
# MongoDB and gRPC

## Assignment 2

COEN 6731: Distributed Software Systems

Rishi Murugesan Gopalakrishnan - 40200594
Pritam Sethuraman - 40230509

**Department of Electrical and Computer Engineering**

Gina Cody School of Engineering and Computer Science

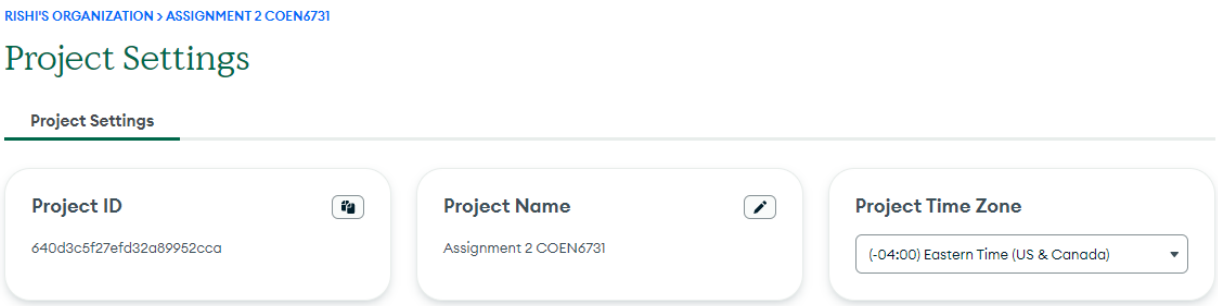
Winter 2023

# Task 1 : **MongoDB Data Storage and Operation**

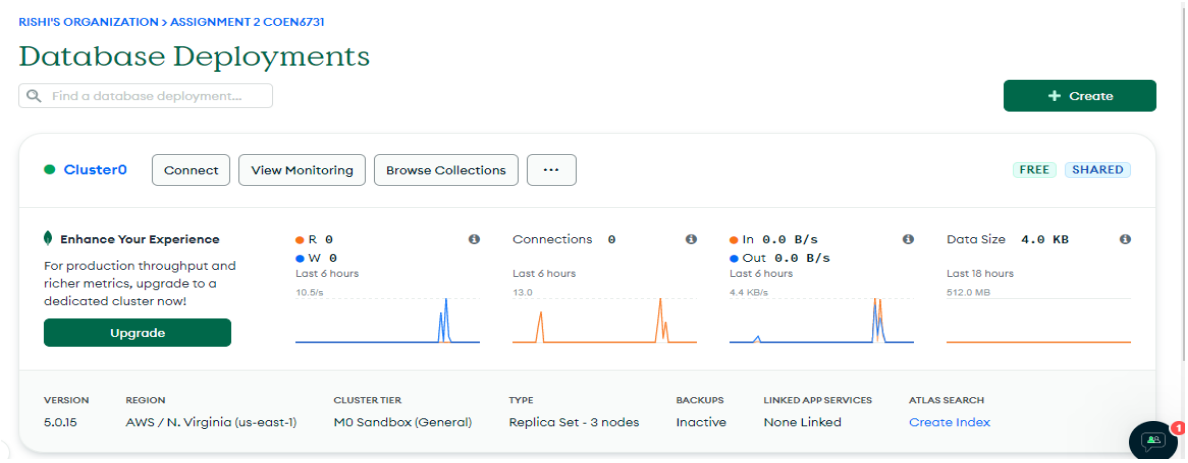
The main objectives of this task are to create a MongoDB collection and store the data from Kaggle data source to the MongoDB instance running on MongoDB online cluster. The next task is to develop Data Access Objects to represent different queries to the data.

## Task 1.1 Creating MongoDB Collection

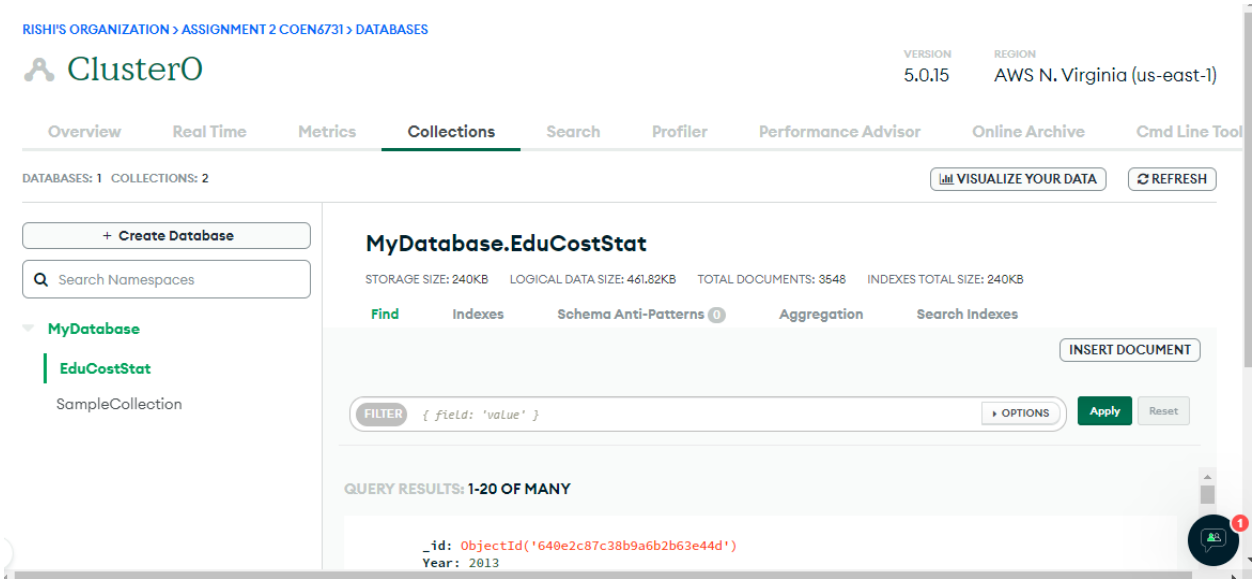
Objective of this task is to create a MongoDB collection named **EduCostStat** to store the data from Kaggle to a MongoDB cloud instance.
First step was to create a MongoDB project on the MongoDB site. A project named "**Assignment 2 COEN6731**" was created for the purpose of this assignment.



