CSCI 5408: Project Report

Team: dpg16

Aman Singh Bhandari (B00910008)
Amankumar Patel (B00888136)
Vivekkumar Patel (B00874162)
Qiwei Sun (B00780054)
Vatsal Yadav (B00893030)

Source Code Repository

Link: https://git.cs.dal.ca/bhandari/dmwaproject/-/tree/feature-user-console
Branch: feature-user-console

1. Summary

Team dpg16 in course DMWA (Jan 2022 – Apr 2022) completed the final project to build a distributed system. The distributed system is built with two GCP VM instances of similar specifications. There are in total two users and each user operates on one single instance. The user needs to register before being able to use any of the features of the database. The system allows users to register a username, password, and 3 sets of security questions and their answers. Password is encrypted with an SHA-1 encryption algorithm and is saved in a text file along with other users' information. Once the user is registered, it can authenticate and log in to the system. Any incorrect information will lead to the failure of the authentication.

This distributed database system allows users to write queries and DML commands. Just like the existing database system in the market, it expects the user to select the database schema or create a schema if the expected database does not exist. The system supports users to create the tables, insert, update, and delete the data. The input queries are first parsed and tokenized into multiple chunks where each chunk represents some meaning in the command. The tokenization of the query is done by applying regex expressions.

As the commands are running in the system each command is generating some output. This output can be positive submission, generation, and selection of data, or it can be a failure/crash. The system is continuously monitoring and writing these logs in JSON files. We are maintaining three different logs file based on the category of the log. The first query log saves the logs when the queries are executed. The second log category is database logs, it logs the changes in the database. The last category is event logs which record any event that happened on a distributed system. It includes crash reports and all the activities.

The data model is created by performing reverse engineering on the database and table metadata. The global metadata provides the information regarding the various tables in a database and their instances. By using this information, the local metadata can be accessed on the corresponding instances to get table and column level information. The reverse engineering is performed on this information to identify relationship, dependencies, and cardinalities. An Entity Relatioship Diagram (ERD) can be created using the relationships between tables and columns. The user can select the Data Model option on the console and provide database name to initiate this process to generate ERD.

Although the user will have a centralized experience, the database built in this project is distributed. The two Virtual machine instances in GCP have their own private IPs and are communicating with each other through SSH protocol. Any input from the user is first processed in the same instance. Through global metadata, it makes sure how many resources will be required to process this command. It makes the list of these resources like R1, R2, R3, …. Rn. Once the list of resources is ready, using global metadata it makes sure to evaluate the location of each resource. It updates the list of resources with their location as well.

The current instance either requires pulling the data or pushing the data to the other instance. To pull the data, cat filename.txt command is executed on other Linux machines to fetch the data in a file. To push the data, echo –e "content" > filaname.txt command is run to save the new content in a file. This way the instance in control makes sure to process the query by pulling and saving the resources that is not present in its persistent storage.

The local metadata is meant to keep the metadata of each table. All the data related to a table are stored under the directory with the name of its database schema. The database schema contains a metadata file that can be identified with <tablename>.txt naming convention. It contains the column information of a particular table like name, datatype, is not null, primary key, foreign key, FK reference table, and column. his information is stored pipe separated ('|') for each column. The table data/rows are present in the table with table_<table_name>.txt naming convention.

Global metadata is to locate the resources of our distributed database system. Each resource can either be present on instance 1 or 2 and the instance in control must figure out where exactly the requested resource resides. It then communicates to the other instance to either pull or pushes the resource. The global metadata data contains multiple lines, and each line is represented by <resource_name>|<instance_number> .

2. Architecture

We have used multilayer architectures in **Figure 1**, where there are multiple layers with unique responsibility and each layer is talking to the layer above and beneath it. It starts with the user console module that is responsible to read the input from user and show output. Once the input is validated, system doesn't know and interpret the input. For this purpose, it is transmitted to query parser layer that parse the command and tokenize it. The tokenized data is easily interpreted the system can now understand the different items individually.

The above discussed two layers are stateful because it stores the data being entered by the user and maintains its state. One example is that it stores the database schema selected by the user and maintains the details of the user currently logged in. All the tasks done under a user will be logged against that user.

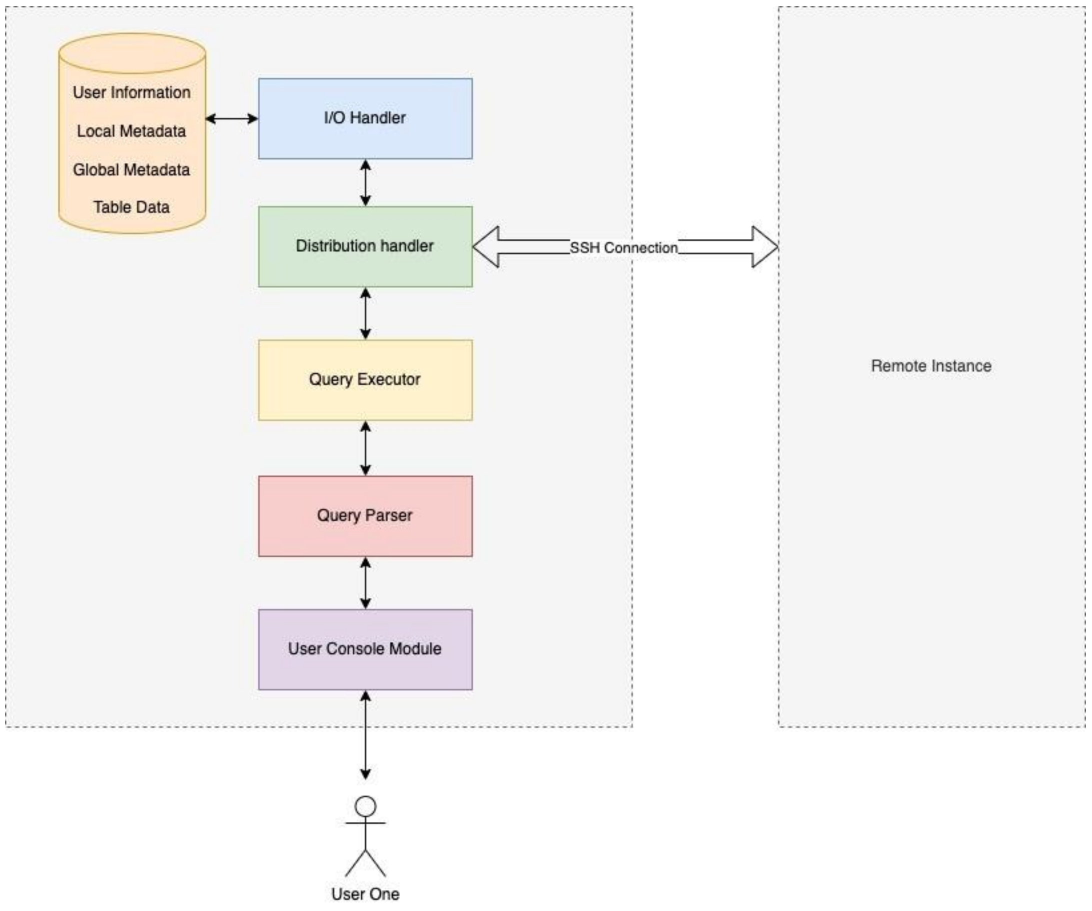


Figure 1: *Architectural representation*

The query executor is stateless as it is solely responsible for executor queries and DML commands. This layer does not maintain any state of the database or user. For example, it will get a tokenized details from the upper layer and executes the command. It doesn't care about which instance exactly the resources are stored and just executed the query. During the execution, requires to read the data from resources and needs to store the data.

The distribution layer is solely responsible to decide where exactly a particular resource is residing. Once the resource is located through global metadata, it builds an SSH connection with the remote machine. Through this SSH connection Linux commands are run on the remote machine to get and save data. Through the I/O handler layer, the data is read and written in both the machines.

3. Local Metadata

Local metadata contains the metadata of each table. It contains the information of each column of the table. Through the local metadata, we can get all the information and constraints on the table. In our solution, we are maintaining the local metadata in a txt format file. The file is present in the folder with the name database schema, where the table exists.

The local metadata structure in the file starts with the column name and followed by the datatype, is not null, primary key, foreign key, foreign key reference table, and foreign key reference column as shown in figure 2. These values are pipe separated. In case a value in column name comes with pipe character, system adds a escape character in front of it. This tells the system that it doesn't need to treat the pipe character as a separator. The name of the file starts with the prefix metadata_ and is concatenated with table name to complete the full file name. For example, metadata_courses.txt is the file name for metadata of course table.

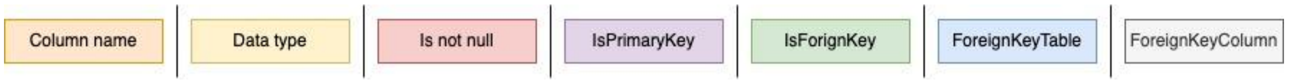
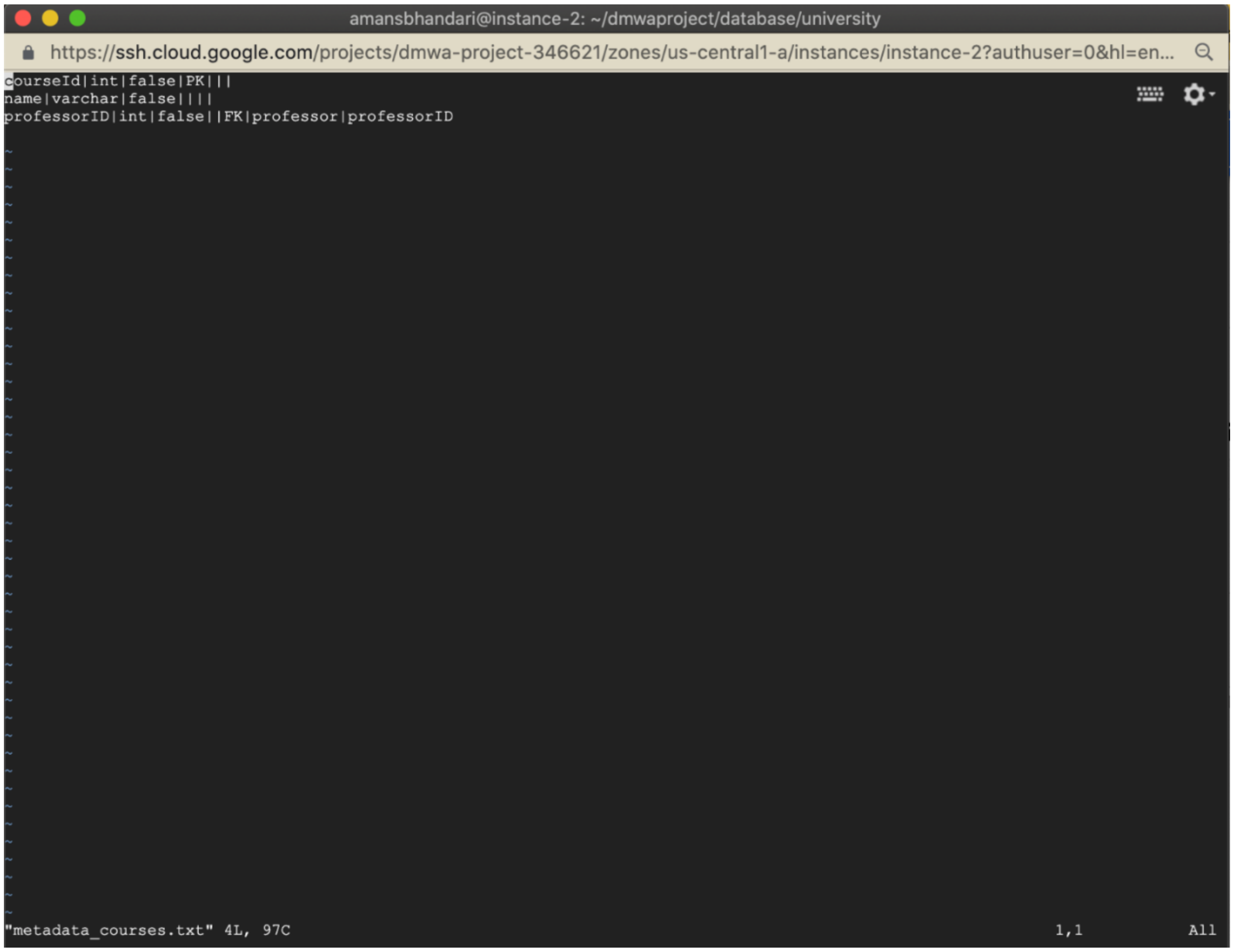


Figure 2: *Local Metadata*

One example of metadata of the table courses is shown below in figure 3.


amansbhandari@instance-2: ~/dmwaproject/database/university
https://ssh.cloud.google.com/projects/dmwa-project-346621/zones/us-central1-a/instances/instance-2?authuser=0&hl=en...

```
courseId|int|false|PK|||
name|varchar|false||||
professorID|int|false||FK|professor|professorID

"metadata_courses.txt" 4L, 97C                                    1,1           All
```

Figure 3: *Example of local metadata*

4. Global Metadata

Global metadata contains the information about the database schema, about their tables, and their locations. In our solution we have established two VM instances, where we are distributing our tables by following the vertical fragmentation. Therefore, in a database schema, table can either be located on instance one or instance two.

To keep this information about the table location, we have designed our global metadata file in such a way that it is easy to extract the location of the same. As noticed in figure 4, the file name followed by the instance name is separated by pipe character separator.

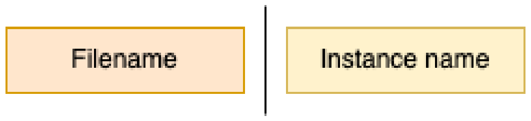
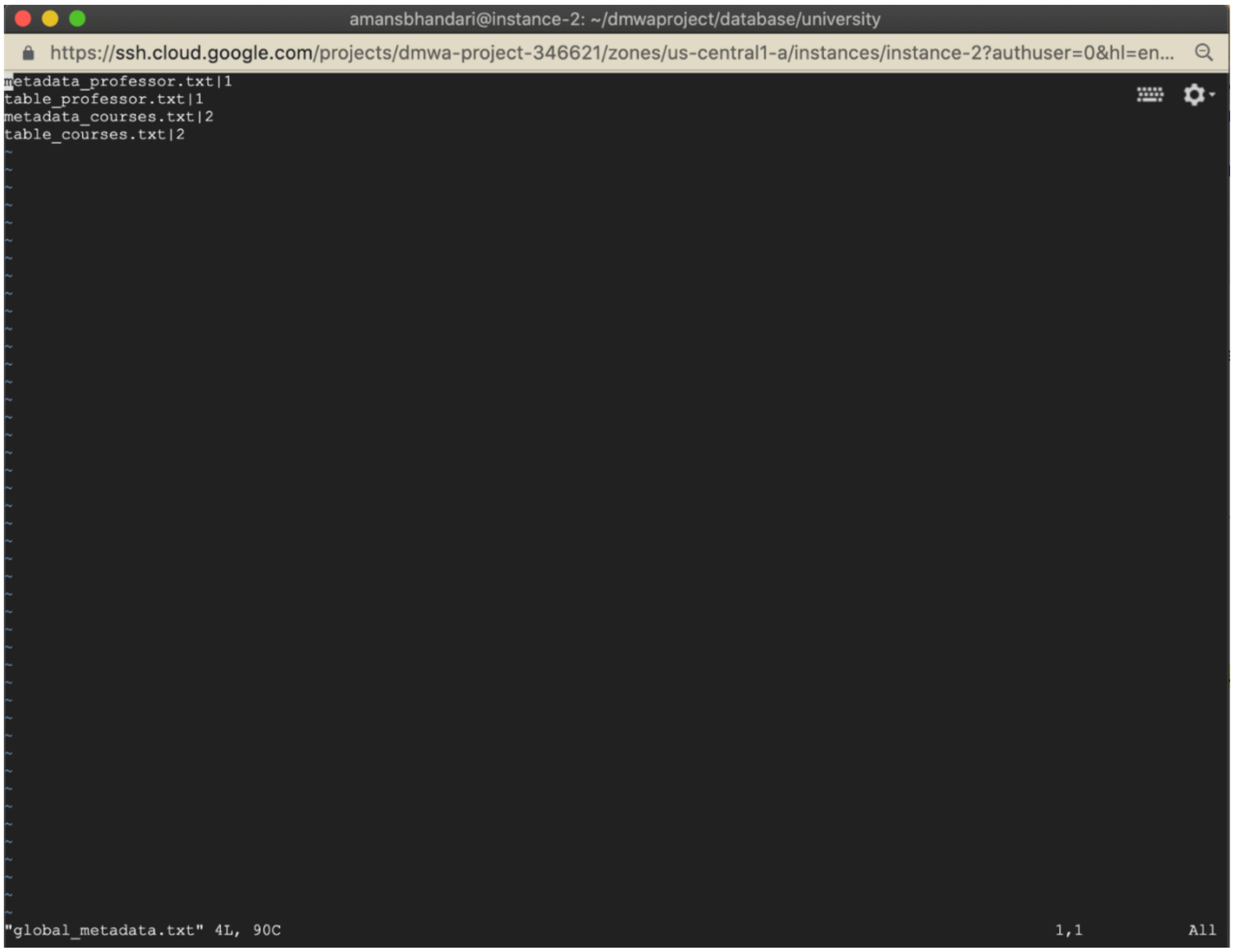


Figure 4: *Structure of global metadata*

The global metadata is kept under each database schema folder with the name global_metadata.txt and one example on the content is shown below in figure 5.



amansbhandari@instance-2: ~/dmwaproject/database/university

https://ssh.cloud.google.com/projects/dmwa-project-346621/zones/us-central1-a/instances/instance-2?authuser=0&hl=en...

```
metadata_professor.txt|1
table_professor.txt|1
metadata_courses.txt|2
table_courses.txt|2
~
"global_metadata.txt" 4L, 90C                                   1,1          All
```

Figure 6: *Example of global metadata*

5. Table's Data

The table rows or the data is kept in a separate file with .txt extension. The name of the file starts with table_ and appends the table name. For example, the name of the tables rows file for courses table would be table_courses.txt. The table data again contains the values pipe separated like previously discussed document. The values are listed in the same order as the column names in the metadata file of the table. Figure 7 and 8 gives more clear picture on how the table rows are stored.

