = 1;

    message StateExpensePair {
        string state = 1;
        double expense = 2;
    }
}

service EduCostStatQueryTwo {
    rpc queryTop5ExpensiveStates(QueryTwoRequest) returns (QueryTwoResponse);
}
```

Query-5



```
syntax = "proto3";

package edu_cost_stat;

message QueryFiveRequest {
    int32 year = 1;
    string type = 2;
    string length = 3;
}

message QueryFiveResponse {
    double average_expense = 1;
}

service EduCostStatQueryFive {
    rpc queryRegionAverageExpense(QueryFiveRequest) returns (QueryFiveResponse);
}
```

Query-3



```
syntax = "proto3";

package edu_cost_stat;

message QueryThreeRequest {
    int32 year = 1;
    string type = 2;
    string length = 3;
}

message QueryThreeResponse {
    repeated StateExpensePair state_expense_pairs = 1;

    message StateExpensePair {
        string state = 1;
        double expense = 2;
    }
}

service EduCostStatQueryThree {
    rpc queryEconomicStates(QueryThreeRequest) returns (QueryThreeResponse);
}
```

Task 2.2) Develop the Java program for each service defined in Task 2.1 as gRPC services. Each service invokes the corresponding data access object classes developed in Task 1..

Now, we need to **generate the Java classes from the Protocol Buffers file using the protoc compiler.**

Here is how to generate the classes:

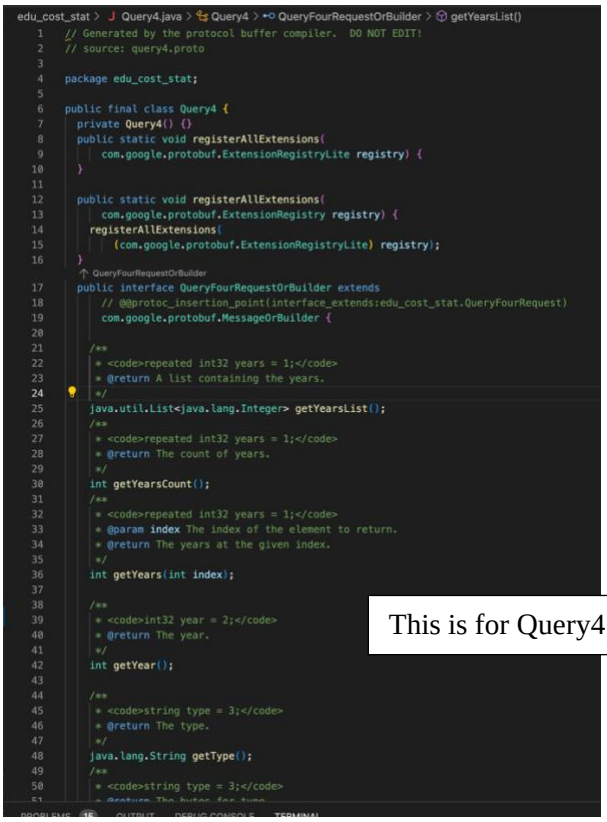
```
(base) rohankodavalla@Rohans-MacBook-Air final_docs %
/Users/rohankodavalla/Downloads/protoc-22.2-osx-x86_64/bin/protoc --
js_out=import_style=commonjs,binary:. --plugin=/usr/local/bin/protoc-gen-js --
proto_path=/Users/rohankodavalla/Desktop/STUDY/winter23/6731-distributed/assign-
2/final_docs/assign/src/main/java/com
/Users/rohankodavalla/Desktop/STUDY/winter23/6731-distributed/assign-
2/final_docs/assign/src/main/java/com/query1.proto
```

The output Java file will be named query1.java and will contain both QueryOneRequest and QueryOneResponse message classes as well as the EduCostStatQueryOne service definition.