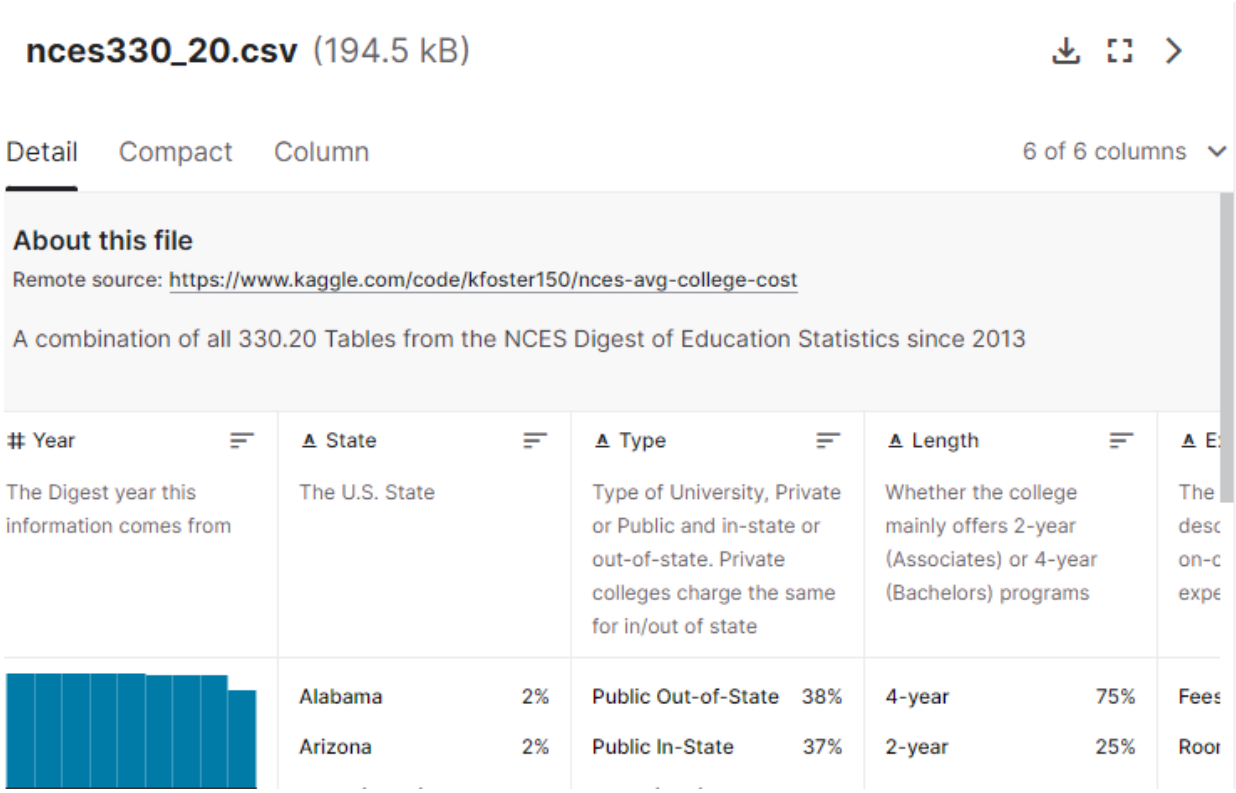

All projects have MongoDB clusters hosted in them. In MongoDB, a **cluster** is a group of servers that store your data. A Cluster can hold a number of databases. A cluster can consist of one or more servers, each of which is called a node. Clusters provide high availability and scalability, allowing your application to grow and handle large amounts of data.
For this assignment, **Cluster0**, a free tier cluster was created in the MongoDB project.



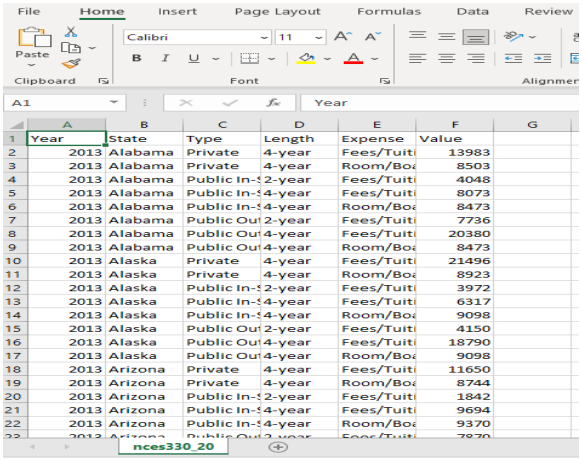
A **database** in MongoDB is a container for collections. You can think of a database as a logical grouping of related data, much like a spreadsheet containing multiple sheets. A MongoDB cluster can have multiple databases, and each database can contain multiple collections.




The **Avg Cost of Undergrad College by State** dataset compiled by National Center of Education Statistics Annual Digest, USA is used as the sample dataset in this assignment.