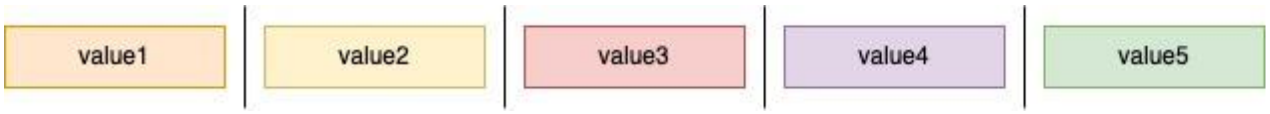
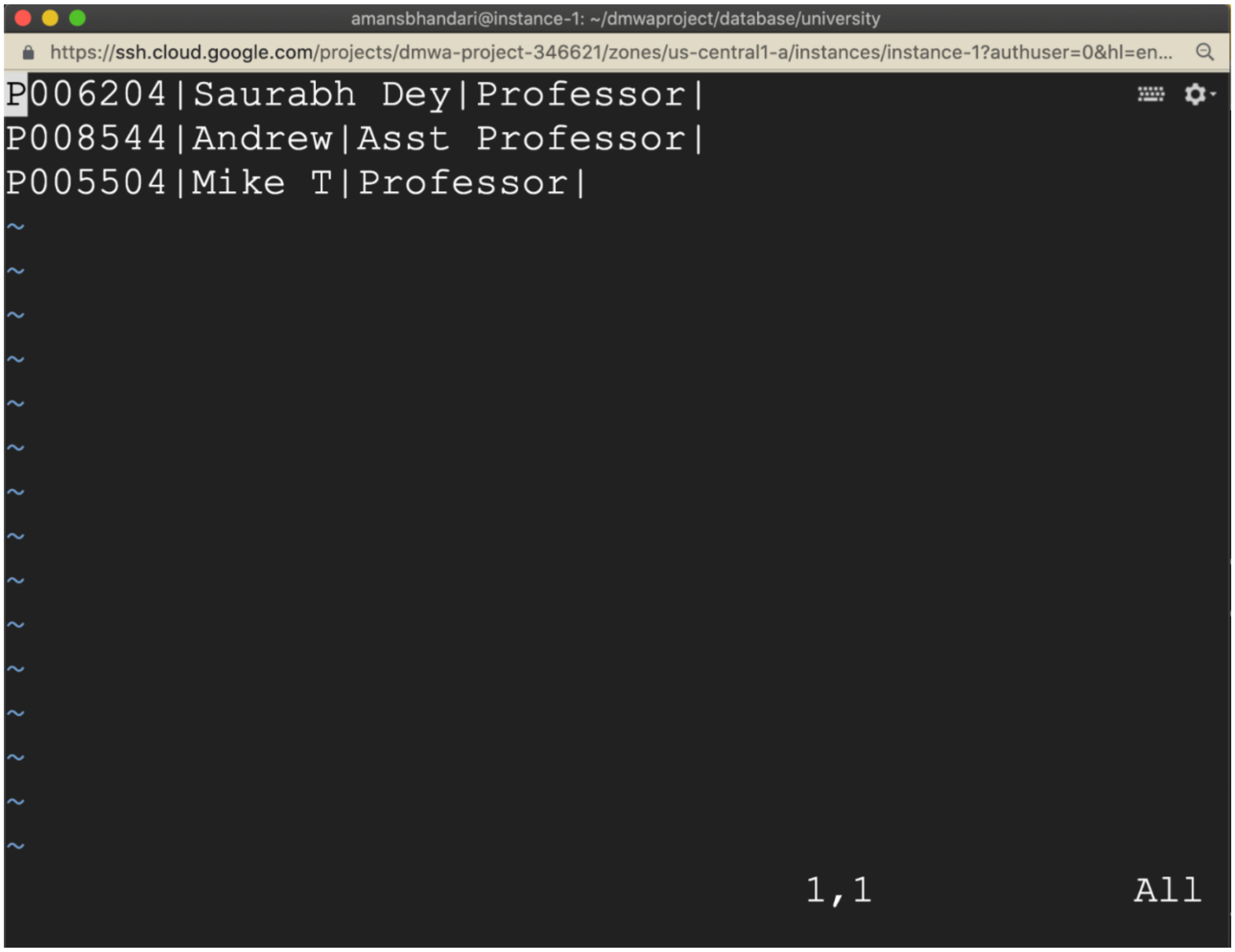


Figure 7: *Table rows structure*



amansbhandari@instance-1: ~/dmwaproject/database/university

https://ssh.cloud.google.com/projects/dmwa-project-346621/zones/us-central1-a/instances/instance-1?authuser=0&hl=en...

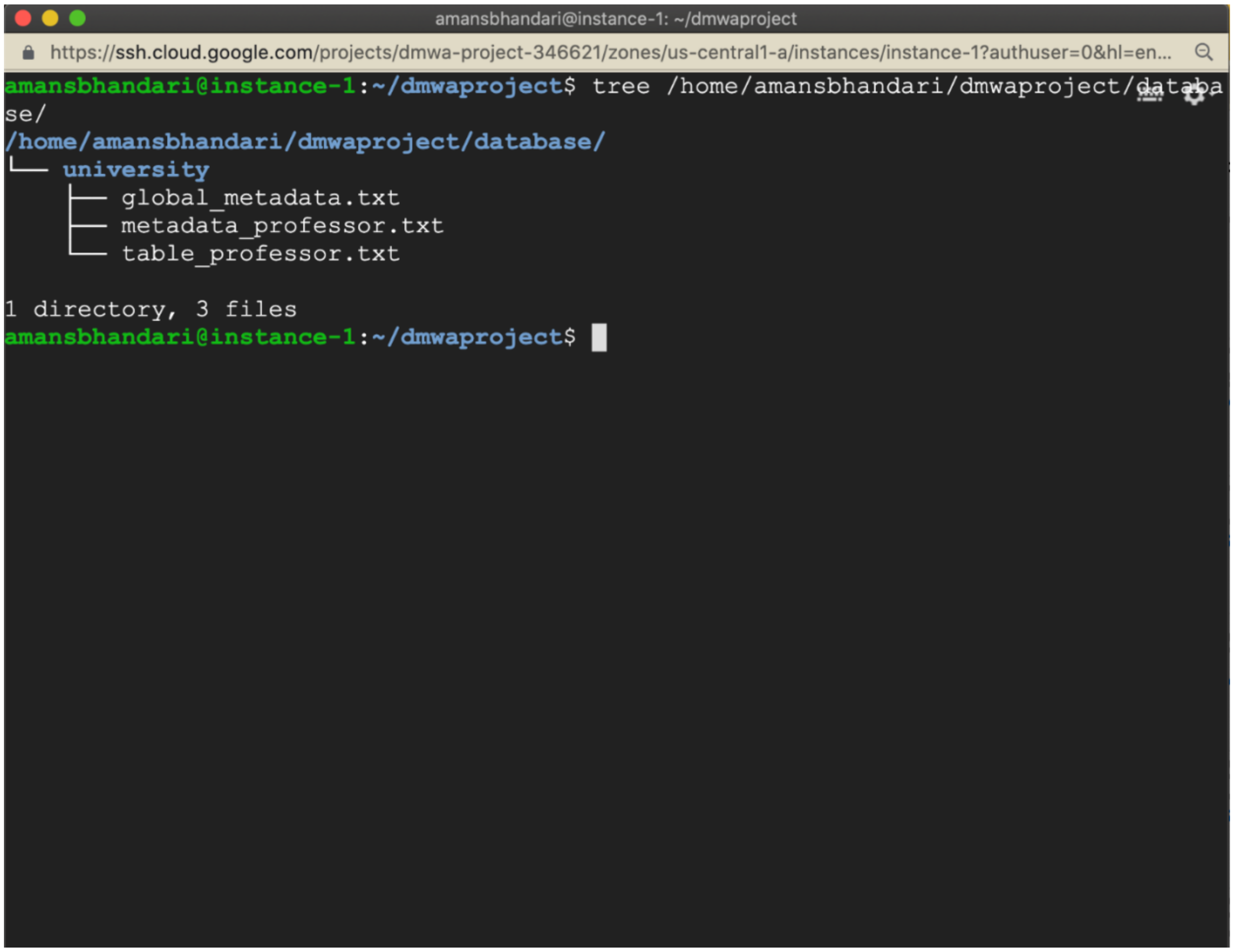
```
P006204|Saurabh Dey|Professor|
P008544|Andrew|Asst Professor|
P005504|Mike T|Professor|
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
                                    1,1           All
```

Figure 8: *Example of table rows structure*

6. File storage structure in DBMS

In this section we will understand that how the directory and file storage look like in our designed DBMS. The root directory is named 'database' and all the tables and metadata is stored under this directory. Under this directory all the database schema directory exists. On creation of each database schema, a directory will be created inside 'database' folder with same name as database schema. Once the table is created, the local metadata file for that table will be created under the database schema folder. Same way, on insertion of the first row, a 'table_<tablename>.txt' will be created inside the same folder. The global metadata file will also be saved under the database schema folder.

Figure 9 shows the directory and file structure in instance1 for the database university. Figure 10 shows the directory structure of second instance for the university database.

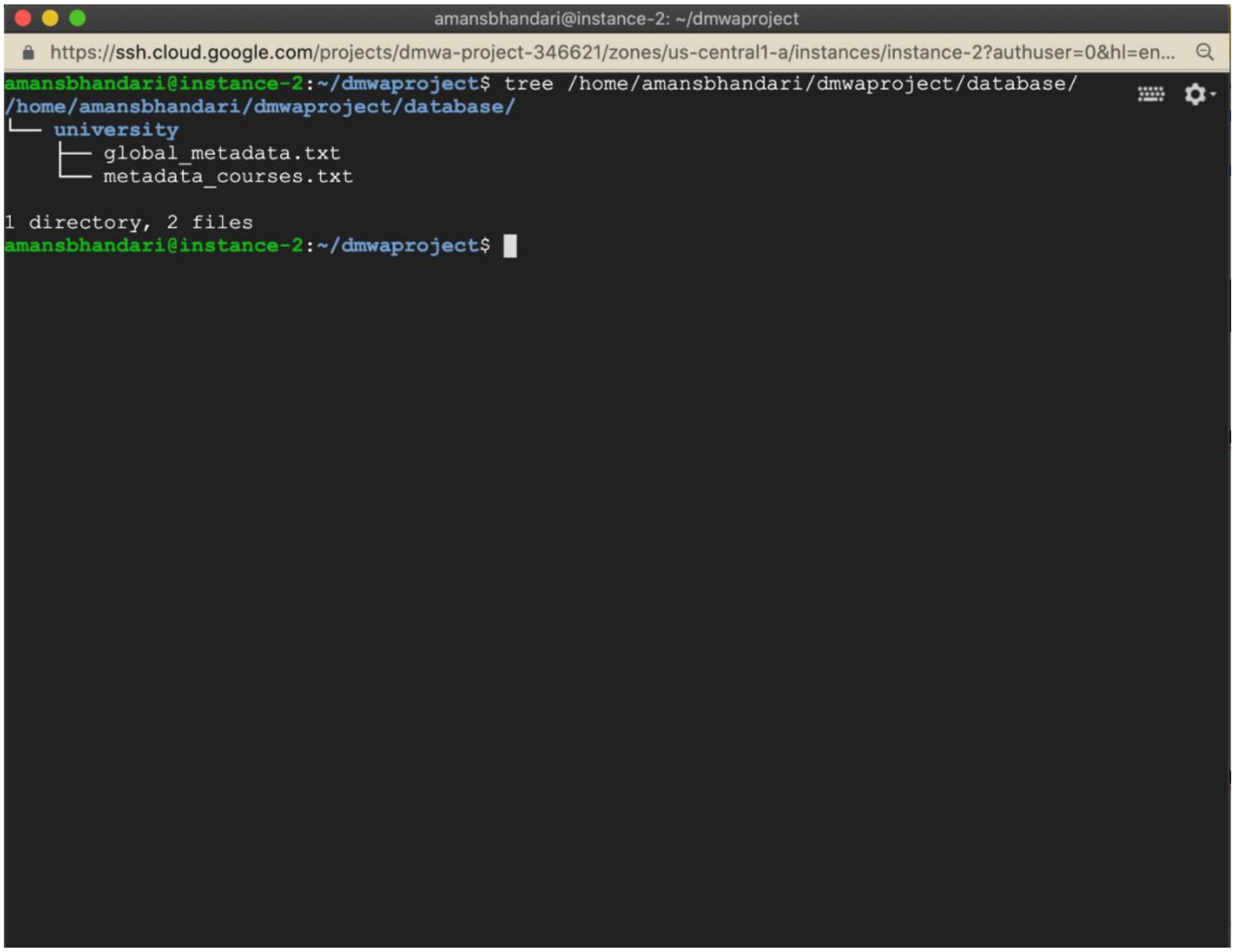


amansbhandari@instance-1: ~/dmwaproject

https://ssh.cloud.google.com/projects/dmwa-project-346621/zones/us-central1-a/instances/instance-1?authuser=0&hl=en...

```
amansbhandari@instance-1:~/dmwaproject$ tree /home/amansbhandari/dmwaproject/databa
se/
/home/amansbhandari/dmwaproject/database/
└── university
    ├── global_metadata.txt
    ├── metadata_professor.txt
    └── table_professor.txt

1 directory, 3 files
amansbhandari@instance-1:~/dmwaproject$
```

Figure 9: *Directory structure on Instance 1*

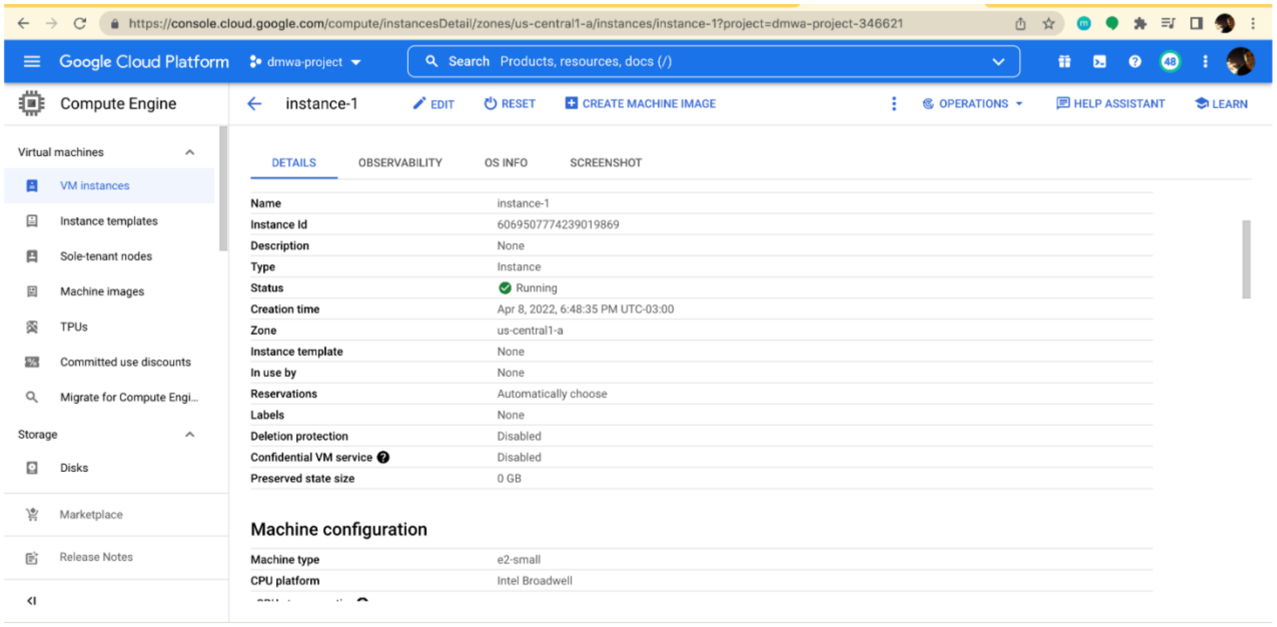

amansbhandari@instance-2: ~/dmwaproject
https://ssh.cloud.google.com/projects/dmwa-project-346621/zones/us-central1-a/instances/instance-2?authuser=0&hl=en...

```
amansbhandari@instance-2:~/dmwaproject$ tree /home/amansbhandari/dmwaproject/database/
/home/amansbhandari/dmwaproject/database/
└── university
    ├── global_metadata.txt
    └── metadata_courses.txt

1 directory, 2 files
amansbhandari@instance-2:~/dmwaproject$
```

Figure 10: *Directory Structure in Instance 2*

7. Installing and Accessing VM Instances

The two VM instances were created in this effort. The configuration of these two ubuntu instances were kept exactly same to avoid biasing on the performance and efficiency. In figure 11 to 16, the configuration of the instances can be observed. To access the two remote machine from local through ssh, created a public and private key command 'ssh-keygen -t rsa -b 4096 -C <username>'. The private key was of openssh and needed a conversion of type RSA and hence next command to run was 'ssh-keygen -p -f ~/.ssh/id_rsa -m pem'.