Similarly, when done for all 4 other .proto files, 4 **query.java** files are obtained -

Below is the view of how the file will look:

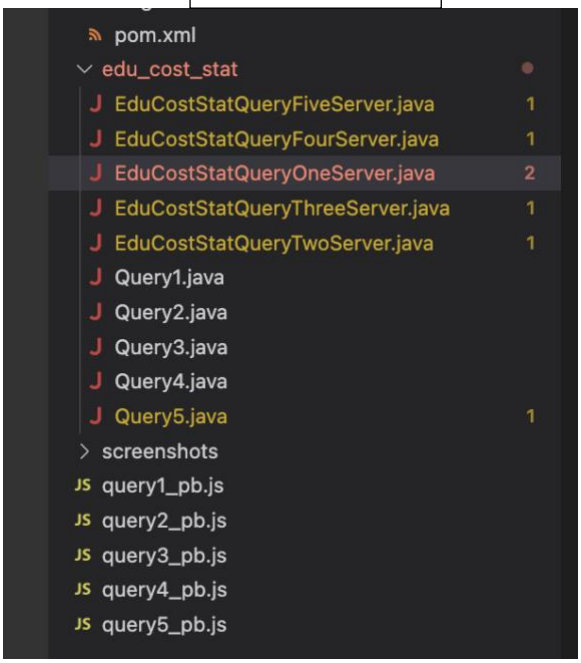
This is for Query3



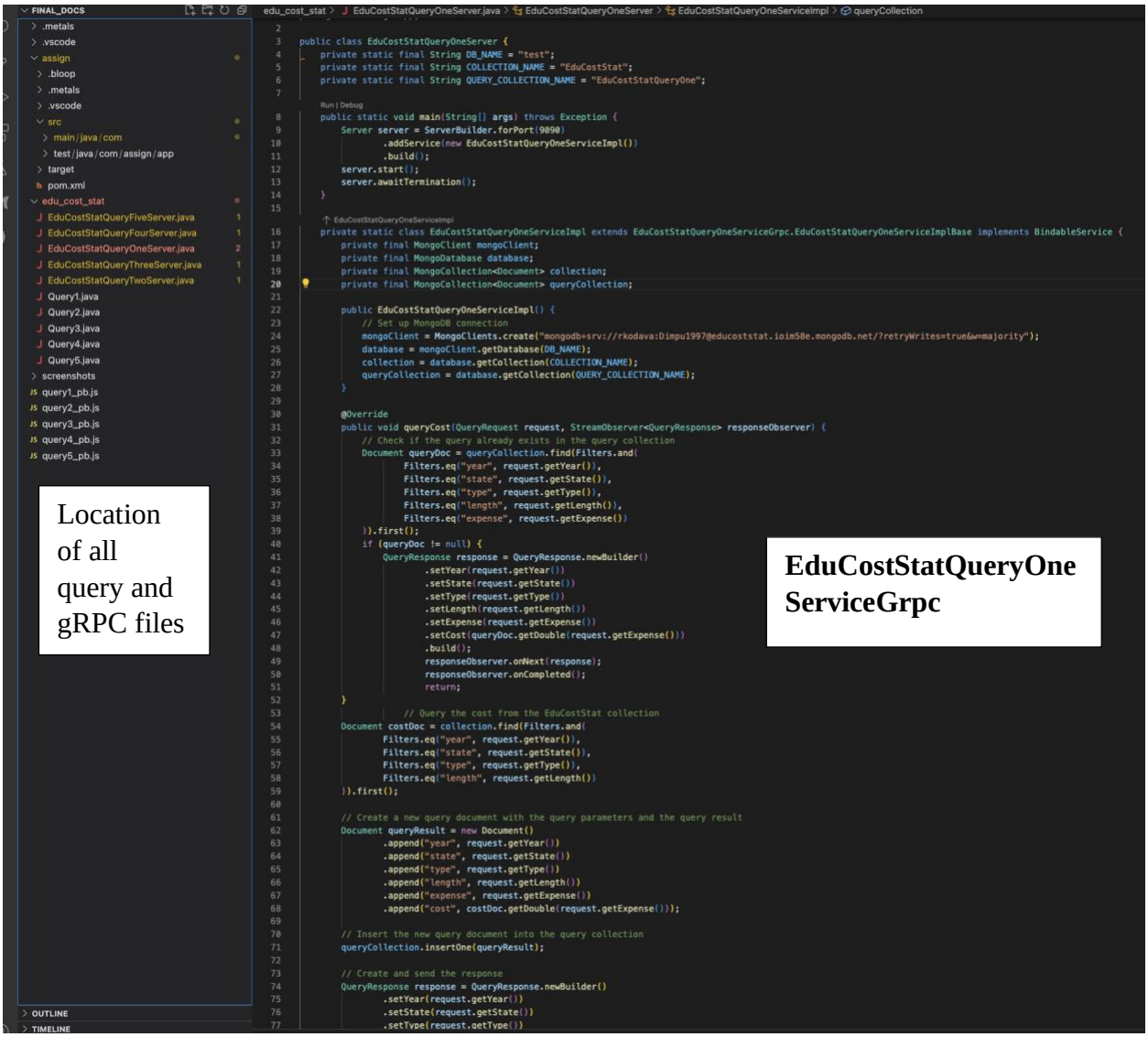
This is for Query4

This is for Query5

Location of all query.java files



Once we have the generated Java classes, we can implement the gRPC service by extending the **EduCostStatQueryOneServiceGrpc** class and overriding the queryCost method. Here is an example of how we can implement the service:

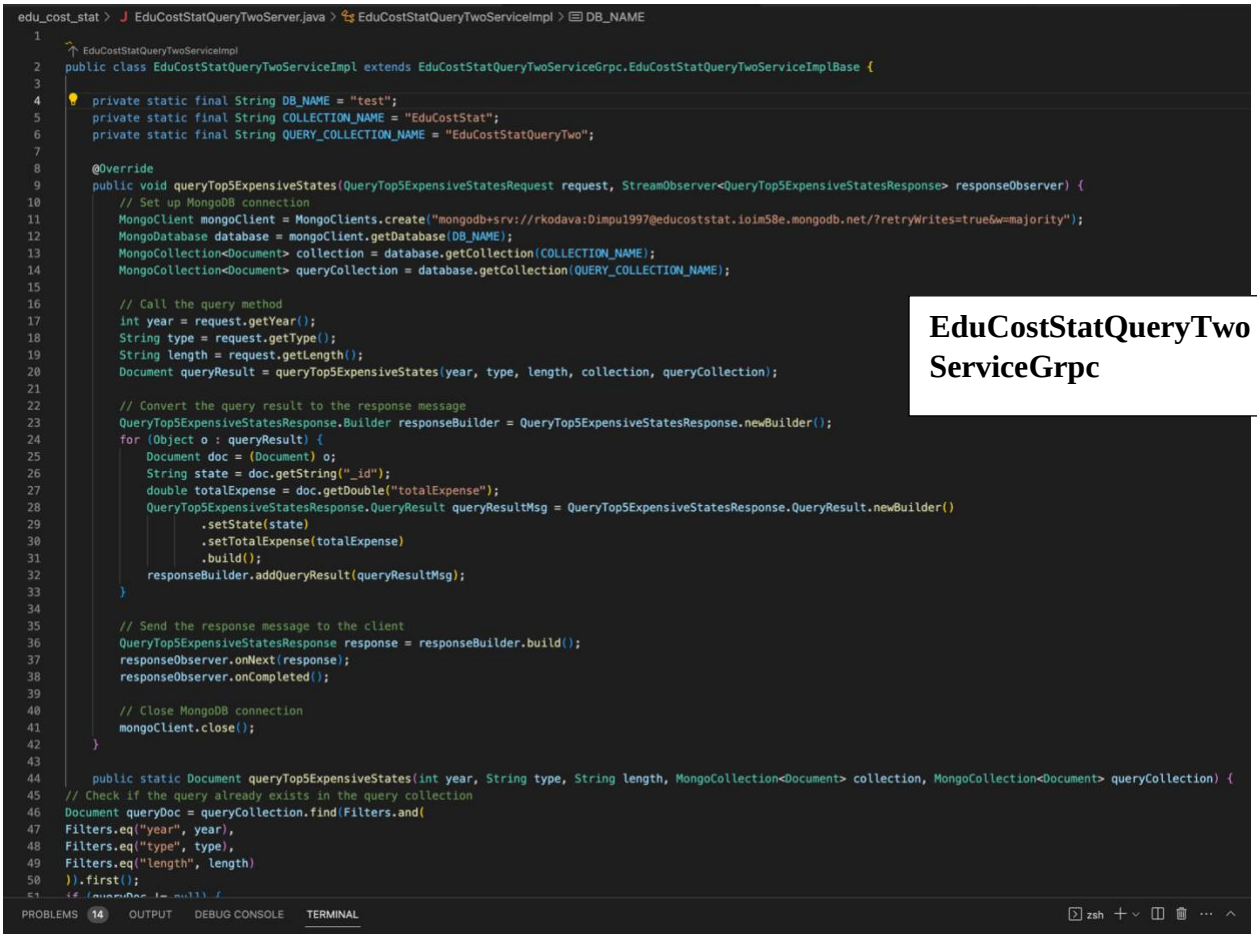


The above code is implementing a gRPC service. The **EduCostStatQueryOneServer** class creates a gRPC server on port 9090 and registers an instance of EduCostStatQueryOneServiceImpl with the server as a service.

The EduCostStatQueryOneServiceImpl class extends EduCostStatQueryOneServiceGrpc.EduCostStatQueryOneServiceImplBase which is a generated abstract class from the gRPC .proto file. The queryCost method in EduCostStatQueryOneServiceImpl defines the service method for the queryCost RPC defined in the .proto file.

Therefore, the code implements a gRPC service based on the EduCostStatQueryOne service definition in the .proto file.