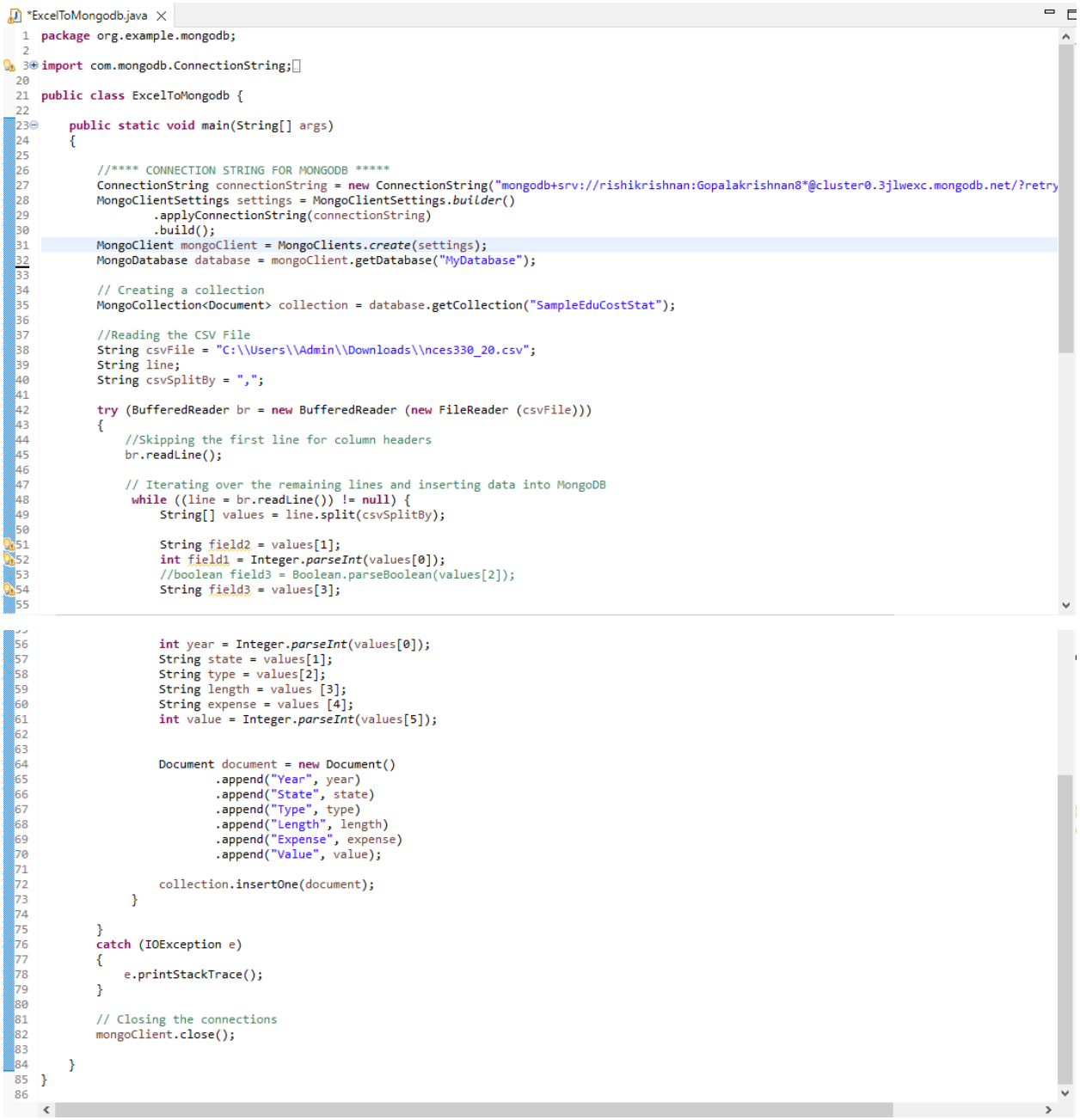

This dataset was downloaded and saved locally as a **CSV (comma-separated values)** file.




A Java program was developed to read and store data from the above csv file into a MongoDB collection named **EduCostStat**.

## Program to read and store data

A program named **ExcelToMongodb.java** was created for the purpose of reading data from the csv file and storing it as a collection in MongoDB.

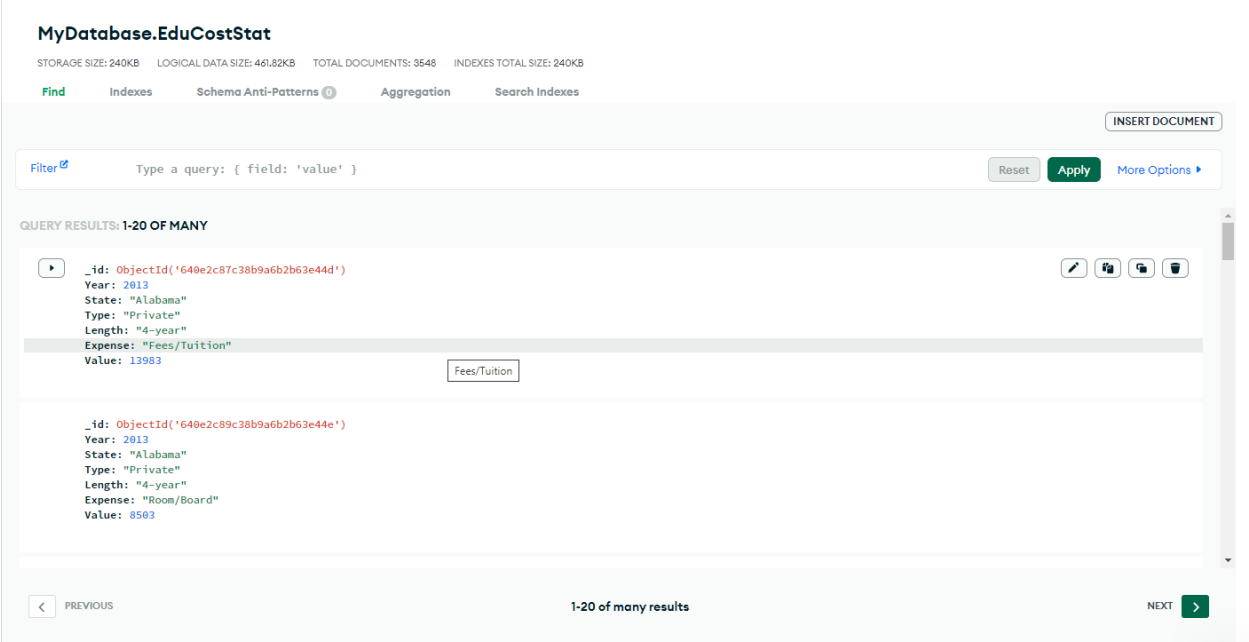


```java
package org.example.mongodb;

import com.mongodb.ConnectionString;

public class ExcelToMongodb {

    public static void main(String[] args)
    {

        //**** CONNECTION STRING FOR MONGODB *****
        ConnectionString connectionString = new ConnectionString("mongodb+srv://rishikrishnan:Gopalakrishnan8*@cluster0.3jlwexc.mongodb.net/?retry
        MongoClientSettings settings = MongoClientSettings.builder()
                .applyConnectionString(connectionString)
                .build();
        MongoClient mongoClient = MongoClients.create(settings);
        MongoDatabase database = mongoClient.getDatabase("MyDatabase");

        // Creating a collection
        MongoCollection<Document> collection = database.getCollection("SampleEduCostStat");

        //Reading the CSV File
        String csvFile = "C:\\Users\\Admin\\Downloads\\nces330_20.csv";
        String line;
        String csvSplitBy = ",";

        try (BufferedReader br = new BufferedReader (new FileReader (csvFile)))
        {
            //Skipping the first line for column headers
            br.readLine();

            // Iterating over the remaining lines and inserting data into MongoDB
             while ((line = br.readLine()) != null) {
                String[] values = line.split(csvSplitBy);

                String field2 = values[1];
                int field1 = Integer.parseInt(values[0]);
                //boolean field3 = Boolean.parseBoolean(values[2]);
                String field3 = values[3];


                int year = Integer.parseInt(values[0]);
                String state = values[1];
                String type = values[2];
                String length = values [3];
                String expense = values [4];
                int value = Integer.parseInt(values[5]);


                Document document = new Document()
                        .append("Year", year)
                        .append("State", state)
                        .append("Type", type)
                        .append("Length", length)
                        .append("Expense", expense)
                        .append("Value", value);

                collection.insertOne(document);
             }

        }
        catch (IOException e)
        {
            e.printStackTrace();
        }

        // Closing the connections
        mongoClient.close();

    }
}
```

- The **connectionstring** variable is used to connect to the MongoDB instance in cloud MongoDB. After getting connected to MongoDB, **MongoClient** library is used to access the database to create collections.
- The **collection** variable fetches the **EduCostStat** collection, if such collection is not present in the database, then a collection with the same name is created.