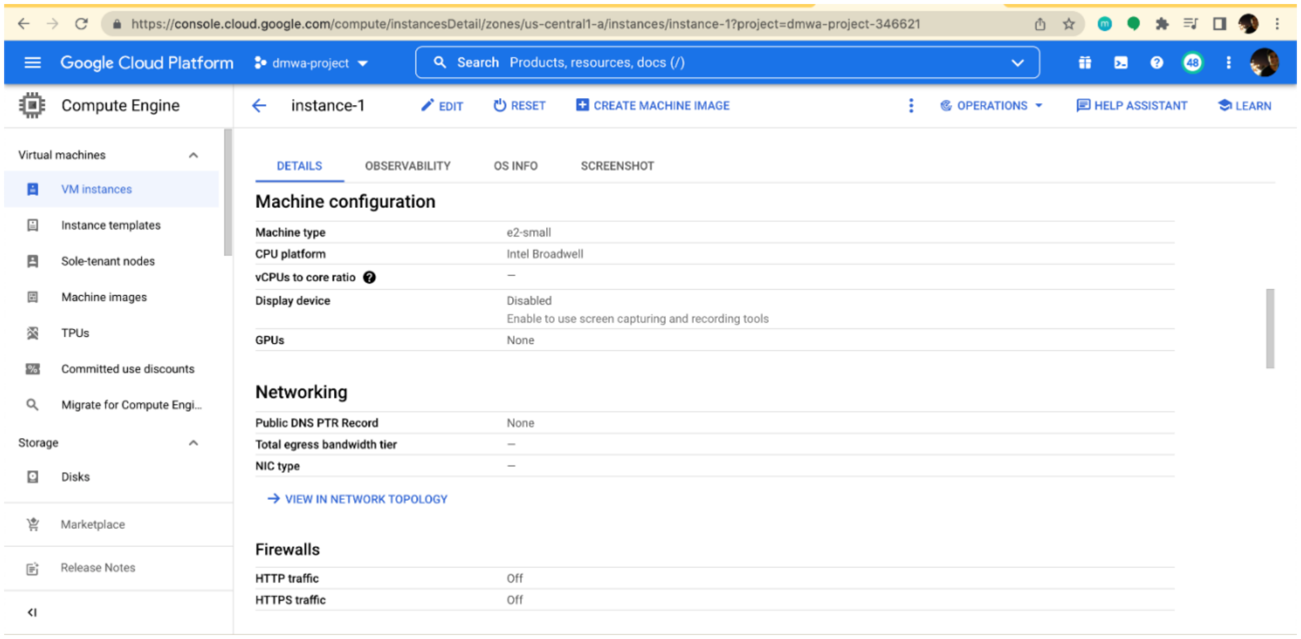

https://console.cloud.google.com/compute/instancesDetail/zones/us-central1-a/instances/instance-1?project=dmwa-project-346621

Name	instance-1
Instance Id	6069507774239019869
Description	None
Type	Instance
Status	Running
Creation time	Apr 8, 2022, 6:48:35 PM UTC-03:00
Zone	us-central1-a
Instance template	None
In use by	None
Reservations	Automatically choose
Labels	None
Deletion protection	Disabled
Confidential VM service	Disabled
Preserved state size	0 GB

Machine configuration

Machine type	e2-small
CPU platform	Intel Broadwell

Figure 11: *VM Instance Configuration*


https://console.cloud.google.com/compute/instancesDetail/zones/us-central1-a/instances/instance-1?project=dmwa-project-346621

Machine configuration

Machine type	e2-small
CPU platform	Intel Broadwell
vCPUs to core ratio	–
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	None

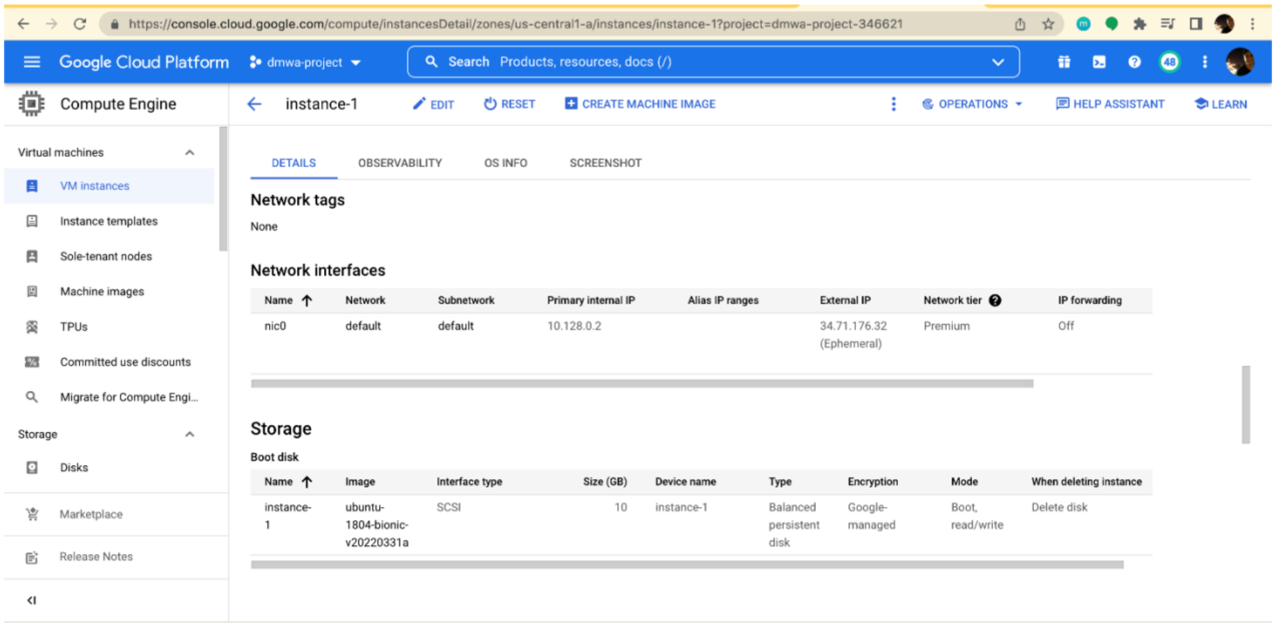
Networking

Public DNS PTR Record	None
Total egress bandwidth tier	–
NIC type	–

Firewalls

HTTP traffic	Off
HTTPS traffic	Off

Figure 12: *VM Instance Configuration*



Network tags

None

Network interfaces

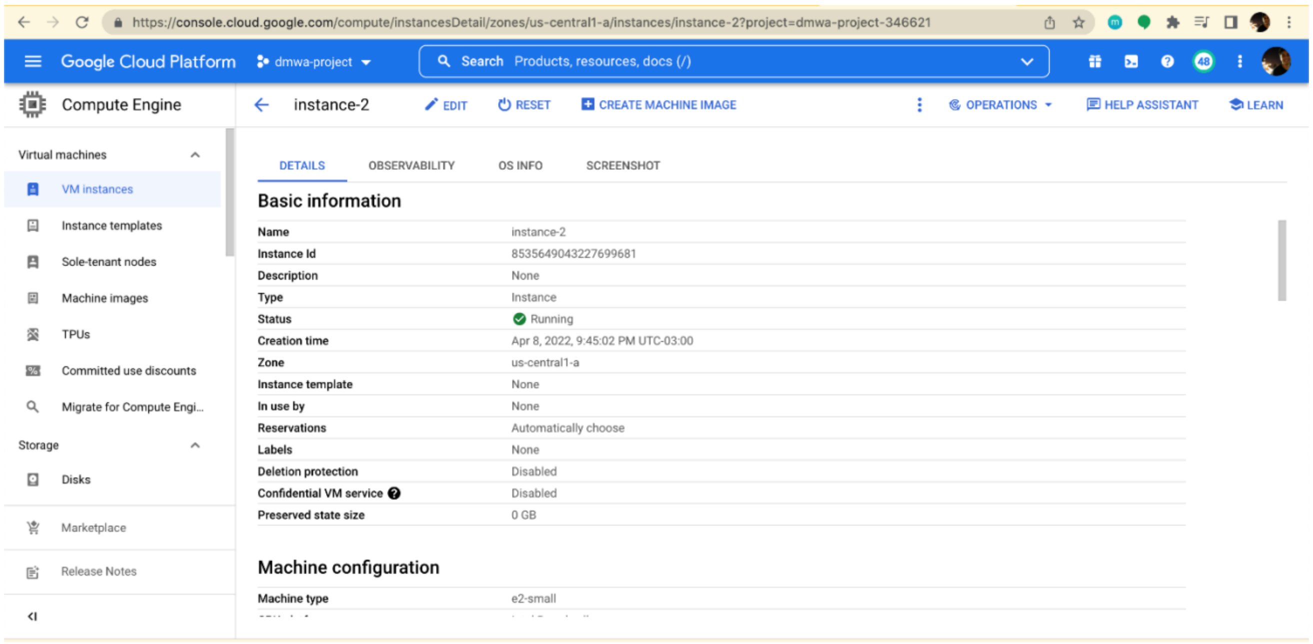
Name	Network	Subnetwork	Primary internal IP	Alias IP ranges	External IP	Network tier	IP forwarding
nic0	default	default	10.128.0.2		34.71.176.32 (Ephemeral)	Premium	Off

Storage

Boot disk

Name	Image	Interface type	Size (GB)	Device name	Type	Encryption	Mode	When deleting instance
instance-1	ubuntu-1804-bionic-v20220331a	SCSI	10	instance-1	Balanced persistent disk	Google-managed	Boot, read/write	Delete disk

Figure 13: *VM Instance Configuration*



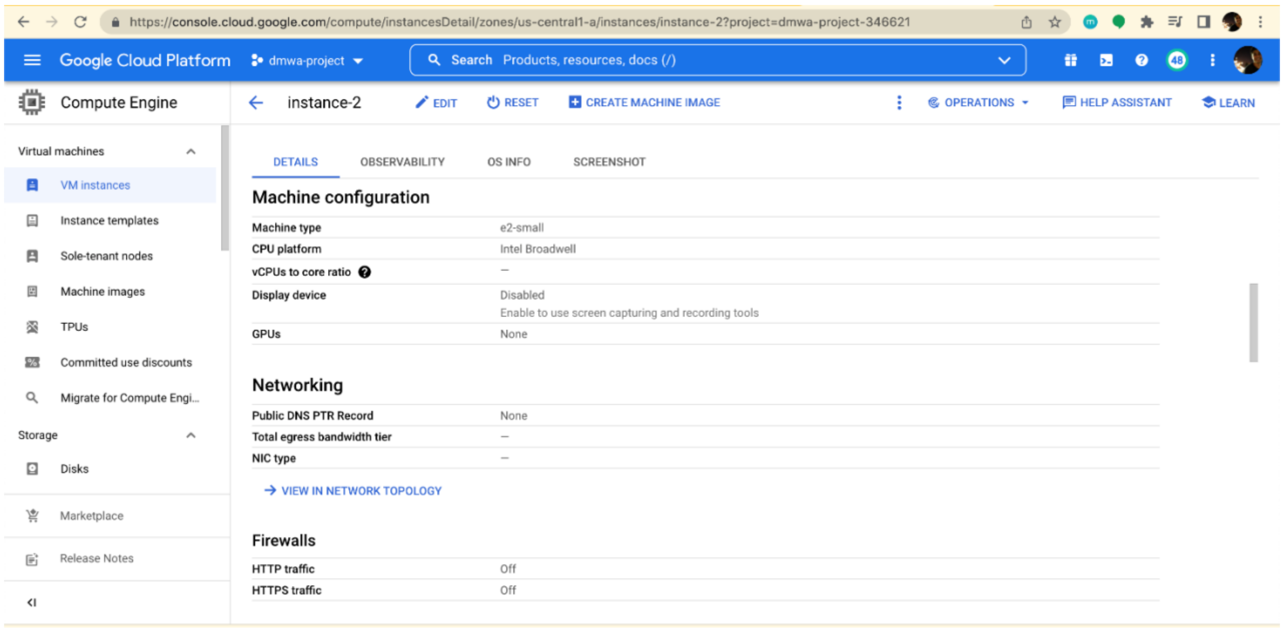
Basic information

Name	instance-2
Instance Id	8535649043227699681
Description	None
Type	Instance
Status	✓ Running
Creation time	Apr 8, 2022, 9:45:02 PM UTC-03:00
Zone	us-central1-a
Instance template	None
In use by	None
Reservations	Automatically choose
Labels	None
Deletion protection	Disabled
Confidential VM service	Disabled
Preserved state size	0 GB

Machine configuration

Machine type	e2-small

Figure 14: *VM Instance Configuration*



Machine configuration

Machine type	e2-small
CPU platform	Intel Broadwell
vCPUs to core ratio	–
Display device	Disabled Enable to use screen capturing and recording tools
GPUs	None

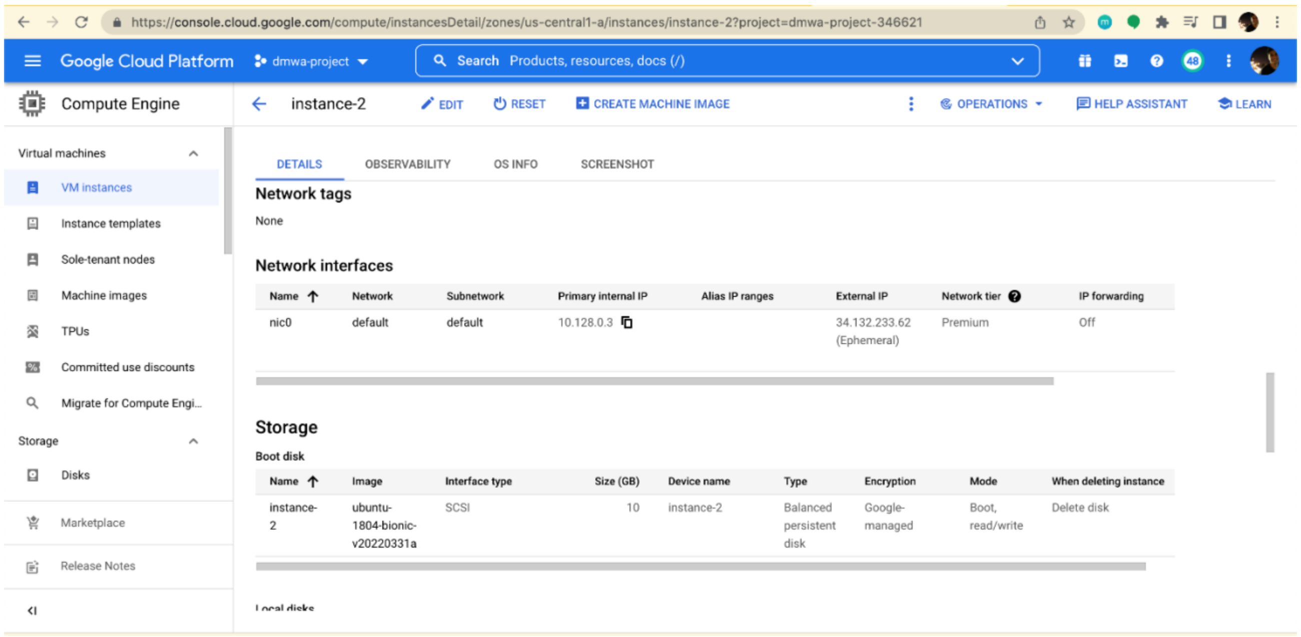
Networking

Public DNS PTR Record	None
Total egress bandwidth tier	–
NIC type	–

Firewalls

HTTP traffic	Off
HTTPS traffic	Off

Figure 15: *VM Instance Configuration*



Network tags

None

Network interfaces

Name ↑	Network	Subnetwork	Primary internal IP	Alias IP ranges	External IP	Network tier	IP forwarding
nic0	default	default	10.128.0.3		34.132.233.62 (Ephemeral)	Premium	Off

Storage

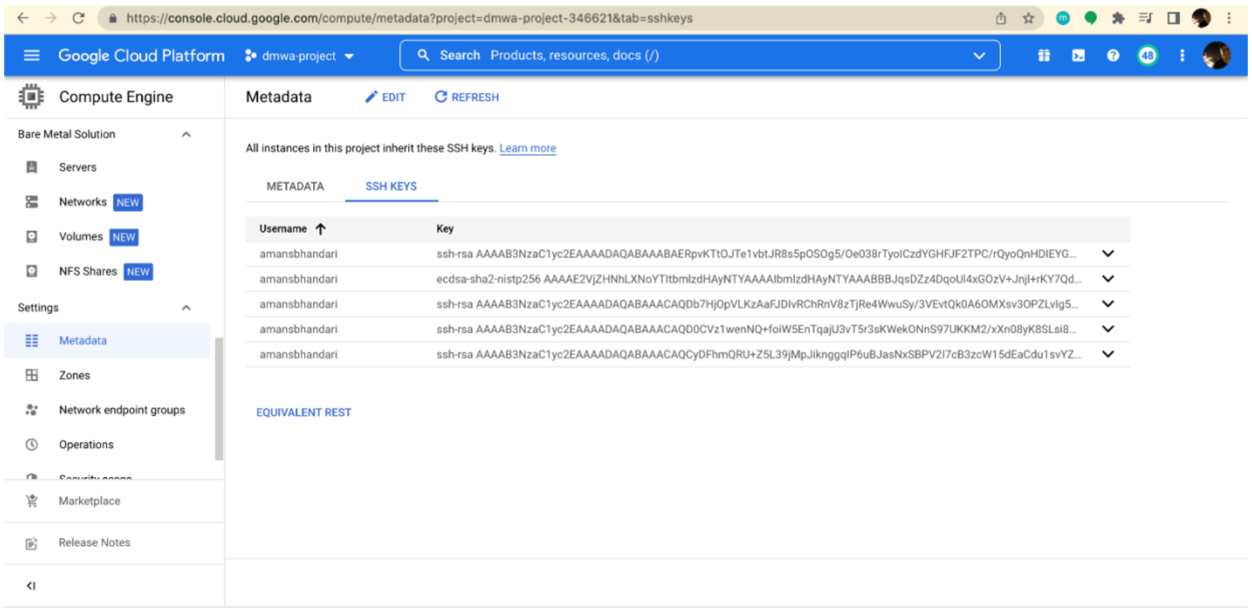
Boot disk

Name ↑	Image	Interface type	Size (GB)	Device name	Type	Encryption	Mode	When deleting instance
instance-2	ubuntu-1804-bionic-v20220331a	SCSI	10	instance-2	Balanced persistent disk	Google-managed	Boot, read/write	Delete disk

Local disks

Figure 16: *VM Instance Configuration*

Once keys are generated it needs to be added to the metadata of the created project (figure 17). We need to the add the public keys of all the machines who wants to access these two instances. Here, we need to make sure to add the keys of both the instances as well so that they can access to each other for distributed database.