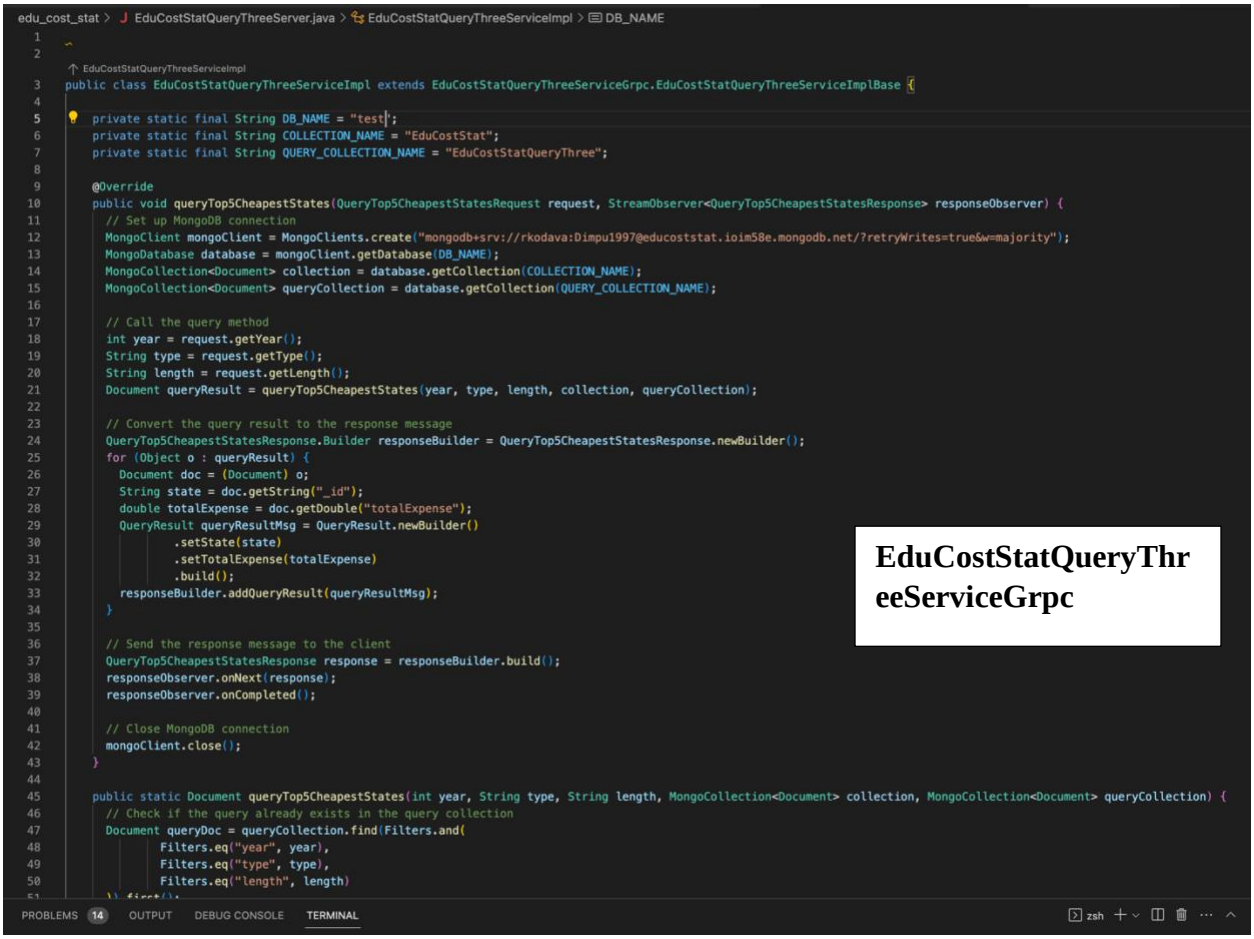
Similarly below are snippets for the rest of the gRPC services for EduCostStat :



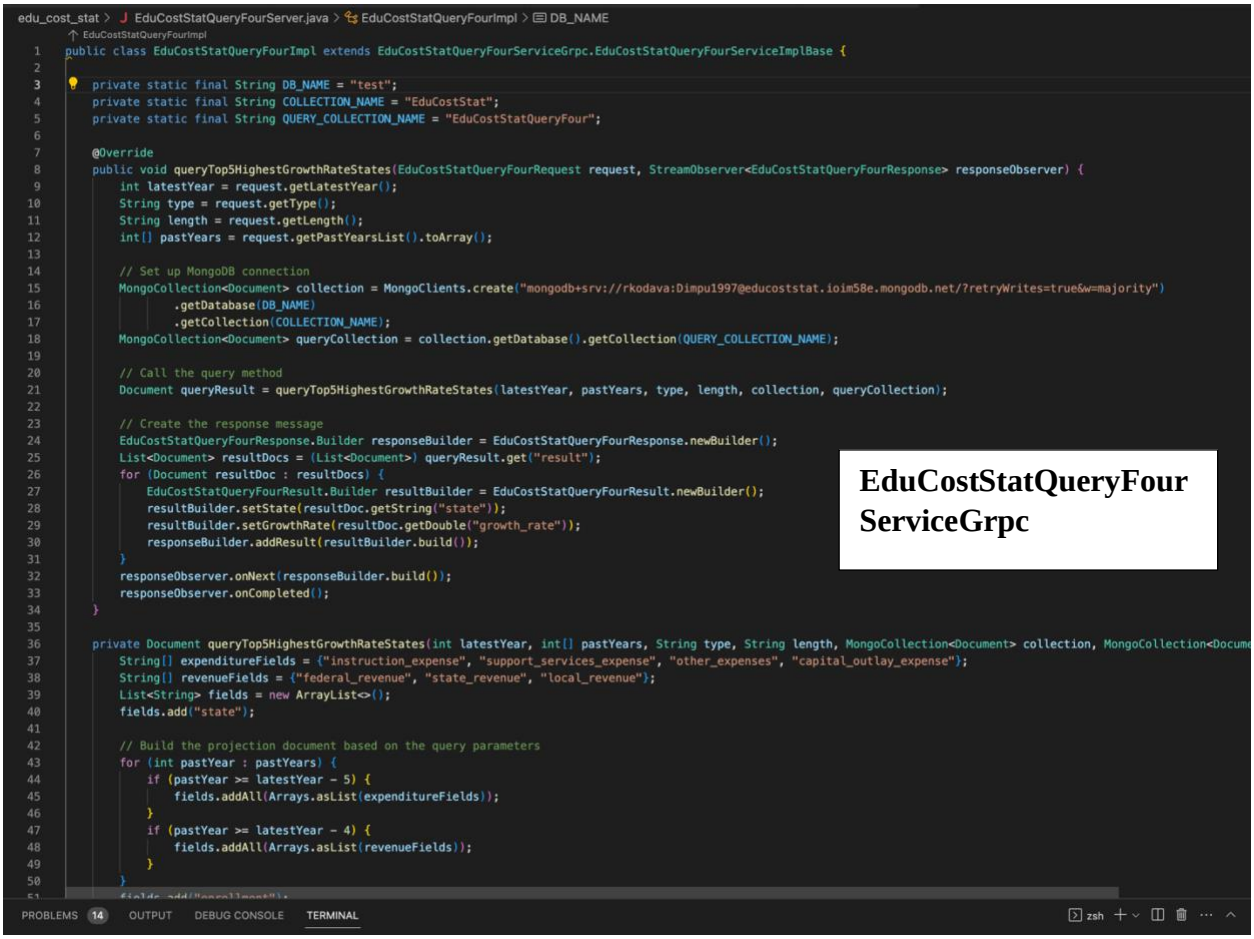
The code above implements a gRPC service.The class **EduCostStatQueryTwoServiceImpl** extends **EduCostStatQueryTwoServiceGrpc**.EduCostStatQueryTwoServiceImplBase, which is generated by the gRPC compiler and provides the base implementation for the service. The methods in EduCostStatQueryTwoServiceImpl are the actual implementation of the gRPC service's methods.

The gRPC service method implemented in the code is queryTop5ExpensiveStates, which takes a QueryTop5ExpensiveStatesRequest and returns a QueryTop5ExpensiveStatesResponse. The method also takes a StreamObserver as a parameter to send the response back to the client.

The implementation of queryTop5ExpensiveStates connects to a MongoDB database, performs a query, and returns the results to the client as a QueryTop5ExpensiveStatesResponse. Therefore, the code is a gRPC service implementation that uses a MongoDB database.

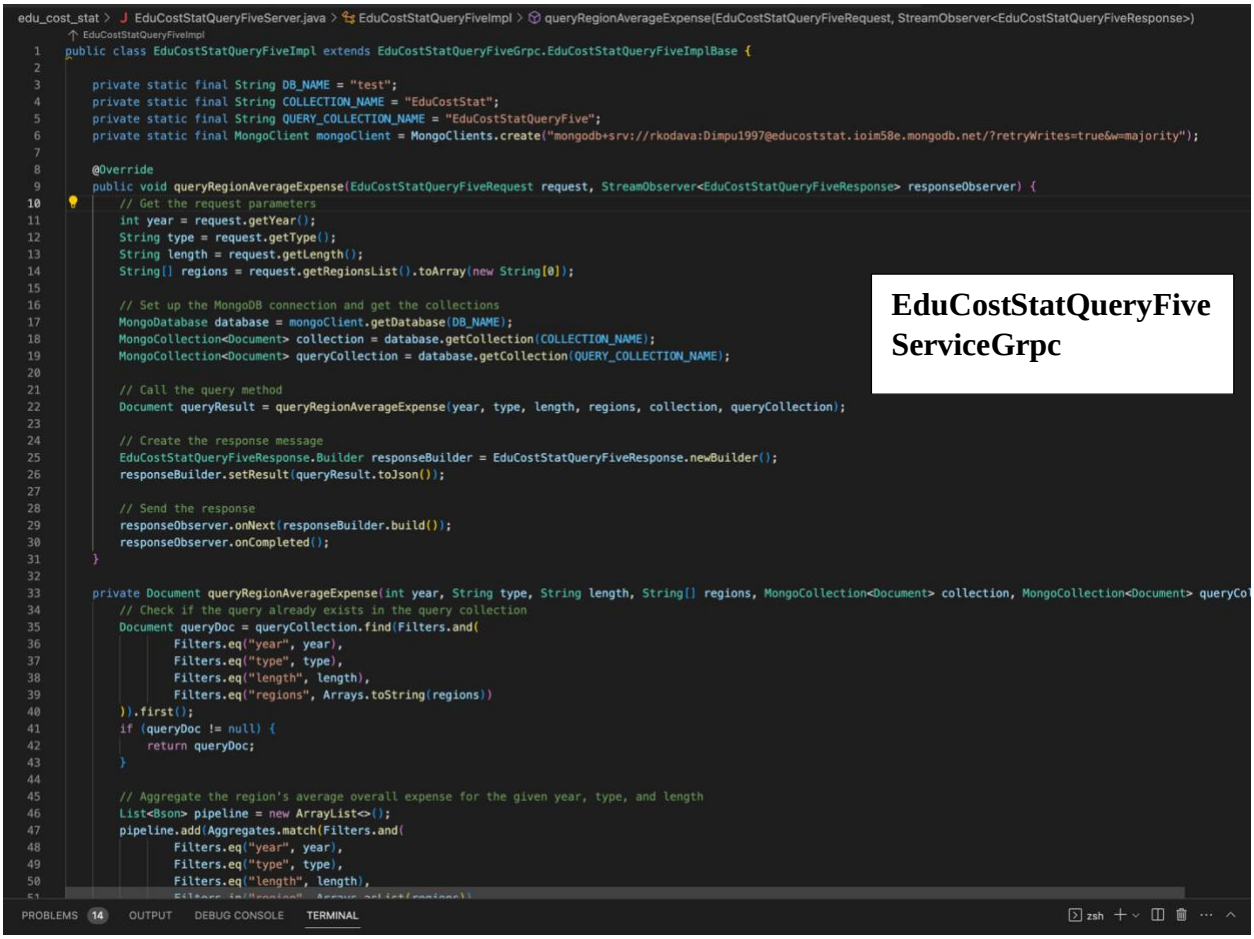


The code above implements a gRPC service defined in a proto file. The service is named **EduCostStatQueryThreeService** and has a unary RPC method named queryTop5CheapestStates. The implementation class **EduCostStatQueryThreeServiceImpl** extends EduCostStatQueryThreeServiceGrpc.EduCostStatQueryThreeServiceImplBase, which is generated by the protobuf compiler and contains the implementation of the EduCostStatQueryThreeService service interface.