- **BufferedredReader** library is used to read data from the csv file and it is then stored in collected EduCostStat.

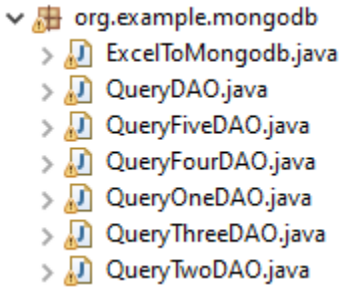
The following screenshot is the MongoDB EduCostStat collection.




From the above image it is evident that all 3548 records from the csv file are inserted into the collection.
This collection is used to create Data Access Objects to query data in Task 1.2

## Task 1.2 Developing Data Access Objects

A Data Access Object (DAO) is a method for compiling code so that it is easier to modify where the application gets its data without affecting the rest of the application's code. This makes it easier to maintain and update the code over time. By using DAO, the code that deals with how data is stored and retrieved can be separated from the rest of the application.



In this task a data access object is created for each query.

1) **EduCostStatQueryOne** - Given a specific year, state, type, length and expense, the DAO should query the cost given for the given parameters and store the result as a document in a collection named **EduCostStatQueryOne**.

```
public List<Document> query_one(int year,String state,String type,String length,String expense) throws InterruptedException {
    //***** CONNECTION STRING FOR MONGODB *****
            ConnectionString connectionString = new ConnectionString("mongodb+srv://rishikrishnan:Gopalakrishnan8*@cluster0.3jlwexc.mongodb.r
            MongoClientSettings settings = MongoClientSettings.builder()
                    .applyConnectionString(connectionString)
                    .build();
            MongoClient mongoClient = MongoClients.create(settings);
            MongoDatabase database = mongoClient.getDatabase("MyDatabase");
            Thread.sleep(1000);

    //***** QUERY ONE *****
    //get collection
    MongoCollection<Document> collection = database.getCollection("SampleEduCostStat");

    Bson query_1 = Filters.and(
            Filters.eq("Year", year),
            Filters.eq("State", state),
            Filters.eq("Type", type),
            Filters.eq("Length", length),
            Filters.eq("Expense", expense)
            );

    Bson projection = Projections.fields(Projections.excludeId(),Projections.include("Value"));

    //Creating a collection for Query 1
    MongoCollection<Document> query_collection = database.getCollection("EduCostStatQueryOne");

    query_collection.createIndex(Indexes.ascending("Value"),new IndexOptions().unique(true));

    FindIterable<Document> docs = collection.find(query_1).projection(projection);

    List<Document> resultDocs = new ArrayList<>();

    for(Document doc : docs)
    {
        Object Value = doc.get("Value");
        Document query_result = new Document()
                .append("Value",Value);
```

The above screenshot is the java code written in IDE to query the value and store it in the collection in MongoDB.

2) **EduCostStatQueryTwo** - Given a year, type and length we have to query the top 5 expensive states and store it in a collection named **EduCostStatQueryTwo**. For this query we have used aggregation to aggregate the values of Room/Board and Fees/Tuition and sort the states based on overall expenses.

```
public List<Document> query_two(int year,String type,String length) throws InterruptedException {
    //***** CONNECTION STRING FOR MONGODB *****
    ConnectionString connectionString = new ConnectionString("mongodb+srv://rishikrishnan:Gopalakrishnan8*@cluster0.3jlwexc.mongodb.net/
    MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build();
    MongoClient mongoClient = MongoClients.create(settings);
    MongoDatabase database = mongoClient.getDatabase("MyDatabase");
    Thread.sleep(1000);
    //***** QUERY TWO *****

    //MongoDB command line example
    //db.SampleEduCostStat.aggregate([{ $match: {Type:"Private",Year:2013,Length:"4-year"}},{$project:{State:1,Value:1,_id:0}},{$group:{

    //get collection
    MongoCollection<Document> collection = database.getCollection("SampleEduCostStat");

    //Building aggregation pipeline
    AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
            new Document("$match",new Document ("Type",type)
                    .append("Year",year)
                    .append("Length",length)),
            new Document("$project",new Document("State",1)
                    .append("Value",1)
                    .append("_id", 0)),
            new Document("$group",new Document("_id","$State")
                    .append("Overall Expense", new Document("$sum","$Value"))),
            new Document("$sort",new Document("Overall Expense",-1)),
            new Document("$limit",5)
            ));

    //Creating a collection for Query 2
    MongoCollection<Document> query_collection = database.getCollection("EduCostStatQueryTwo");

    query_collection.createIndex(Indexes.ascending("_id","Overall Expense"),new IndexOptions().unique(true));


    //save results to a new collection
    List<Document> resultDocs = new ArrayList<>();
```

3) **EduCostStatQueryThree -** Given a year, type and length we have to query the top 5 economic states and store it in a collection named **EduCostStatQueryThree**. For this query we have used aggregation to aggregate the values of Room/Board and Fees/Tuition and sort the states based on overall expenses. Here economic states are considered to be the less expensive states.

```
public List<Document> query_three(int year,String type,String length) throws InterruptedException {

    //***** CONNECTION STRING FOR MONGODB *****
    ConnectionString connectionString = new ConnectionString("mongodb+srv://rishikrishnan:Gopalakrishnan8*@cluster0.3jlwexc.mongodb.net/?retr
    MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build();
    MongoClient mongoClient = MongoClients.create(settings);
    MongoDatabase database = mongoClient.getDatabase("MyDatabase");
            Thread.sleep(1000);
    //***** QUERY THREE *****

    //MongoDB command line example
    //db.SampleEduCostStat.aggregate([{ $match: {Type:"Private",Year:2013,Length:"4-year"}},{$project:{State:1,Value:1,_id:0}},{$group:{_id:"

    //get collection
    MongoCollection<Document> collection = database.getCollection("SampleEduCostStat");

    //Building aggregation pipeline
    AggregateIterable<Document> result = collection.aggregate(Arrays.asList(
            new Document("$match",new Document ("Type",type)
                    .append("Year",year)
                    .append("Length",length)),
            new Document("$project",new Document("State",1)
                    .append("Value",1)
                    .append("_id", 0)),
            new Document("$group",new Document("_id","$State")
                    .append("Overall Expense", new Document("$sum","$Value"))),
            new Document("$sort",new Document("Overall Expense",1)),
            new Document("$limit",5)
            ));

    //save results to a new collection
    List<Document> resultDocs = new ArrayList<>();
    for(Document doc:result) {
        resultDocs.add(doc);
    }

    //Creating a collection for Query 3
```