All instances in this project inherit these SSH keys. Learn more

Username	Key
amansbhandari	ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAERpvKTtOJTe1vbtJR8s5pOSOg5/Oe038rTyolCzdYGHFJF2TPC/rQyoQnHDIEYG...
amansbhandari	ecdsa-sha2-nistp256 AAAAE2VjZHNhLXNoYTItbmlzdHAyNTYAAAAIbmlzdHAyNTYAAABBBJqsDZz4DqoUl4xGOzV+Jnjl+rKY7Qd...
amansbhandari	ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQDb7HjOpVLKzAaFJDIvRChRnV8zTjRe4WwuSy/3VEvtQk0A6OMXsv3OPZLvlg5...
amansbhandari	ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQD0CVz1wenNQ+foiW5EnTqajU3vT5r3sKWekONnS97UKKM2/xXn08yK8SLsi8...
amansbhandari	ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQCyDFhmQRU+Z5L39jMpJiknggqIP6uBJasNxSBPV2I7cB3zcW15dEaCdu1svYZ...

Figure 17: *VM allowed SSH keys*

8. Making the database distributed

The distribution layer in our architecture keeps track of the resources and in which instance do they exists. It takes the help of global metadata to locate the resources. Other modules don't care about the distribution of resources and just focusses on their part. They communicate with distribution layer to retrieve and store the data. Figure 18 shows the architecture of this layer.

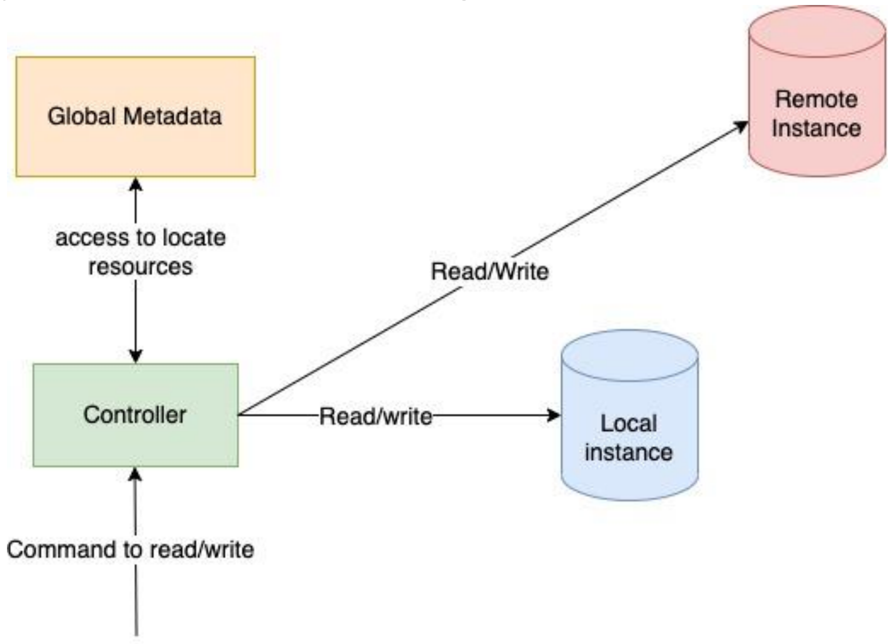


Figure 18: *Architecture diagram of Distribution layer*

Figure 19 shows the algorithm of reading and writing the files in a distributed system.

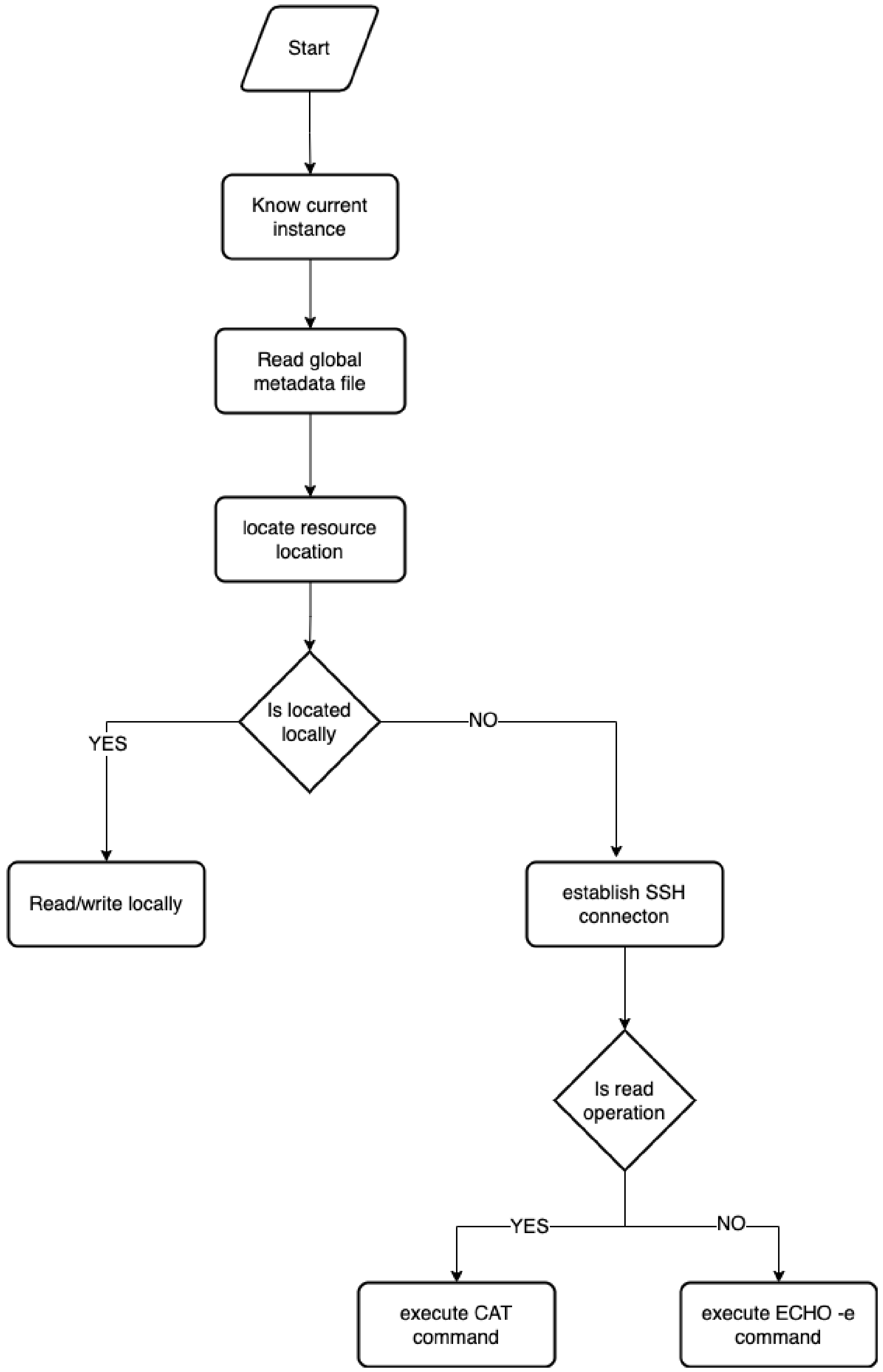
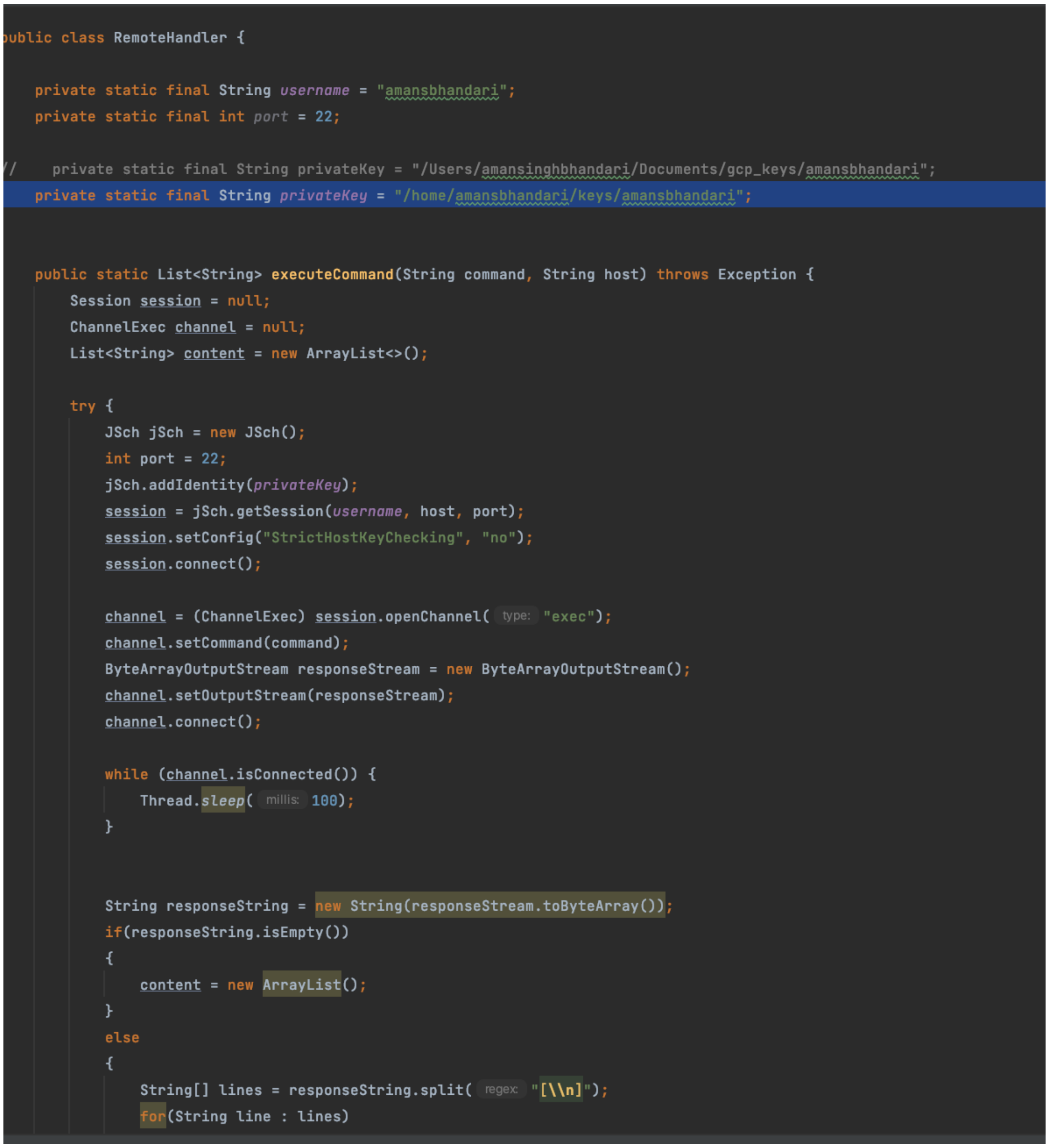


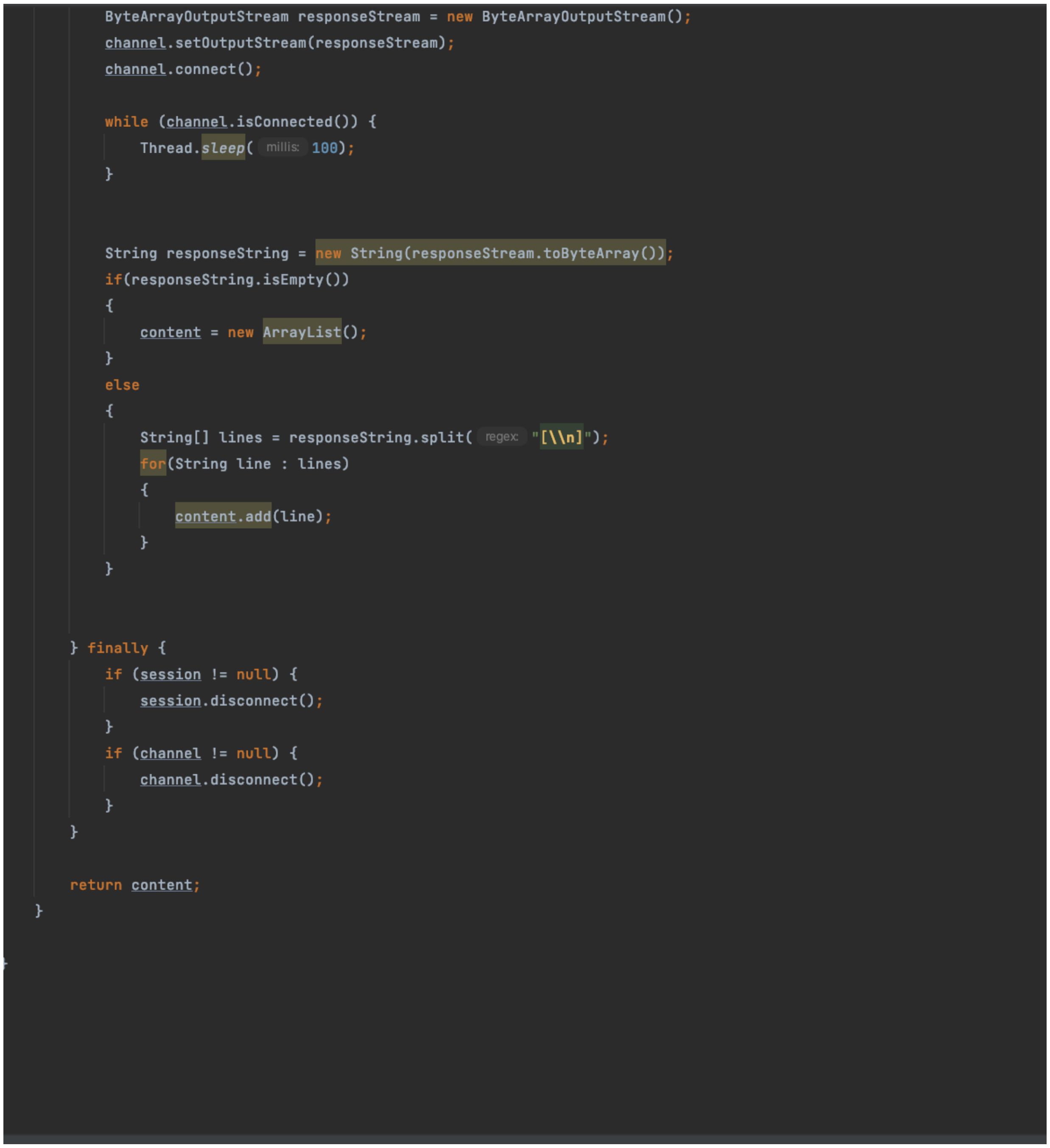
Figure 19: Algorithm of distribution layer

Figure 20 and 21 shows the code according to the above algorithm. Here we used JSCH[1] library to build the SSH connection and execute the Linux command on the remote machine. Also, refer to the code [2] to implement the SSH connection and command executor.



```
ublic class RemoteHandler {

   private static final String username = "amansbhandari";
   private static final int port = 22;

//    private static final String privateKey = "/Users/amansinghbhandari/Documents/gcp_keys/amansbhandari";
   private static final String privateKey = "/home/amansbhandari/keys/amansbhandari";


   public static List<String> executeCommand(String command, String host) throws Exception {
       Session session = null;
       ChannelExec channel = null;
       List<String> content = new ArrayList<>();

       try {
           JSch jSch = new JSch();
           int port = 22;
           jSch.addIdentity(privateKey);
           session = jSch.getSession(username, host, port);
           session.setConfig("StrictHostKeyChecking", "no");
           session.connect();

           channel = (ChannelExec) session.openChannel( type: "exec");
           channel.setCommand(command);
           ByteArrayOutputStream responseStream = new ByteArrayOutputStream();
           channel.setOutputStream(responseStream);
           channel.connect();

           while (channel.isConnected()) {
               Thread.sleep( millis: 100);
           }


           String responseString = new String(responseStream.toByteArray());
           if(responseString.isEmpty())
           {
               content = new ArrayList();
           }
           else
           {
               String[] lines = responseString.split( regex: "[\\n]");
               for(String line : lines)
```

Figure 20: Code for to build SSH connection and execute commands



```
            ByteArrayOutputStream responseStream = new ByteArrayOutputStream();
            channel.setOutputStream(responseStream);
            channel.connect();

            while (channel.isConnected()) {
                Thread.sleep( millis: 100);
            }


            String responseString = new String(responseStream.toByteArray());
            if(responseString.isEmpty())
            {
                content = new ArrayList();
            }
            else
            {
                String[] lines = responseString.split( regex: "[\\n]");
                for(String line : lines)
                {
                    content.add(line);
                }
            }


        } finally {
            if (session != null) {
                session.disconnect();
            }
            if (channel != null) {
                channel.disconnect();
            }
        }

        return content;
    }
```

Figure 21: Code for to build SSH connection and execute commands

9. Executing SQL Commands and Queries.

The queries inputs to the query executor layer through a POJO class. For every type of query, project contains a separate POJO class contain different data fields and constructor. Figure 22 shows all such classes.