The above code implements a gRPC service. The class **EduCostStatQueryFourImpl** extends **EduCostStatQueryFourServiceGrpc**.**EduCostStatQueryFourServiceImplBase**, which is a generated class by the gRPC compiler based on the protobuf definition of the service. The queryTop5HighestGrowthRateStates method is an implementation of the queryTop5HighestGrowthRateStates RPC method defined in the protobuf file, and it takes in a EduCostStatQueryFourRequest and a StreamObserver<EduCostStatQueryFourResponse> as parameters, which is also defined in the protobuf file.

Additionally, the code creates a MongoDB connection and executes a MongoDB query inside the queryTop5HighestGrowthRateStates method, which suggests that this service is using a database to store and retrieve data for the gRPC client requests.



The above code implements a gRPC service using the EduCostStatQueryFiveImpl class. This class extends the EduCostStatQueryFiveGrpc.EduCostStatQueryFiveImplBase class, which is a generated class by the gRPC compiler based on the protobuf definition of the service. The queryRegionAverageExpense method is an implementation of the queryRegionAverageExpense RPC method defined in the protobuf file, and it takes in a EduCostStatQueryFiveRequest and a StreamObserver<EduCostStatQueryFiveResponse> as parameters, which is also defined in the protobuf file.

The code sets up a MongoDB connection using the MongoClient class from the MongoDB Java driver and executes a MongoDB query inside the queryRegionAverageExpense method. The query method first checks if the query already exists in the query collection, and if it does, returns the result. If the query does not exist, the method executes an aggregation pipeline to compute the average overall expense for each region for a given year, type, length, and regions. The result is stored in a query document and inserted into the query collection for future use.

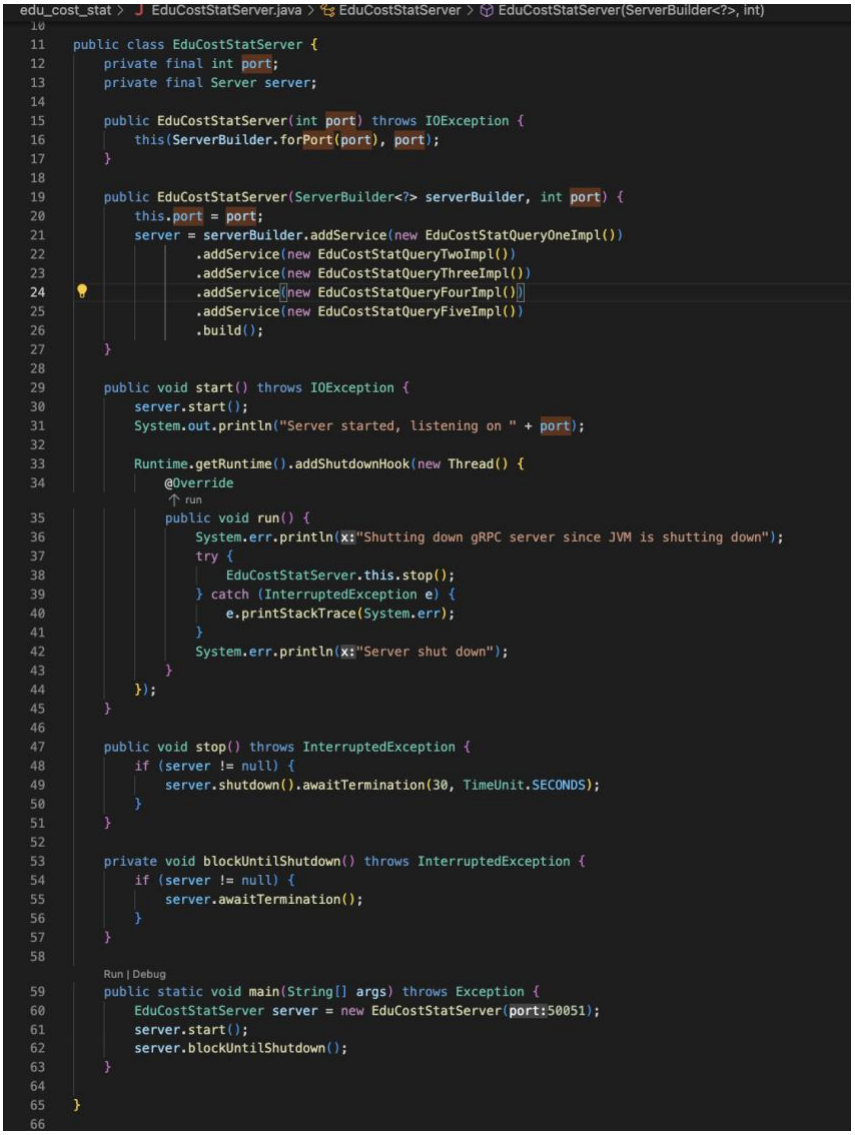
Overall, this suggests that the EduCostStatQueryFive gRPC service is using a database to store and retrieve data for client requests.

Task 2.3) Develop the gRPC client and server (or gateway) code to communicate as RPC calls for the five queries defined in Task 1

SERVER :

The server – 'EduCostStatServer' starts the gRPC server and binds the EduCostStatQueryOneServiceImpl , EduCostStatQueryTwoServiceImpl , EduCostStatQueryThreeServiceImpl , EduCostStatQueryFourServiceImpl , EduCostStatQueryFiveServiceImpl implementations (in above task) to it.

Snippets of the code to the server – 'EduCostStatServer' is below -



The above code is an implementation of a gRPC server called EduCostStatServer that exposes five different services to clients. The server listens on port 50051.