4) **EduCostStatQueryFour -** Given a range, type and length we have to query the top 5 states with highest growth rate and store it in a collection named **EduCostStatQueryFour**. For this query we had to create an aggregate query to calculate growth rate between the range of years, sort them and store the values as documents in the collection.

```
public List<Document> query_four(String type,String length,int switch_var) throws InterruptedException {
    //***** CONNECTION STRING FOR MONGODB *****
    ConnectionString connectionString = new ConnectionString("mongodb+srv://rishikrishnan:Gopalakrishnan8*@cluster0.3jlwexc.mongodb.net/?ret
    MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build();
    MongoClient mongoClient = MongoClients.create(settings);
    MongoDatabase database = mongoClient.getDatabase("MyDatabase");
    Thread.sleep(1000);
    //***** QUERY FOUR *****

    //MongoDB command line example
    //db.SampleEduCostStat.aggregate([{$match:{Type:"Private",Length:"4-year",Year:{$gte:2019}}},{$sort:{State:1,Year:-1}},{$group:{_id:"$St

    //get collection
    MongoCollection<Document> collection = database.getCollection("SampleEduCostStat");

    //Building aggregation pipeline

    switch(switch_var)
    {
    case 1:
        //pipeline for 1-year year range from current year ( 2020 - 2021)
        List<Document> pipeline1 = Arrays.asList(
                new Document("$match", new Document("Type", "Private").append("Length", "4-year").append("Year", new Document("$gte", 2020))
                new Document("$sort", new Document("State", 1).append("Year", -1)),
                new Document("$group", new Document("_id", "$State").append("Base_Value", new Document("$first", "$Value")).append("Current_
                new Document("$project", new Document("State", "$_id").append("Growth_Rate_1_Year", new Document("$let", new Document("vars"
                new Document("$sort", new Document("Growth_Rate_1_Year", -1)),
                new Document("$limit", 5));

        AggregateIterable<Document> results1 = collection.aggregate(pipeline1);
        List<Document> resultDocs1 = new ArrayList<>();
        for (Document doc : results1) {
            System.out.println(doc);
        }
        //Creating a collection for Query 4
        MongoCollection<Document> query_collection1 = database.getCollection("EduCostStatQueryFour");
```

5) **EduCostStatQueryFive -** Given a year, type and length we have to calculate average overall expense for every region in use and save it in collection **EduCostStatQueryFive**. For this query we had to create an aggregate query to categorize all the states in the US into five regions. This is done with the help of https://education.nationalgeographic.org/resource/united-states-regions/ . After categorization the regions and their overall average expense is stored in the collection.

```
public List<Document> query_five(int year,String length,String type) throws InterruptedException {

    //***** CONNECTION STRING FOR MONGODB *****
    ConnectionString connectionString = new ConnectionString("mongodb+srv://rishikrishnan:Gopalakrishnan8*@cluster0.3jlwexc.mongodb.net/?ret
    MongoClientSettings settings = MongoClientSettings.builder()
            .applyConnectionString(connectionString)
            .build();
    MongoClient mongoClient = MongoClients.create(settings);
    MongoDatabase database = mongoClient.getDatabase("MyDatabase");
    Thread.sleep(1000);

    //***** QUERY FIVE *****

    //MongoDB command line example
    //db.SampleEduCostStat.aggregate([{$match:{Year:2013,Type:"Private",Length:"4-year",State:{$in:["Washington","Oregan","California","Neva

    //get collection
    MongoCollection<Document> collection = database.getCollection("SampleEduCostStat");

    List<Document> results = new ArrayList<>();

    //WEST STATES
    AggregateIterable<Document> westResult = collection.aggregate(Arrays.asList(
            new Document("$match", new Document("Year", year)
                .append("Type", type)
                .append("Length", length)
                .append("State", new Document("$in", Arrays.asList("Washington", "Oregan", "California", "Nevada", "Idaho", "Utah", "Montana
            new Document("$group", new Document("_id", "West")
                .append("total_value", new Document("$sum", "$Value")))
        ));
    results.add(westResult.first());


    //SOUTH WEST STATES
    AggregateIterable<Document> southWestResult = collection.aggregate(Arrays.asList(
        new Document("$match", new Document("Year", 2013)
            .append("Type", "Private")
            .append("Length", "4-year")
            .append("State", new Document("$in", Arrays.asList("Arizona","New Mexico","Texas","Oklahoma")))),
```

To make sure that no duplicate queries are inserted as a new document in a collection we have created an indexing function to make every document in the collection to be unique and execute them using a try catch block. For example, a screenshot of query two validation is inserted here.

```
//Creating a collection for Query 2
MongoCollection<Document> query_collection = database.getCollection("EduCostStatQueryTwo");

query_collection.createIndex(Indexes.ascending("_id","Overall Expense"),new IndexOptions().unique(true));


//save results to a new collection
List<Document> resultDocs = new ArrayList<>();
for(Document doc:result) {
    resultDocs.add(doc);
}

//Insert the results in the collection
try {

    query_collection.insertMany(resultDocs);

}catch (MongoWriteException e)
{
    if(e.getError().getCode() == 11000 )
    {
        //duplicate key error
        System.out.println("Duplicate Document Found");
    }else
    {
        throw e;
    }
}
```

# Task 2 : **Data Communication Interface Definition and Service Implementation**

The gRPC is a Remote Procedure Call framework that is designed to be an efficient communication channel. It uses Protobuf as its interface definition language (IDL) to specify the messages and services that will be used in the communication between the client and server. To use gRPC, the first step is to define the messages and services in a Protobuf file. Then, the gRPC code generator automatically generates the code for the client and server in the desired programming language. This code provides an easy-to-use and effective API that enables the client and server to communicate with each other.