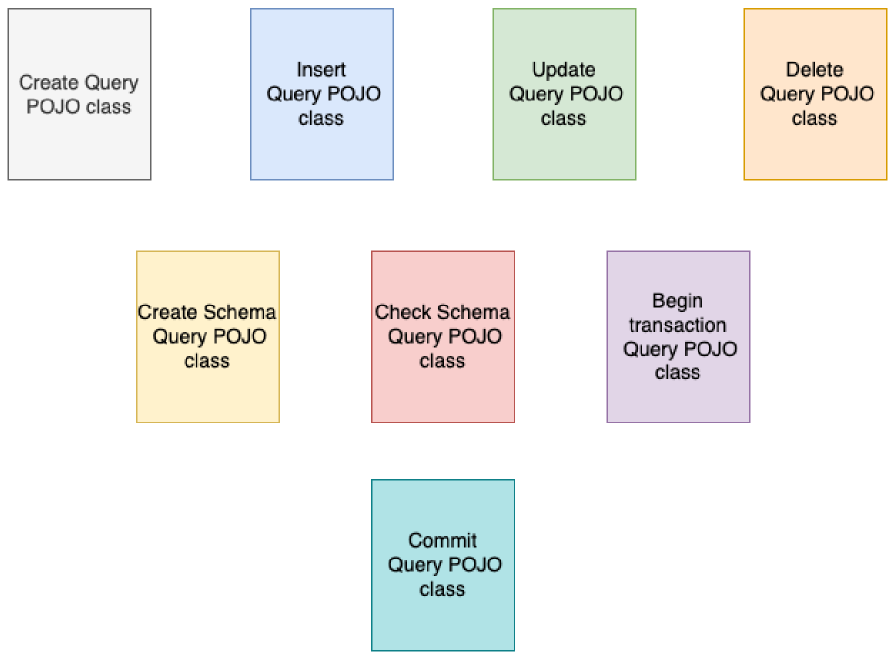


Figure 22: POJO classes

With these pojo class, the handler has all the necessary details with them to execute the specific query. The handler works closely with the distribution layer to read/write the file. This layer accesses the local metadata and table file frequently to read/update the information in them.

10.Dumps

10.1 Use case:

- For regenerate the database on another machine
- Back up the database

10.2 Goals

Goals: when the use execute the dump query, create a dump SQL file which generate the SQL commands file to recreate the current database.

10.3 Process flow

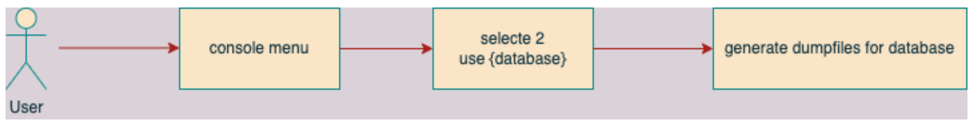
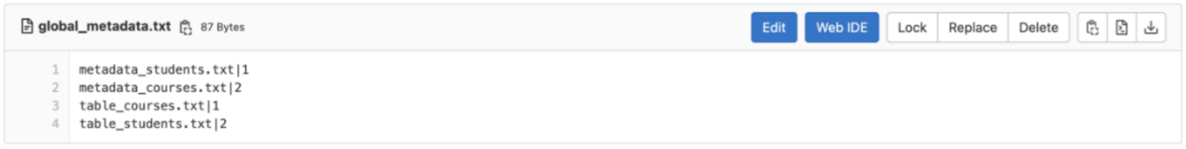


Figure 23: *process flow of generate dump file*

10.4 Pseudo code

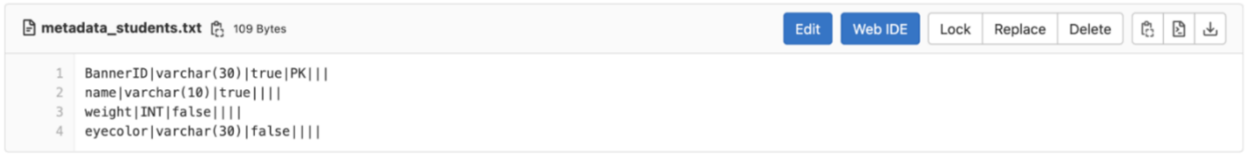
```
input: int command = 2, String databaseName = {some database name}
output: list of SQLs queries store in the sql files.
user register and login through the console. And select command 2 from console menu.
If (input Command == 2) {
    Enter database name
    Triger the generate Dump function (database name)
}
Generate Dump () {
    Read global metadata of that database.
    Count how many tables in that database.
    Get information where the tables stored.
    Create a list to store the tables.
    Find and Read the Tables's local metadata to get the information about
    Columns.
    Sort the list base on relation between tables. The independent tables should be created
    first. And the tables dependent on other tables will create after the that table.
    Create and write "create {database}" and use {database} into dump file
    For (tables of that database in the sorted list) {
        create and write create table SQL into the dump file.
        Find the table by reading the global metadata
        Read records of the table.
        Then create and write insert SQL into the dump file.
    }
}
```

10.5 Screenshot



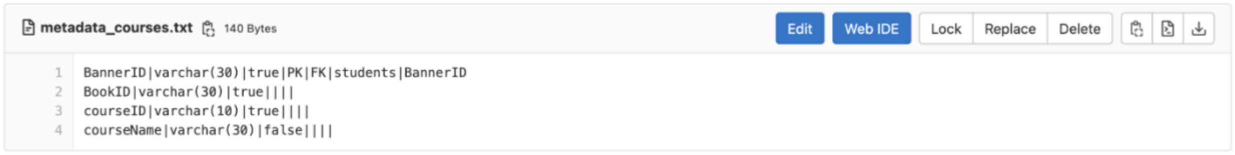
```
metadata_students.txt|1
metadata_courses.txt|2
table_courses.txt|1
table_students.txt|2
```

Figure 24: *global metadata of the distributed database that indicate where the tables stored*



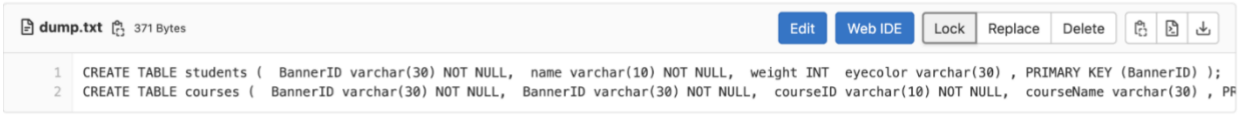
```
BannerID|varchar(30)|true|PK|||
name|varchar(10)|true||||
weight|INT|false||||
eyecolor|varchar(30)|false||||
```

Figure 25: *Local metadata of the student table which shows it do not depend on another table*



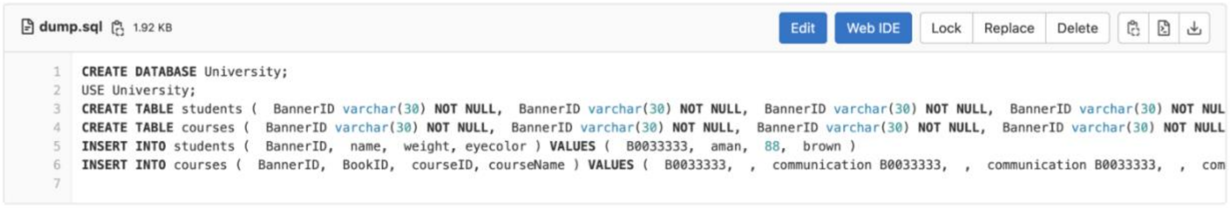
```
BannerID|varchar(30)|true|PK|FK|students|BannerID
BookID|varchar(30)|true||||
courseID|varchar(10)|true||||
courseName|varchar(30)|false||||
```

Figure 26: Local metadata of the courses table which shows it depends on the student table



```
CREATE TABLE students (  BannerID varchar(30) NOT NULL,  name varchar(10) NOT NULL,  weight INT  eyecolor varchar(30) , PRIMARY KEY (BannerID) );
CREATE TABLE courses (  BannerID varchar(30) NOT NULL,  BannerID varchar(30) NOT NULL,  courseID varchar(10) NOT NULL,  courseName varchar(30) , PR
```

Figure 27: Structure dump file



```
CREATE DATABASE University;
USE University;
CREATE TABLE students (  BannerID varchar(30) NOT NULL,  BannerID varchar(30) NOT NULL,  BannerID varchar(30) NOT NULL,  BannerID varchar(30) NOT NUL
CREATE TABLE courses (  BannerID varchar(30) NOT NULL,  BannerID varchar(30) NOT NULL,  BannerID varchar(30) NOT NULL,  BannerID varchar(30) NOT NULL
INSERT INTO students (  BannerID,  name,  weight, eyecolor ) VALUES (  B0033333,  aman,  88,  brown )
INSERT INTO courses (  BannerID,  BookID,  courseID, courseName ) VALUES (  B0033333,  ,  communication B0033333,  ,  communication B0033333,  ,  com

```

Figure 28: structure and data dump file

11 Logs

11.1 Use case:

- For generating the analytics
- Checking the status of the distributed system whether it is consistent or not
- For debugging or recover purpose

11.2 Function

A log function will be monitoring and record the status of the distributed database. There are three types of logs. Query log which will record each query execution like whether it is successful and time stamp and other essential information about the query. The database log which will record the states of the database. It is snapshot of the database states. At that time, how many table in that database and how many records in the tables. For the event logs, it will log all activities that done in the database. Such as crash report, query execution and transaction. And analytics function will read the logs to generate report

11.3 Data structure

The format of the logs is Json file.

We use the JSON to store the logs, because it is easy to retrive the information by key. It is coinvent for generate database analytics. And it is easy to transfer between the JSON string to String. It can be used when we want to store the information into in the database.

The logs will be store in both instances. So, every activity for instances will be logged.

11.4 Process flow

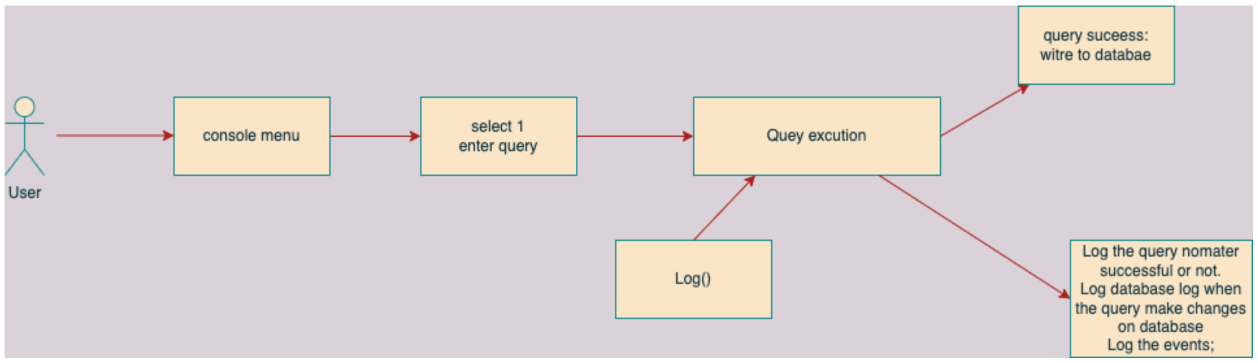


Figure 29: *Process flow of Log management*

11.5 Pseudo code

Input: A object call Log Parameters. The Log Parameters has the attributes:

- String event.
- String excutionTime;
- String queryString;
- String user;
- String timeStamp;
- String type;
- String condition;
- String columns;
- String values;
- String database;
- String table;
- int numberOfRowsChanges;
- int numberOfTables;
- String[]tableNames;
- int[] numberOfRows;
- String isSuccessful;
- String crashReport;

Output :event logs File, query logs file and datbase log file

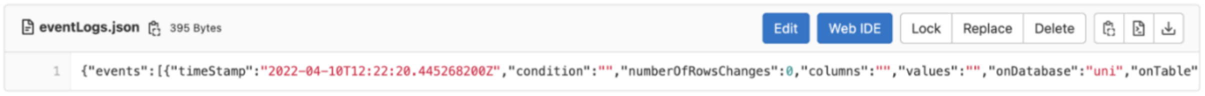
Algorithm:
When the query executed. Gather all the information associated with the query

- Calculate the time of excution (end time minus start time)
- Record the timestamp when the query is executed
- Store use information
- Table name
- Database name
- Query String
- And other Log parameters

Pass the log parameter to the log function (Log parameters)

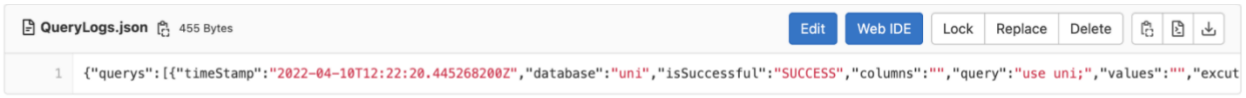
```
    Create JSON object Event
    Store it into instance one
    Store it into instance two
    Create JSON object Querys
    Store it into instance one
    Store it into instance two
    Create JSON object Database
    Store it into instance one
    Store it into instance two
```

11.6 Screenshot



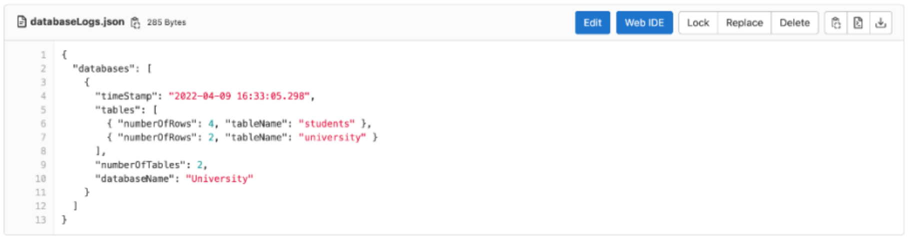
```
{"events":[{"timeStamp":"2022-04-10T12:22:20.445268200Z","condition":"","numberOfRowsChanges":0,"columns":"","values":"","onDatabase":"uni","onTable"
```

Figure 30: sample event Logs file



```
{"querys":[{"timeStamp":"2022-04-10T12:22:20.445268200Z","database":"uni","isSuccessful":"SUCCESS","columns":"","query":"use uni;","values":"","excut
```

Figure 31: sample Query Log file



```
{
  "databases": [
    {
      "timeStamp": "2022-04-09 16:33:05.298",
      "tables": [
        { "numberOfRows": 4, "tableName": "students" },
        { "numberOfRows": 2, "tableName": "university" }
      ],
      "numberOfTables": 2,
      "databaseName": "University"
    }
  ]
}
```

Figure 32: sample database log file

12.Analytics

12.1 Use case:

1. Have a statistics insight about the database
2. For optimizing the system. Based on the analytics to adopt the load balance to improve the efficient of the system.

12.2 Function:

This function will generate essential analytics about the database.