The EduCostStatServer constructor takes a port number and a ServerBuilder object as arguments. The ServerBuilder object is used to create the server and add the five different services (EduCostStatQueryOneImpl, EduCostStatQueryTwoImpl, EduCostStatQueryThreeImpl, EduCostStatQueryFourImpl, and EduCostStatQueryFiveImpl) to the server.

The start() method starts the server and prints a message to the console indicating that the server is listening on the specified port. It also adds a shutdown hook to the JVM so that the server can be stopped gracefully when the JVM shuts down.

The stop() method shuts down the server and blocks until the shutdown is complete.

The blockUntilShutdown() method blocks until the server is terminated.

The main() method creates a new instance of the EduCostStatServer class and starts it. The program blocks until the server is shut down or interrupted.

CLIENT :

The client is used to communicate with a server that provides information about educational costs statistics. The client communicates with the server using gRPC, a high-performance open-source remote procedure call (RPC) framework that enables client and server applications to communicate transparently.

Snippets of the code to the client – 'EduCostStatClient' is below -

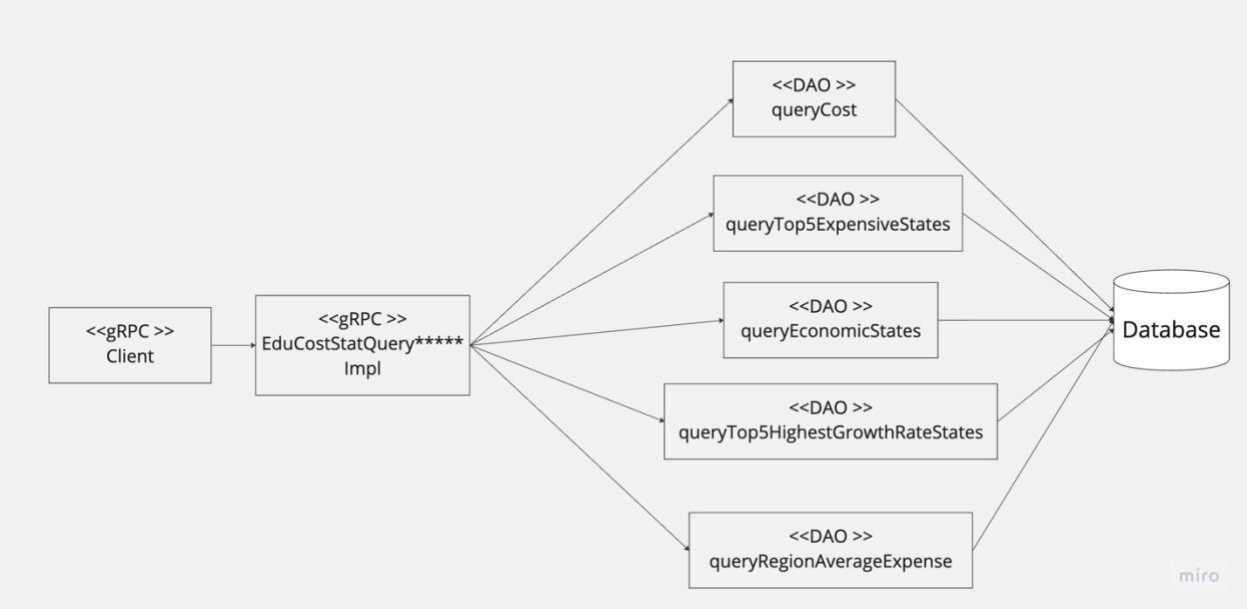
```
edu_cost_stat > J EduCostStatClient.java > ...