## Task 2.1 Defining a protobuff definition file

A protobuff definition file is used to represent the request, response and service for every query in Task 1. It is named as query.proto in the source code.

```
syntax ="proto3";

option java_package = "com.assignment.grpc";
option java_multiple_files = true;
message QueryOneRequest {
    int32 year = 1;
    string state = 2;
    string type = 3;
    string length = 4;
    string expense = 5;
}
message QueryOneResponse{
    repeated int32 value = 1;
}
service EduCostStatQueryOneService{
    rpc QueryOne(QueryOneRequest) returns (QueryOneResponse);
}
message QueryTwoRequest{
    int32 year = 1;
    string type = 2;
    string length = 3;
}
message QueryTwoResponse{
    repeated StateExpenseQueryTwo state_expense = 1;
}
message StateExpenseQueryTwo{
    string state = 1;
    int32 overall_expense = 2;
}
service EduCostStatQueryTwoService{
    rpc QueryTwo(QueryTwoRequest) returns (QueryTwoResponse);
}
message QueryThreeRequest{
    int32 year = 1;
    string type = 2;
    string length = 3;
}
message QueryThreeResponse{
    repeated StateExpenseQueryThree state_expense = 1;
    repeated StateExpenseQueryThree state_expense = 1;
}
message StateExpenseQueryThree{
    string state = 1;
    int32 overall_expense = 2;
}
service EduCostStatQueryThreeService{
    rpc QueryThree(QueryThreeRequest) returns (QueryThreeResponse);
}
message QueryFourRequest{
    string type = 1;
    string length = 2;
    int32 range = 3;
}
message QueryFourResult{
    string state = 1;
    float growth_rate = 2;
}
message QueryFourResponse{
    repeated QueryFourResult results = 1;
}
service EduCostStatQueryFourService{
    rpc QueryFour(QueryFourRequest) returns (QueryFourResponse);
}
message QueryFiveRequest{
    int32 year = 1;
    string type = 2;
    string length = 3;
}
message QueryFiveResult{
    string id = 1;
    int32 total_value = 2;
}
message QueryFiveResponse{
    repeated QueryFiveResult results = 1;
}
service EduCostStatQueryFiveService{
    rpc QueryFive(QueryFiveRequest) returns (QueryFiveResponse);
}
```

**Request message :** A request message is a communication sent from a client to a server to ask for a particular service. Usually, the request message contains essential information that is required by the server to process the request.

**Response message :** A response message is a communication that a server sends back to a client after receiving a request message. The purpose of the response message is to provide information to the client about the outcome of the request. Typically, the response message includes data that is generated by the server as a result of processing the request.

**Service message :** A service message is a type of communication that establishes the communication interface between a client and a server. It usually includes one or more request and response messages, which define the structure and content of the information exchanged between client and server. The service message also specifies the methods that the client can use to perform various operations on the server.

There are four types of services in protobuff

1. **Unary Service** : Simplest type of service. The communication between the client and server is **one-to-one**.
2. **Server Streaming Service** : Client sends one request to the server, and the server sends back a stream of responses. The communication between the client and server is **one-to-many**.
3. **Client Streaming Service** : Client sends a stream of requests to the server and the server sends back a single response. The communication between the client and server is **many-to-one**.
4. **Bi-Directional Streaming Service** : Client and server both send and receive a stream of messages. The communication between the client and server is **many-to-many**.

For this assignment all the services defined in the proto file are **Unary Service** for the sake of simplicity. The Screenshot below is an example request, response and service message for the QueryOneDAO.

```
message QueryOneRequest {
    int32 year = 1;
    string state = 2;
    string type = 3;
    string length = 4;
    string expense = 5;
}
message QueryOneResponse{
    repeated int32 value = 1;
}
service EduCostStatQueryOneService{
    rpc QueryOne(QueryOneRequest) returns (QueryOneResponse);
}
```

- QueryOneRequest - Request message
- QueryOneResponse - Response Message
- EduCostStatQueryOneService - Unary Service Message in which QueryOne sends a request and QueryOneResponse is the response from the server.

## Task 2.2 Developing Java Program for each queryService

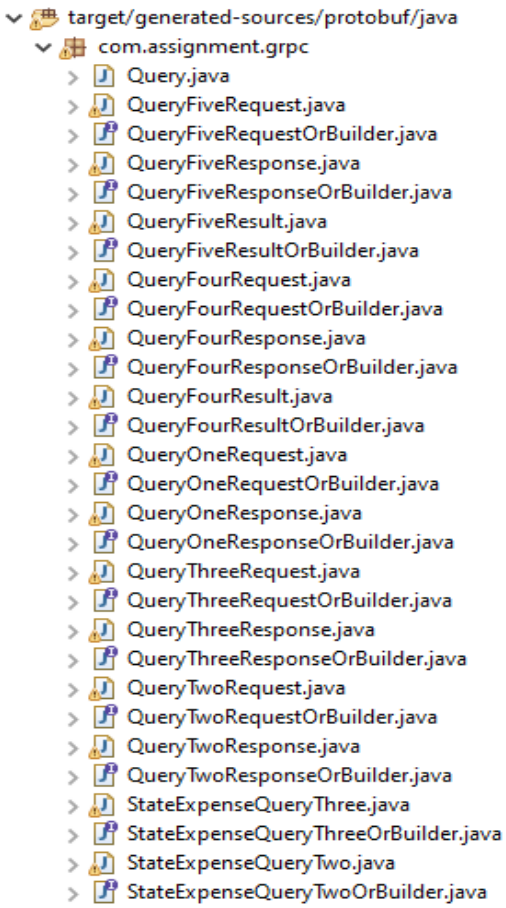
After creating the proto file. It can be used to generate code in desired language which can be used by the client and server.
In this assignment, the **dummy.proto** is used to generate the required code. The following steps were followed to generate the code.

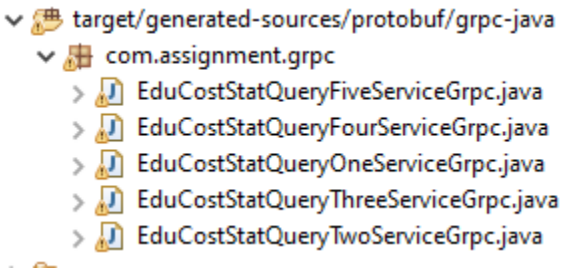
Step 1. Save **dummy.proto** in **src/main/proto** folder in the source code

Step 2. Add the necessary dependencies from the google's protobuf github repository to pom.xml

Step 3. Run **mvn clean install** on the project terminal. This will generate the necessary gRPC files.

The following is the screenshot of generated files for every response, request and service.



Using the generated code, Service code for each query was programmed in the IDE.

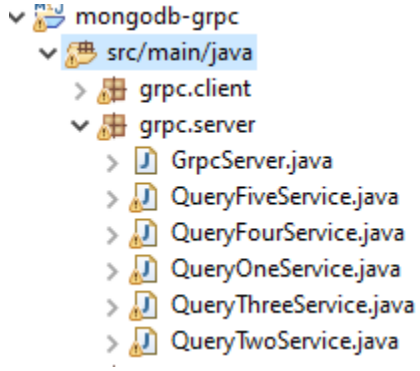
**The Service code is programmed using the following steps:**

1. Each query service class extends Service Implementation Base from the generated grpc code for the corresponding query.
2. Define the query method in which the parameters will be request and response message class.
3. Inside the method, set all the parameters and define request and response parameters.
4. Set an object for the corresponding data access object to call the query.
5. Call the query
6. The server will respond by performing the query
7. The query will be executed and the corresponding result will be stored in the mongoDB.