- Number of the queries are submitted by the user on which database running on which VM
- Number of the {Type} queries are successful executed by the user on which tables

12.3 Data structure

Plain text file and print to the console

12.4 Process flow

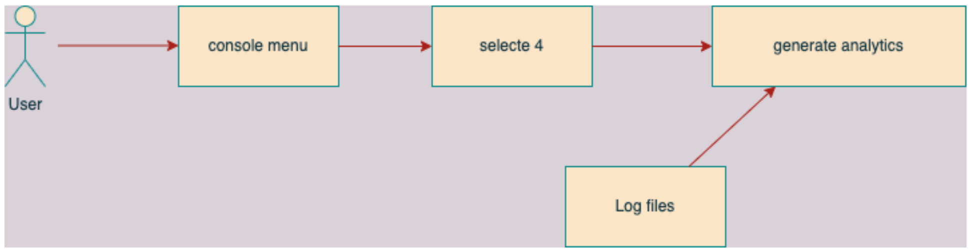
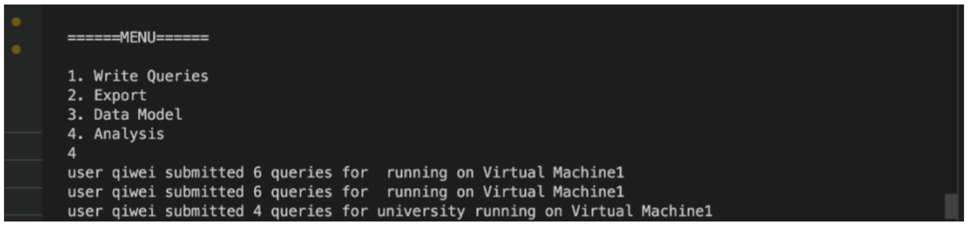


Figure 33: process flow of generating database analytics

12.5 Pseudo code

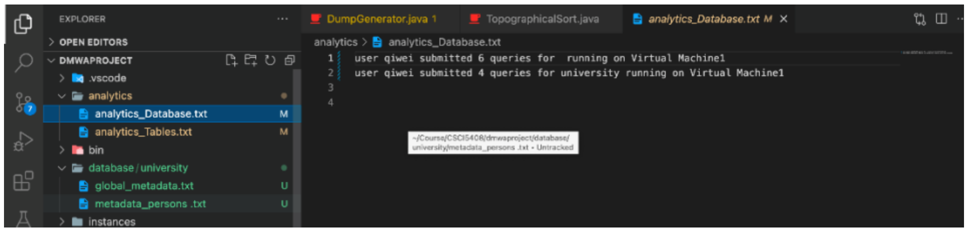
```
Read the database logs and count unique name of the database to know number of the database.
Find the queries log file
Read the queries log file
For each database in the system {
        filter with the database name
        Filter with the username
Count number of the query
Store number of the the query in list
        Print out {username} submitted {number of queries} on {database name} running on
        {VM};
Write the print message into the analytics file.
 }
```

12.6 Screenshot:



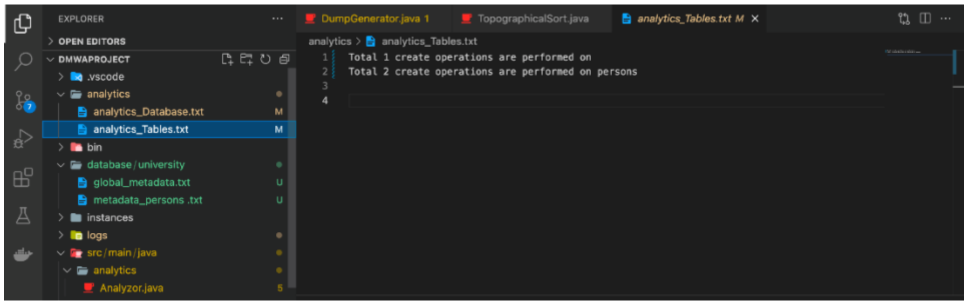
```
======MENU======

1. Write Queries
2. Export
3. Data Model
4. Analysis
4
user qiwei submitted 6 queries for  running on Virtual Machine1
user qiwei submitted 6 queries for  running on Virtual Machine1
user qiwei submitted 4 queries for university running on Virtual Machine1
```

Figure 34: console print out demonstrates



```
user qiwei submitted 6 queries for  running on Virtual Machine1
user qiwei submitted 4 queries for university running on Virtual Machine1
```

Figure 35: Demo database analytics



```
Total 1 create operations are performed on
Total 2 create operations are performed on persons
```

Figure 36: Demo table analytics

13. Data Modelling – Reverse Engineering

1. Use case

- Generate an Entity Relationship Diagram (ERD) of the current state of the database.
- Graphical representation of tables, columns, relationships, and cardinality of a database.

2. Definition
   Data modeling and Reverse Engineering is creating a data model of a database from its scripts and stored data and representing it in a graphical form. The graphical representation or ERD includes the entities, their relationships, and cardinality. It includes the tables, columns, and their relationships with other tables in a database. Cardinality is the measure of elements in a given set and cardinality in ERD shows the relationship between rows of one table with another in a numerical form like one-to-one, one-to-many and many-to-many.

3. Implementation

   Implementation classes for Reverse Engineering:
   a. ReverseEngineering.java
   b. TopographicalSort.java
   c. DrawERD.java
   d. ErdExecutor.java

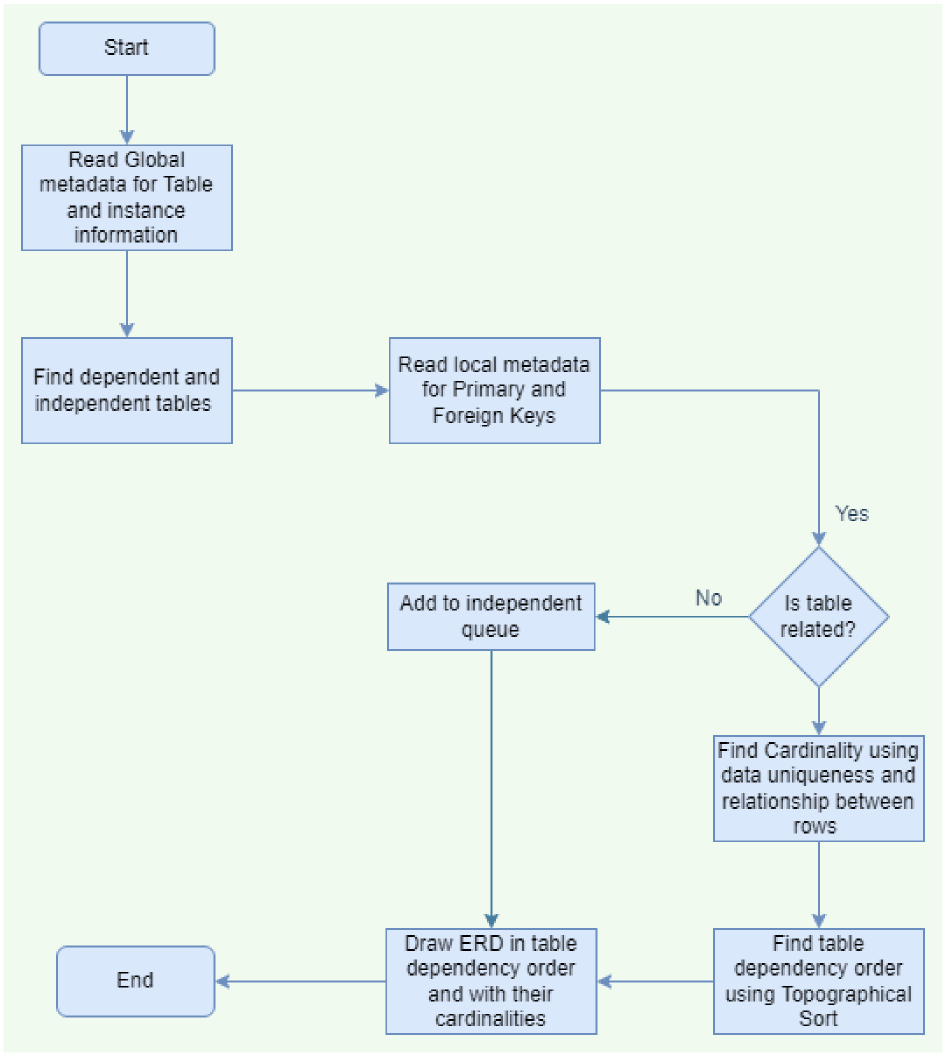

Start
Read Global metadata for Table and instance information
Find dependent and independent tables
Read local metadata for Primary and Foreign Keys
Yes
Is table related?
No
Add to independent queue
Find Cardinality using data uniqueness and relationship between rows
Find table dependency order using Topographical Sort
Draw ERD in table dependency order and with their cardinalities
End

Figure 37: *Flowchart of drawing ERD by reverse engineering*

Figure 37 shows the flowchart for extracting information from the global and local metadata for a database and its table information to create Entity Relationship Diagram.

It includes the following steps:

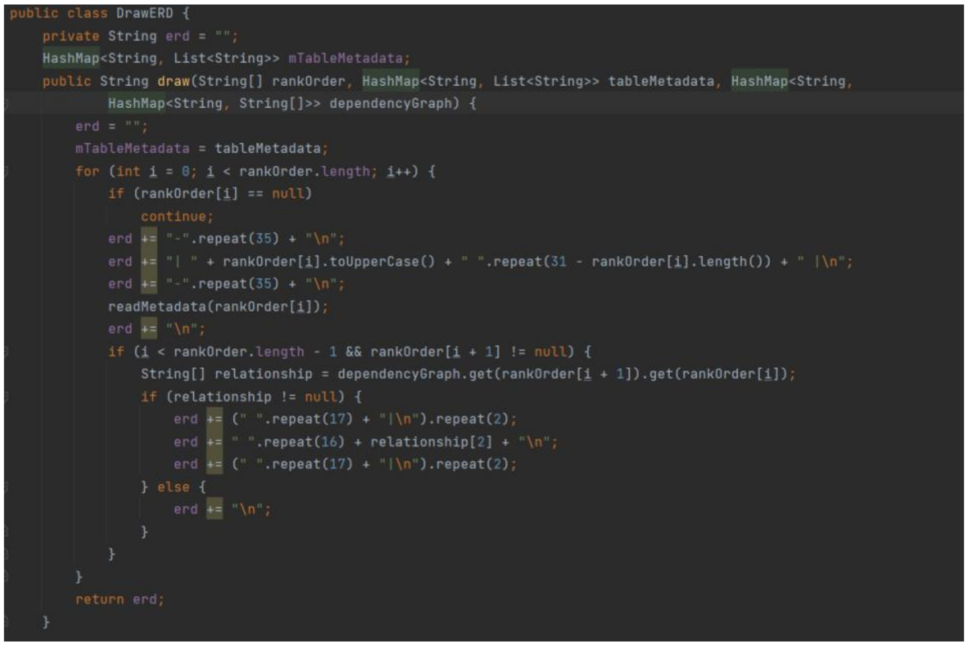
a. Reading the Global metadata for instance-level information of a database to local tables and related local metadata.
b. Accessing the local metadata for column information, constraints, and relationships for including them in the ERD.
c. Find related tables from the metadata and foreign key relationships for creating a dependency graph.
d. Create a dependency graph using Topographical Sort [3], [4] for drawing dependent tables together. TopographicalSort will order the tables considering the tables that have dependencies to arrive after the tables they are dependent on.
e. Draw ERD with table details, identified relationships, and in recognized order.

Psuedo Code:

ErdExecutor.java
```
1. Take input from console for database name
2. Check if database exists, if exists GOTO 3 else GOTO 1
3. Call ReverseEngineering.java class to get rankOrder and dependencyGraph for drawing ERD using DrawERD.java
```

ReverseEngineering.java
```
1. Fetch global metadata for the given database
2. Get table and their instance details
3. Read metadata of each table from their respective instances
4. Find relationships between tables and store it in a hashmap
5. Calculate cardinality of the relationship of two tables by identifying data uniqueness
6. If all table metadata are read, GOTO 7, else GOTO 3
7. Send the dependencyGraph to TopographicalSort.java for generating sort order and create rankedTables
8. Return rankedTables to ErdExecutor.java
```

DrawERD.java (Figure 38) will take the information and create a graphical representation.



```java
public class DrawERD {
    private String erd = "";
    HashMap<String, List<String>> mTableMetadata;
    public String draw(String[] rankOrder, HashMap<String, List<String>> tableMetadata, HashMap<String,
            HashMap<String, String[]>> dependencyGraph) {
        erd = "";
        mTableMetadata = tableMetadata;
        for (int i = 0; i < rankOrder.length; i++) {
            if (rankOrder[i] == null)
                continue;
            erd += "-".repeat(35) + "\n";
            erd += "| " + rankOrder[i].toUpperCase() + " ".repeat(31 - rankOrder[i].length()) + " |\n";
            erd += "-".repeat(35) + "\n";
            readMetadata(rankOrder[i]);
            erd += "\n";
            if (i < rankOrder.length - 1 && rankOrder[i + 1] != null) {
                String[] relationship = dependencyGraph.get(rankOrder[i + 1]).get(rankOrder[i]);
                if (relationship != null) {
                    erd += (" ".repeat(17) + "|\n").repeat(2);
                    erd += " ".repeat(16) + relationship[2] + "\n";
                    erd += (" ".repeat(17) + "|\n").repeat(2);
                } else {
                    erd += "\n";
                }
            }
        }
        return erd;
    }
```

Figure 38: *DrawERD code snippet*

4. Process Flow

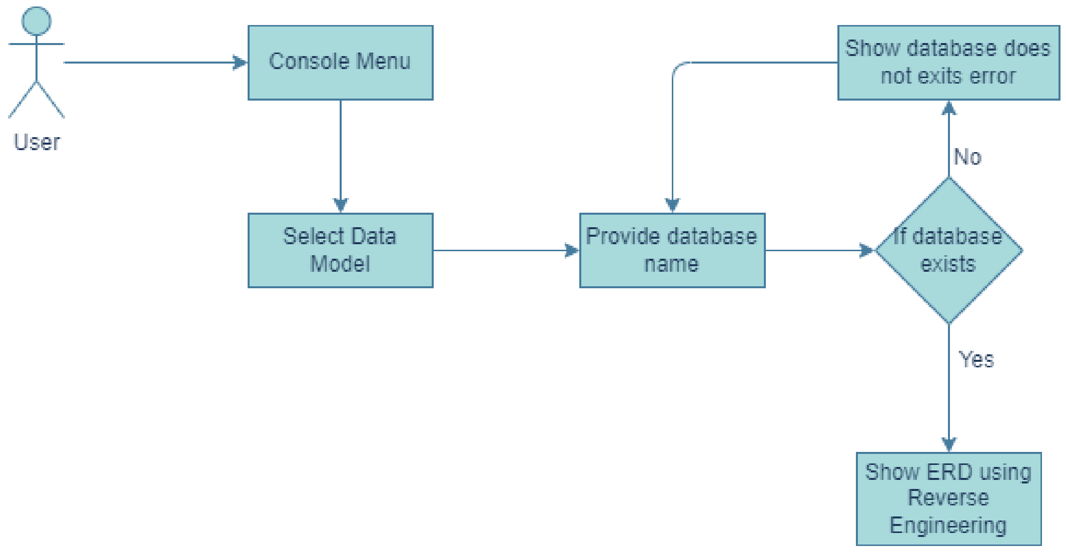
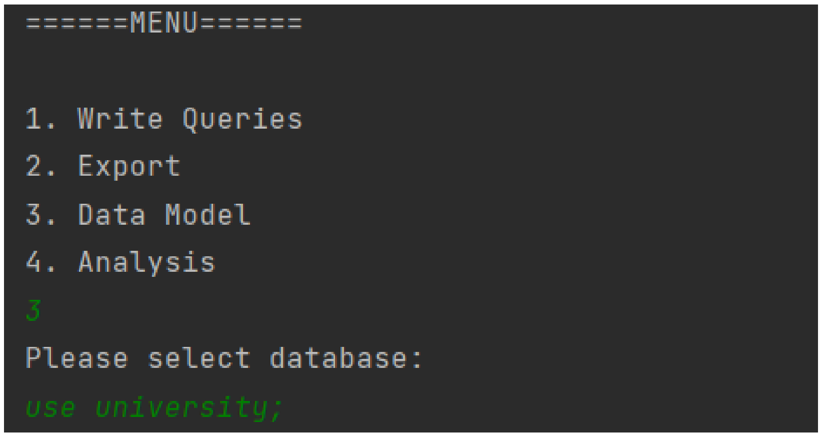


Figure 39: *Flowchart of using Data Model function from console*

Figure 39 shows flowchart to use the Data Model feature from the console to create an ERD for a given database using reverse engineering.