1   import io.grpc.ManagedChannel;
2   import io.grpc.ManagedChannelBuilder;
3
4   public class EduCostStatClient {
5       private final ManagedChannel channel;
6       private final EduCostStatQueryOneGrpc.EduCostStatQueryOneBlockingStub queryOneStub;
7       private final EduCostStatQueryTwoGrpc.EduCostStatQueryTwoBlockingStub queryTwoStub;
8       private final EduCostStatQueryThreeGrpc.EduCostStatQueryThreeBlockingStub queryThreeStub;
9       private final EduCostStatQueryFourGrpc.EduCostStatQueryFourBlockingStub queryFourStub;
10      private final EduCostStatQueryFiveGrpc.EduCostStatQueryFiveBlockingStub queryFiveStub;
11
12      public EduCostStatClient(String host, int port) {
13          this(ManagedChannelBuilder.forAddress(host, port)
14                  .usePlaintext()
15                  .build());
16      }
17
18      public EduCostStatClient(ManagedChannel channel) {
19          this.channel = channel;
20          this.queryOneStub = EduCostStatQueryOneGrpc.newBlockingStub(channel);
21          this.queryTwoStub = EduCostStatQueryTwoGrpc.newBlockingStub(channel);
22          this.queryThreeStub = EduCostStatQueryThreeGrpc.newBlockingStub(channel);
23          this.queryFourStub = EduCostStatQueryFourGrpc.newBlockingStub(channel);
24          this.queryFiveStub = EduCostStatQueryFiveGrpc.newBlockingStub(channel);
25      }
26
27      public void shutdown() throws InterruptedException {
28          channel.shutdown().awaitTermination(5, TimeUnit.SECONDS);
29      }
30
31      public void queryOne(int year, String state, String type, String length, int expense) {
32          EduCostStatQueryOneRequest request = EduCostStatQueryOneRequest.newBuilder()
33                  .setYear(year)
34                  .setState(state)
35                  .setType(type)
36                  .setLength(length)
37                  .setExpense(expense)
38                  .build();
39
40          EduCostStatQueryOneResponse response = queryOneStub.queryCost(request);
41
42          System.out.println("Query One Response: " + response.getResult());
43      }
44
45
46      public void queryTwo(int year, String type, String length) {
47          EduCostStatQueryTwoRequest request = EduCostStatQueryTwoRequest.newBuilder()
48                  .setYear(year)
49                  .setType(type)
50                  .setLength(length)
51                  .build();
52
53          EduCostStatQueryTwoResponse response = queryTwoStub.queryTop5ExpensiveStates(request);
54
55          System.out.println("Query Two Response: " + response.getResult());
56      }
57
58      public void queryThree(int year, String type, String length) {
59          EduCostStatQueryThreeRequest request = EduCostStatQueryThreeRequest.newBuilder()
60                  .setYear(year)
61                  .setType(type)
62                  .setLength(length)
63                  .build();
```

The above client code defines a Java class called EduCostStatClient, which is used to communicate with a gRPC server. The EduCostStatClient class has several methods for running different queries on the server, and it takes care of setting up the gRPC communication channel and stubs for each query.

The EduCostStatClient constructor takes a host and port, which are used to set up the gRPC communication channel with the server. The queryOne, queryTwo, queryThree, queryFour, and queryFive methods take different input parameters for each query and use the corresponding gRPC stubs to send the request to the server and receive the response.

The main method creates an instance of the EduCostStatClient class and calls the different query methods with sample input parameters. Finally, the shutdown method is called to terminate the gRPC channel connection.

So, therefore from above tasks the Overall Assignment 2 Data Operation Architecture with RPC Communication is as below –



→ EduCostStatQuery****Impl may be considered as any of the query – EduCostStatQueryOneImpl , EduCostStatQueryTwoImpl, EduCostStatQueryThreeImpl, EduCostStatQueryFourImpl or EduCostStatQueryFiveImpl .

When the client code invokes the appropriate query method, it will communicate with the server using the gRPC protocol to call the corresponding method in the EduCostStatQuery*****Impl implementation on the server-side. The server will then execute the query and return the results back to the client through the gRPC response message. The client will receive the response and print the results to the console.