For example, The following screenshot shows the Service code for QueryOneDAO. It is named **QueryOneService.java.**

```java
package grpc.server;

import com.assignment.grpc.EduCostStatQueryOneServiceGrpc.EduCostStatQueryOneServiceImplBase;

public class QueryOneService extends EduCostStatQueryOneServiceGrpc.EduCostStatQueryOneServiceImplBase{

    @Override
    public void queryOne(QueryOneRequest request,StreamObserver<QueryOneResponse> responseObserver) {
        System.out.println("Query One Started");
        int Year = request.getYear();
        String State = request.getState();
        String Type = request.getType();
        String Length = request.getLength();
        String Expense = request.getExpense();
        QueryOneResponse.Builder response = QueryOneResponse.newBuilder();

        QueryOneDAO dao = new QueryOneDAO();


            List<Document> docs;
            try {
                docs = dao.query_one(Year, State, Type, Length, Expense);
                for(Document doc:docs)
                {
                    response.addValue(doc.getInteger("Value"));
                }
                System.out.println(docs);
                responseObserver.onNext(response.build());
                responseObserver.onCompleted();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }


    }

}
```

Similarly, service code for all the queries are programmed.




## Task 2.3 gRPC client and server

**gRPC Client**

A gRPC client is a module that takes care of sending Remote Procedure Call (RPC) requests to a gRPC server and receiving the corresponding RPC responses. The client can be written in different programming languages using the generated code from protobuf. Here in this code **stubs** are used. A stub is a client-side entity that represents a remote service and enables the client to call methods on the server. The gRPC stub is automatically created by the gRPC code generator using the Protobuf service definition file. It offers an interface that the client can use to interact with the remote service.

**gRPC Server**

A gRPC server is responsible for accepting Remote Procedure Call (RPC) requests from gRPC clients, handling them, and returning the relevant responses. The server-side code for the gRPC server can be generated in various programming languages from the protobuf files. The server listens on a particular port for incoming requests and then matches the request with the corresponding method before processing it and returning to the client.

The following are the steps used to write server side code, named as **GrpcServer.java**:

- Initialize a port (port : 9090)
- Create a server instance for the port and add the Query Services to it using **addService() method**
- Start the server

The following image is the screenshot for the server code

```
public class GrpcServer {

    public static void main(String[] args) throws IOException, InterruptedException {

        int port = 9090;

        Server server = ServerBuilder.forPort(port)
                .addService(new QueryOneService())
                .addService(new QueryTwoService())
                .addService(new QueryThreeService())
                .addService(new QueryFourService())
                .addService(new QueryFiveService())
                .build();

        server.start();
        System.out.println("Server Started");
        System.out.println("Listening on port: "+port);

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.out.println("Received Shutdown Request");
            server.shutdown();
            System.out.println("Server Stopped");
        }));

        server.awaitTermination();
    }

}
```

The following are the steps used to write client side code, named as **GrpcClient.java**:

- A channel is created to send requests to the port 9090 using **ManagedChannel** class.

```
public static void main(String[] args) throws InterruptedException {
    ManagedChannel channel = ManagedChannelBuilder
            .forAddress("localhost",9090)
            .usePlaintext()
            .build();
```

- A **Client stub** is created for all the query services using **BlockingStub** class from the grpc code created.

```
EduCostStatQueryOneServiceBlockingStub query1stub = EduCostStatQueryOneServiceGrpc.newBlockingStub(channel);
EduCostStatQueryTwoServiceBlockingStub query2stub = EduCostStatQueryTwoServiceGrpc.newBlockingStub(channel);
EduCostStatQueryThreeServiceBlockingStub query3stub = EduCostStatQueryThreeServiceGrpc.newBlockingStub(channel);
EduCostStatQueryFourServiceBlockingStub query4stub = EduCostStatQueryFourServiceGrpc.newBlockingStub(channel);
EduCostStatQueryFiveServiceBlockingStub query5stub = EduCostStatQueryFiveServiceGrpc.newBlockingStub(channel);
```

- Use client stubs to call the query services from the server. Request and Response classes of each message are also defined. For example the following screenshot is for Query Service One.

```
QueryOneRequest query1 = QueryOneRequest.newBuilder().setYear(2013).setState("Alabama").setType("Private").setLength("4-year").setExpense("Fees/Tuition").build();
QueryOneResponse response1 = query1stub.queryOne(query1);
System.out.println("Query 1 Executed");
List op1 = new ArrayList();
op1 = response1.getValueList();
System.out.println("Value : "+op1.get(0));
```

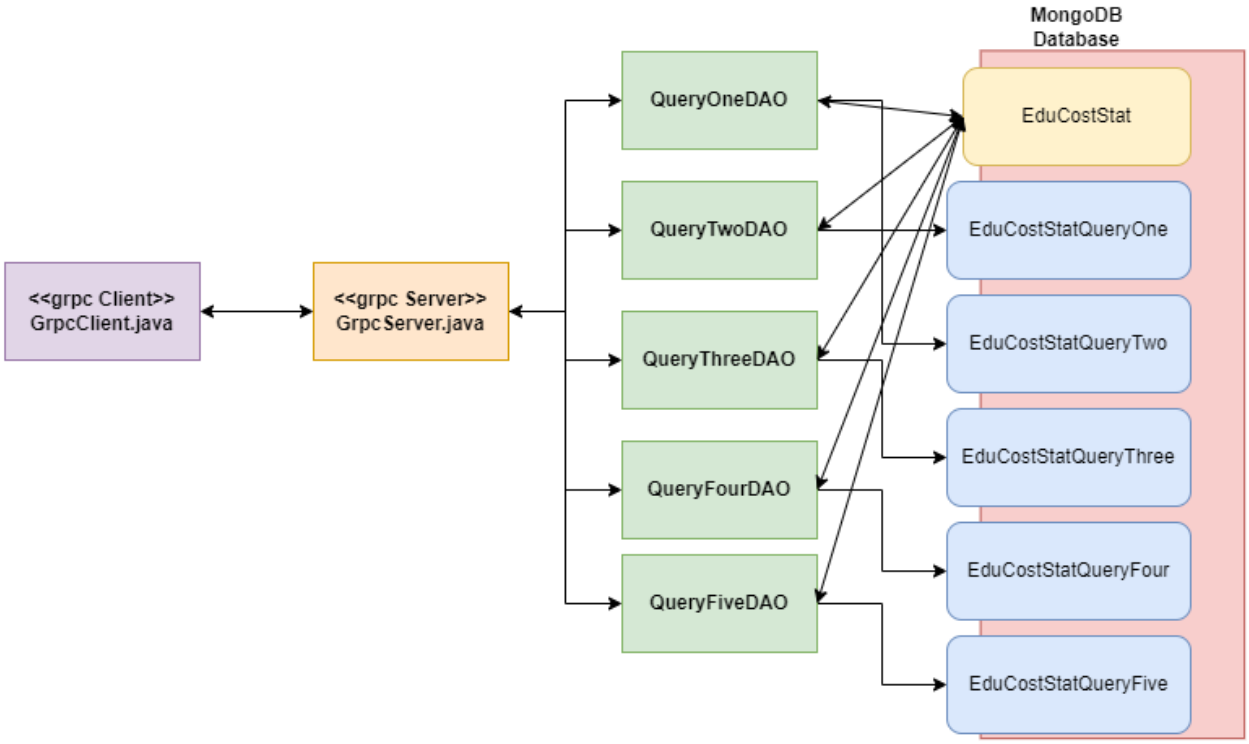
- QueryOneRequest is used to pass all the values to the queryOne method
- The response is stored in response1 which is of QueryOneResponse
- Similarly service calls for all the queries are written.
- Channel is closed after successful execution of all the query services.

```
System.out.println("Shutting down");
channel.shutdown();
```