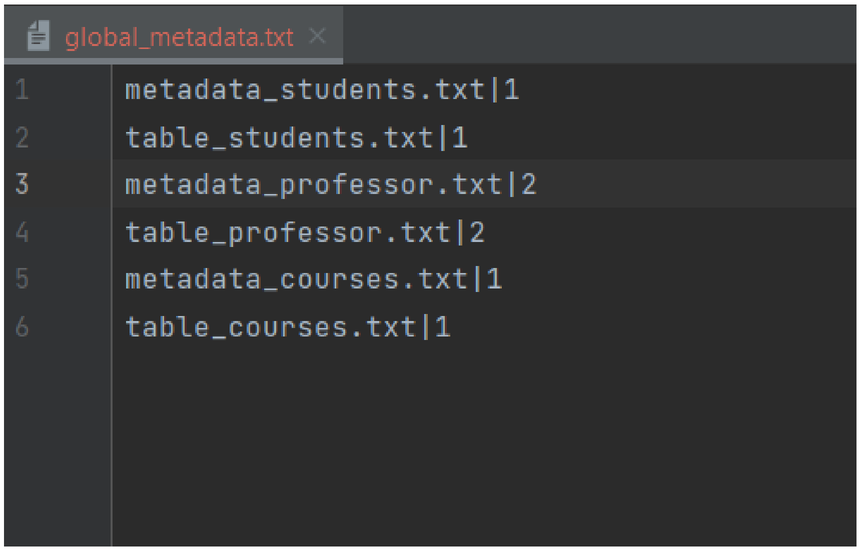
5. Example

A user can use the console to select Data Model option to create the ERD for a given database (Figure 40).
Figure 41 shows the Global Metadata of University database with Figure 42, 43, 44 having Students, Courses and Professors metadata respectively.
Figure 45 is the ERD diagram generated by reverse engineering for the given tables.



```
======MENU======

1. Write Queries
2. Export
3. Data Model
4. Analysis
3
Please select database:
use university;
```

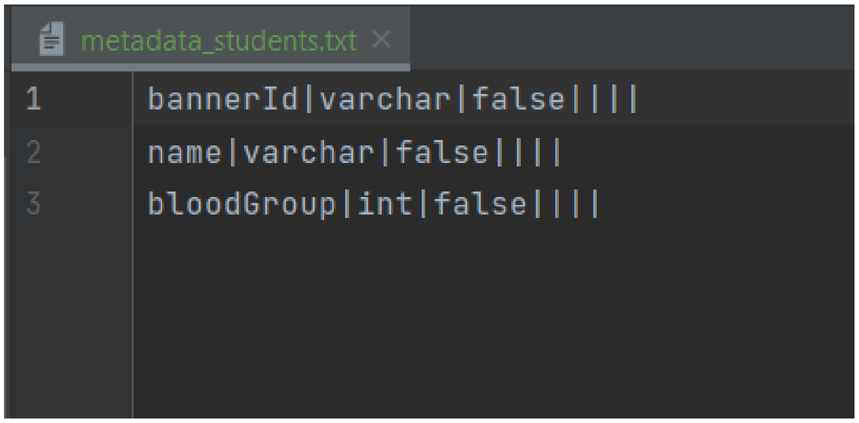
Figure 40: *Console menu for selecting Data Model*



global_metadata.txt

```
metadata_students.txt|1
table_students.txt|1
metadata_professor.txt|2
table_professor.txt|2
metadata_courses.txt|1
table_courses.txt|1
```

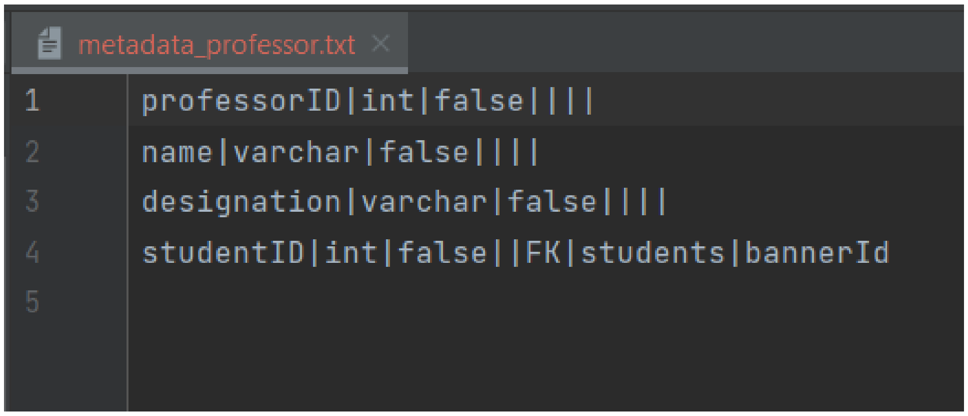
Figure 41: *Global metadata for tables, metadata and instance detail*



metadata_students.txt

```
bannerId|varchar|false|||
name|varchar|false|||
bloodGroup|int|false|||
```

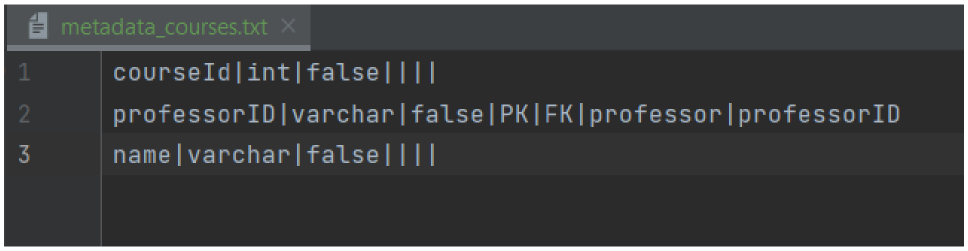
Figure 42: *Students table local metadata*



metadata_professor.txt

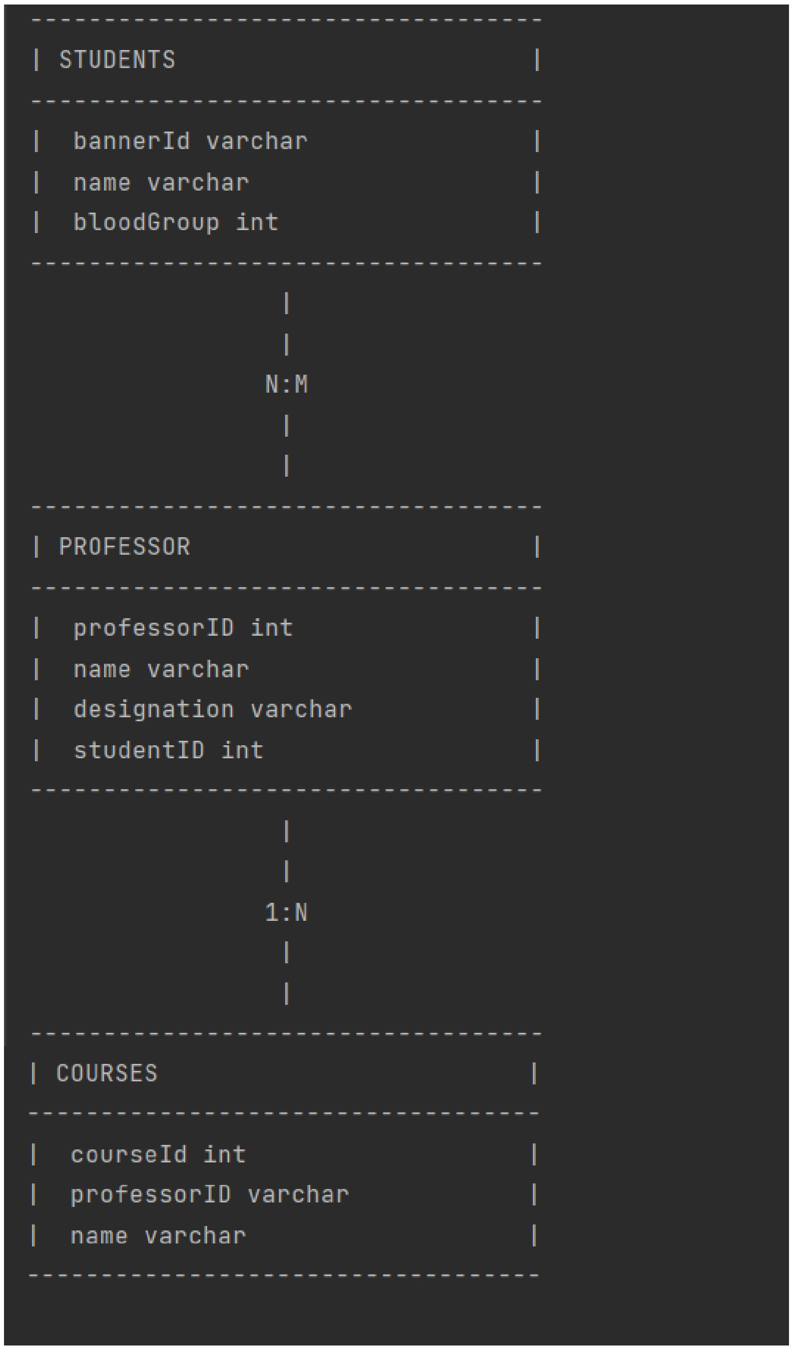
```
professorID|int|false|||
name|varchar|false|||
designation|varchar|false|||
studentID|int|false||FK|students|bannerId
```

Figure 43: *Professor table local metadata*



```
courseId|int|false||||
professorID|varchar|false|PK|FK|professor|professorID
name|varchar|false||||
```

Figure 44: *Courses table local metadata*



```
-----------------------------------
| STUDENTS                        |
-----------------------------------
|  bannerId varchar               |
|  name varchar                   |
|  bloodGroup int                 |
-----------------------------------
                 |
                 |
                N:M
                 |
                 |
-----------------------------------
| PROFESSOR                       |
-----------------------------------
|  professorID int                |
|  name varchar                   |
|  designation varchar            |
|  studentID int                  |
-----------------------------------
                 |
                 |
                1:N
                 |
                 |
-----------------------------------
| COURSES                         |
-----------------------------------
|  courseId int                   |
|  professorID varchar            |
|  name varchar                   |
-----------------------------------
```

Figure 45: *ERD for Students, Professor and Courses table*

14. Login-Signup Module

Basically, this part of the code is responsible for authenticating the user, that if the user is already registered or not. The upcoming part will explain the core algorithm with the help of code snippets and a high-level ER diagram.

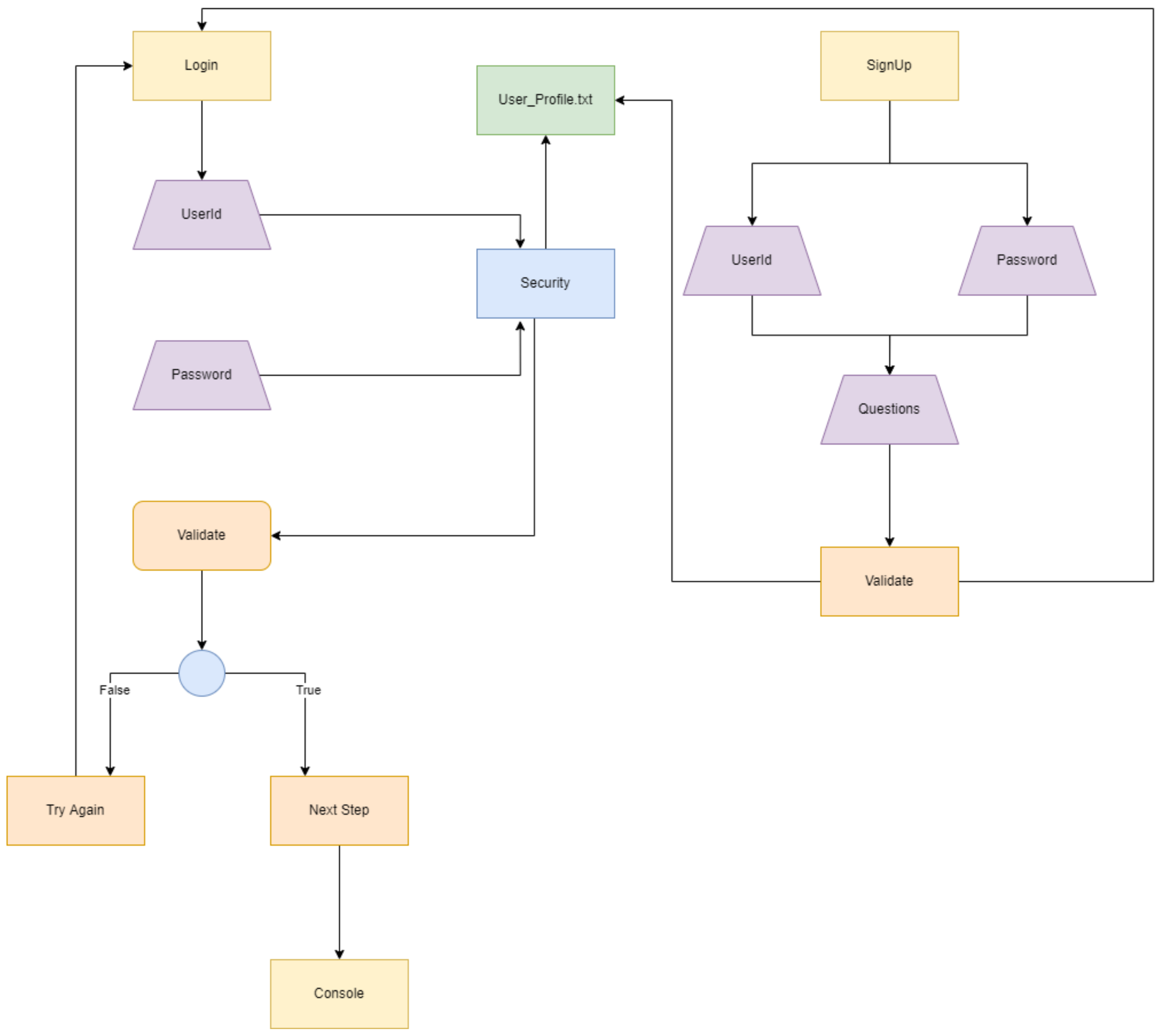


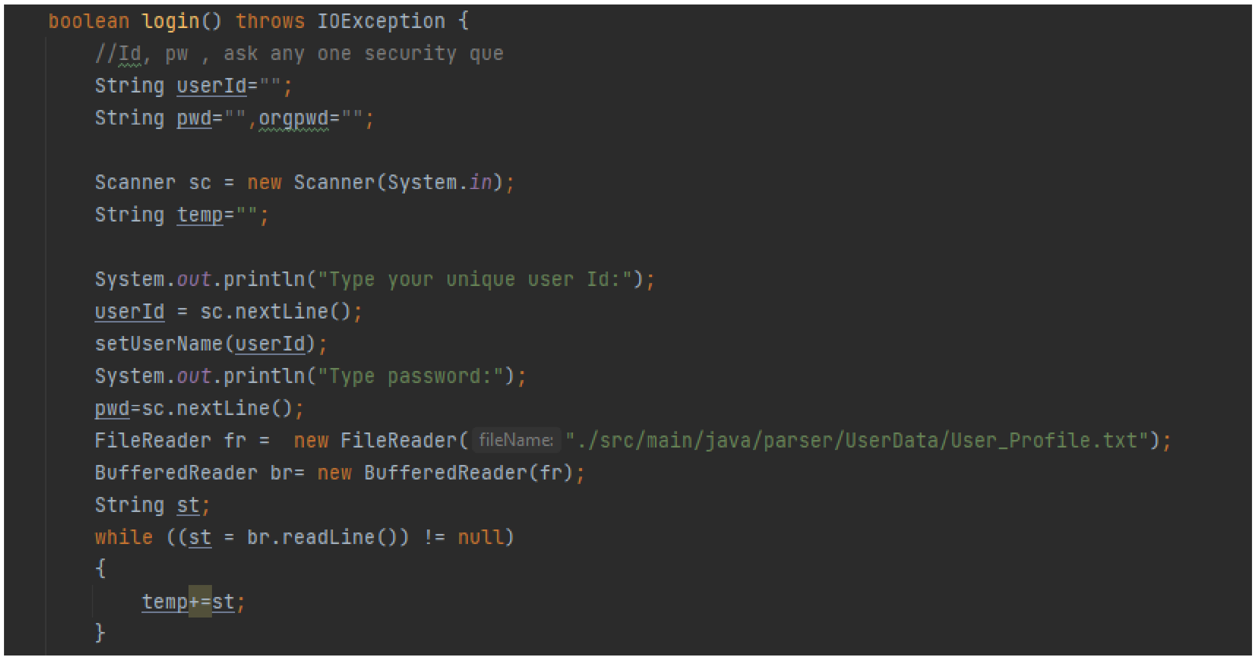
Figure 46: *Login-Signup module*

As we can see in the above screenshot, the program first gives the user an option to select that is the user has already been registered and directly want to sign up or register himself.

According to the selection, the code will take the user to the Login or signup page. Suppose the user wants to register himself, then firstly the program will ask for the unique User Id as well as password, after that the program will ask for a set of questions. In the next step after validating, the program will encrypt the password string with SHA-1, of size 32 byte, and store it into the User_Profile.txt as seen in the above screenshot. After doing these all the users will be redirected to the login page, where they can log in using the recently created UserId, password, and any given randomized question.

In the Login part, first, the program will ask the user for inputting the login UserId and the password and answer any randomized question. Then according to the answer, the program will validate the user. It'll decrypt the password in the database of the user and match it with the given details provided by the user, if they will match then the user will be redirected to the console page where he can select the options, and if not then again the user needs to log in as seen in the ER diagram.