The following screenshot is the full client code:

```
public class GrpcClient {

    public static void main(String[] args) throws InterruptedException {
        ManagedChannel channel = ManagedChannelBuilder
                .forAddress("localhost",9090)
                .usePlaintext()
                .build();

        EduCostStatQueryOneServiceBlockingStub query1stub = EduCostStatQueryOneServiceGrpc.newBlockingStub(channel);
        EduCostStatQueryTwoServiceBlockingStub query2stub = EduCostStatQueryTwoServiceGrpc.newBlockingStub(channel);
        EduCostStatQueryThreeServiceBlockingStub query3stub = EduCostStatQueryThreeServiceGrpc.newBlockingStub(channel);
        EduCostStatQueryFourServiceBlockingStub query4stub = EduCostStatQueryFourServiceGrpc.newBlockingStub(channel);
        EduCostStatQueryFiveServiceBlockingStub query5stub = EduCostStatQueryFiveServiceGrpc.newBlockingStub(channel);



        QueryOneRequest query1 = QueryOneRequest.newBuilder().setYear(2013).setState("Alabama").setType("Private").setLength("4-year").setExpense("Fees/Tuition").build();
        QueryOneResponse response1 = query1stub.queryOne(query1);
        System.out.println("Query 1 Executed");
        List op1 = new ArrayList();
        op1 = response1.getValueList();
        System.out.println("Value : "+op1.get(0));

        QueryTwoRequest query2 = QueryTwoRequest.newBuilder().setYear(2013).setType("Private").setLength("4-year").build();
        QueryTwoResponse response2 = query2stub.queryTwo(query2);
        System.out.println("Query 2 Executed");
        List op2 = new ArrayList();
        op2 = response2.getStateExpenseList();
        System.out.println("Output : "+ op2);

        QueryThreeRequest query3 = QueryThreeRequest.newBuilder().setYear(2013).setType("Private").setLength("4-year").build();
        QueryThreeResponse response3 = query3stub.queryThree(query3);
        System.out.println("Query 3 Executed");
        List op3 = new ArrayList();
        op3 = response3.getStateExpenseList();
        System.out.println("Output : "+ op3);

        Thread.sleep(1000);

        QueryFourRequest query4 = QueryFourRequest.newBuilder().setType("Private").setLength("4-year").setRange(3).build();
        QueryFourResponse response4 = query4stub.queryFour(query4);
        System.out.println("Query 4 Executed");
        List op4 = new ArrayList();
        op4 = response4.getResultsList();
        System.out.println("Output : "+ op4);

        QueryFiveRequest query5 = QueryFiveRequest.newBuilder().setYear(2013).setLength("4-year").setType("Private").build();
        QueryFiveResponse response5 = query5stub.queryFive(query5);
        System.out.println("Query 5 Executed");
        List op5 = new ArrayList();
        op5 = response5.getResultsList();
        System.out.println("Output : "+ op5);

        System.out.println("Shutting down");
        channel.shutdown();
    }
```

# Architecture Diagram



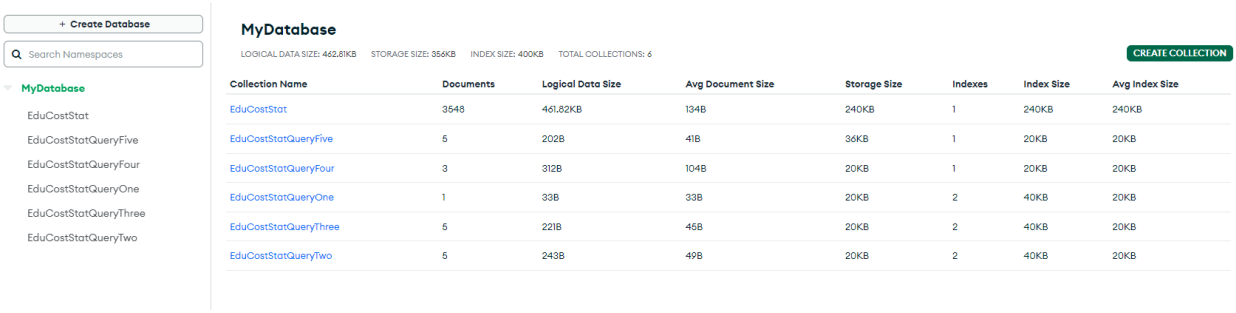

- GrpcClient.java  is the client code that sends client stubs to the server
- GrpcServer.java is the grpcServer that run the QueryServices using the Data Access Objects
- The QueryOneDAO, QueryTwoDAO, QueryThreeDAO, QueryFourDAO and QueryFiveDAO communicate with the MongoDB database.
- The DAO's query the database and create the new collections.
- The server then returns response back to the client

## Running the Program

- After programming the server and client, we have to start by running the client first.
- When we run the server, we get the following output

```
GrpcServer [Java Application] C:\Use
Server Started
Listening on port: 9090
```

- After this the client has to be run to call the service methods
- Once the client is run, the queries are executed and the corresponding collections are created and stored in MongoDB database




- The following is the output from the client in the console

```
Query 1 Executed
Value : 13983
 ********************

Query 2 Executed
Output : [state: "Massachusetts"
overall_expense: 49871
, state: "District of Columbia"
overall_expense: 48440
, state: "Connecticut"
overall_expense: 48262
, state: "Vermont"
overall_expense: 46255
, state: "Rhode Island"
overall_expense: 46114
]
 ********************

Query 3 Executed
Output : [state: "Idaho"
overall_expense: 11544
, state: "Wyoming"
overall_expense: 13562
, state: "Utah"
overall_expense: 15330
, state: "North Dakota"
overall_expense: 17742
, state: "West Virginia"
```