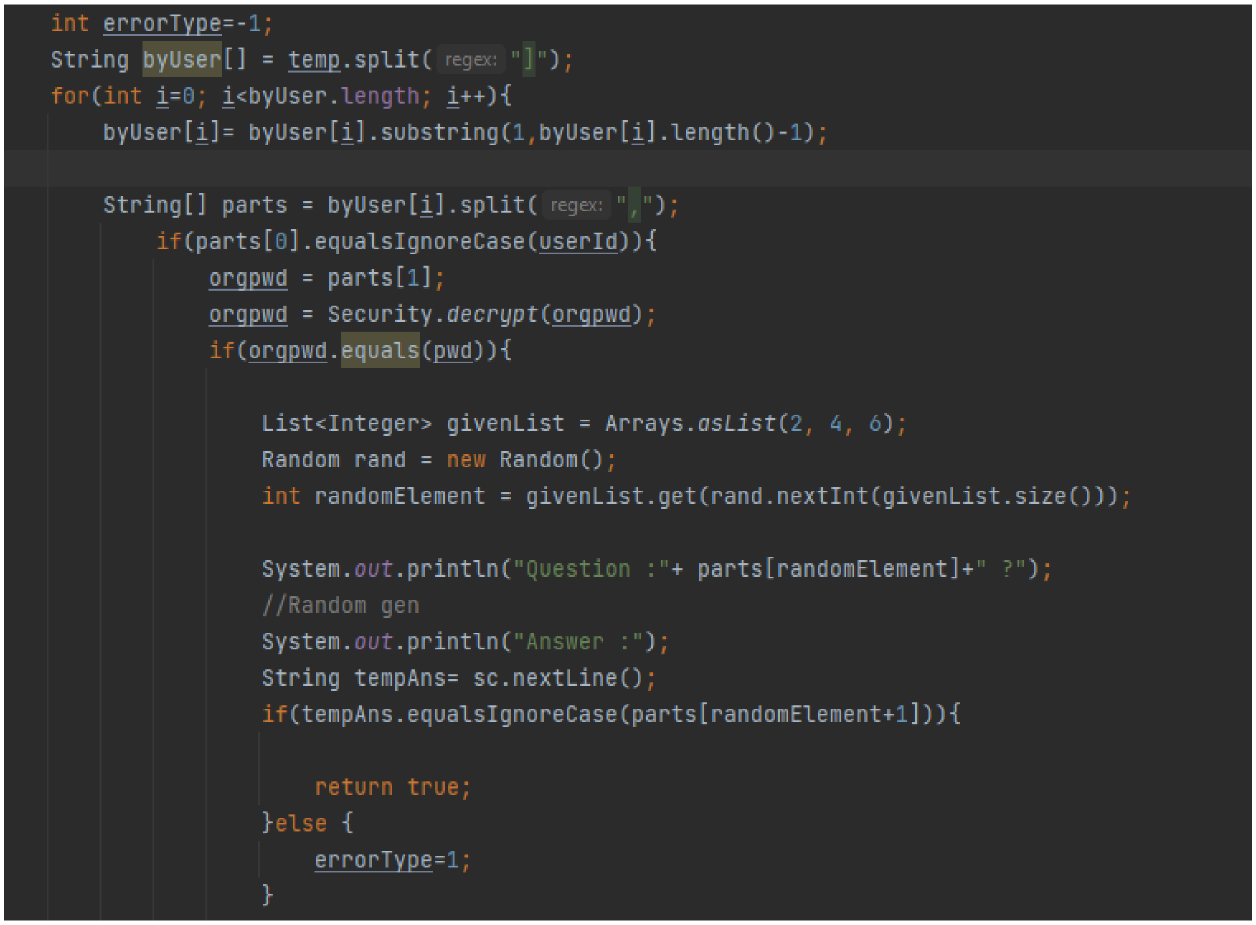
Now let's deep dive into the detailed section of the code:



```
boolean login() throws IOException {
    //Id, pw , ask any one security que
    String userId="";
    String pwd="",orgpwd="";

    Scanner sc = new Scanner(System.in);
    String temp="";

    System.out.println("Type your unique user Id:");
    userId = sc.nextLine();
    setUserName(userId);
    System.out.println("Type password:");
    pwd=sc.nextLine();
    FileReader fr =  new FileReader( fileName: "./src/main/java/parser/UserData/User_Profile.txt");
    BufferedReader br= new BufferedReader(fr);
    String st;
    while ((st = br.readLine()) != null)
    {
        temp+=st;
    }
```

Figure 47: *Login code snippet*

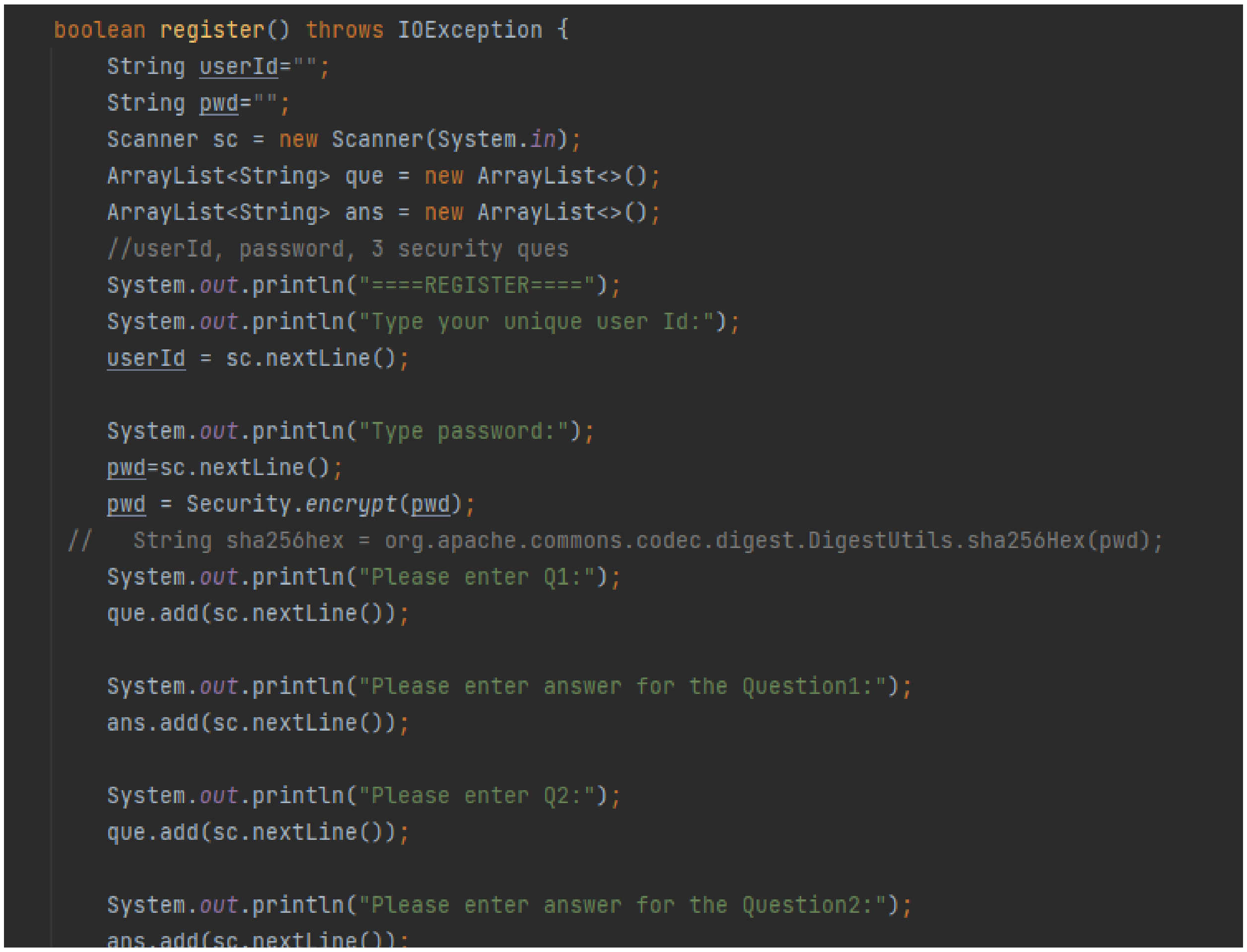
This is the login() logic of the code, the function has been triggered every time user pressed 1 from the login-signup screen. In the code, first it asks for the user Id adn the password then it retrieved the appropriate password from the User_Profile.txt file, decryptys the password and then matches it with the user given password. If both the password matches then it moves to the console screen else it redirects again to the login page of the module.



```
int errorType=-1;
String byUser[] = temp.split( regex: "]");
for(int i=0; i<byUser.length; i++){
    byUser[i]= byUser[i].substring(1,byUser[i].length()-1);

    String[] parts = byUser[i].split( regex: ",");
        if(parts[0].equalsIgnoreCase(userId)){
            orgpwd = parts[1];
            orgpwd = Security.decrypt(orgpwd);
            if(orgpwd.equals(pwd)){

                List<Integer> givenList = Arrays.asList(2, 4, 6);
                Random rand = new Random();
                int randomElement = givenList.get(rand.nextInt(givenList.size()));

                System.out.println("Question :"+ parts[randomElement]+" ?");
                //Random gen
                System.out.println("Answer :");
                String tempAns= sc.nextLine();
                if(tempAns.equalsIgnoreCase(parts[randomElement+1])){

                    return true;
                }else {
                    errorType=1;
                }
```

Figure 48*: Errors, authentication code snippet*

As we can see, if any error occurs, then the code will store the appropriate errorType and displays to the user that because of which reason, the code is been breaked. I.e. wrong username, wrong password, wrong anwer to the question, etc. Because of this, the user will know the exact reason and check is he/she made any mistake.



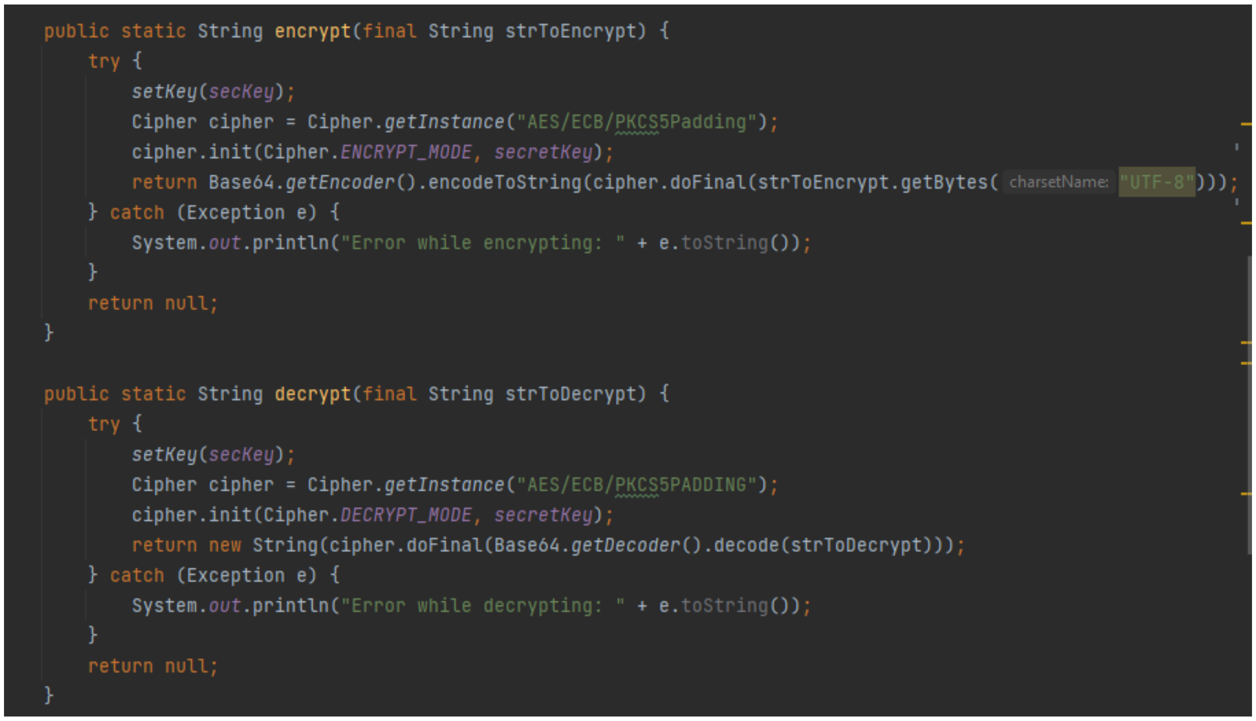
```
boolean register() throws IOException {
    String userId="";
    String pwd="";
    Scanner sc = new Scanner(System.in);
    ArrayList<String> que = new ArrayList<>();
    ArrayList<String> ans = new ArrayList<>();
    //userId, password, 3 security ques
    System.out.println("====REGISTER====");
    System.out.println("Type your unique user Id:");
    userId = sc.nextLine();

    System.out.println("Type password:");
    pwd=sc.nextLine();
    pwd = Security.encrypt(pwd);
//    String sha256hex = org.apache.commons.codec.digest.DigestUtils.sha256Hex(pwd);
    System.out.println("Please enter Q1:");
    que.add(sc.nextLine());

    System.out.println("Please enter answer for the Question1:");
    ans.add(sc.nextLine());

    System.out.println("Please enter Q2:");
    que.add(sc.nextLine());

    System.out.println("Please enter answer for the Question2:");
    ans.add(sc.nextLine());
```

Figure 49: *Register code snippet*

Now this is the register logic of the code. As seen in the screenshot 4 first the code takes all the inputs from the user then it encrypts the password and stores the encrypted password into the User_Profile.txt file. By doing this the program provides the level of security to the code.

After registering, as the logic says, if the registration is successful, then the page will redirect the user to the login page again. By doing this, again the level of security is been achieved.

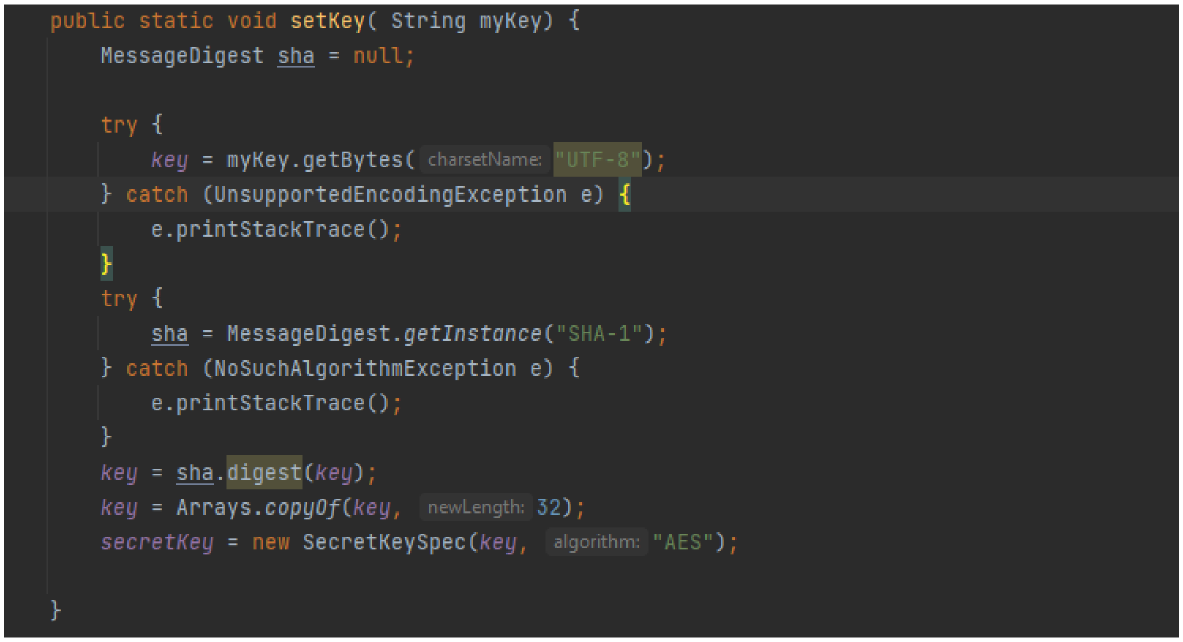
Now, lets understand, how the code is doing encryption and decryption to the password string:



```java
public static String encrypt(final String strToEncrypt) {
    try {
        setKey(secKey);
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        cipher.init(Cipher.ENCRYPT_MODE, secretKey);
        return Base64.getEncoder().encodeToString(cipher.doFinal(strToEncrypt.getBytes( charsetName: "UTF-8")));
    } catch (Exception e) {
        System.out.println("Error while encrypting: " + e.toString());
    }
    return null;
}

public static String decrypt(final String strToDecrypt) {
    try {
        setKey(secKey);
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5PADDING");
        cipher.init(Cipher.DECRYPT_MODE, secretKey);
        return new String(cipher.doFinal(Base64.getDecoder().decode(strToDecrypt)));
    } catch (Exception e) {
        System.out.println("Error while decrypting: " + e.toString());
    }
    return null;
}
```

Figure 50: *Encrypt/decrypt code snippet*

There are two functions called encrypt and decrypt inside the security class file. Basically, both the functions take strings as input and return encrypted/decrypted string as an output. In the encrypt function, it takes a normal string as an input and then as shown below goes into the code and settes the key.



```java
public static void setKey( String myKey) {
    MessageDigest sha = null;

    try {
        key = myKey.getBytes( charsetName: "UTF-8");
    } catch (UnsupportedEncodingException e) {
        e.printStackTrace();
    }
    try {
        sha = MessageDigest.getInstance("SHA-1");
    } catch (NoSuchAlgorithmException e) {
        e.printStackTrace();
    }
    key = sha.digest(key);
    key = Arrays.copyOf(key,  newLength: 32);
    secretKey = new SecretKeySpec(key,  algorithm: "AES");

}
```

Figure 51: *Setting the key in cypher code snippet*

As we can see here this code first settes the message digest to the sha-1 algorithm, and the keyset to the UTF-8. In the last step the code settes the size of the key to 32, which is even more secure than the 16. The downside is that it takes more space, but as our program is only going to have some number of users, the space isn't a big issue for us.

In the decryption part of the code, the function basically takes the encrypted text as an input and by using the cypher.init function it will decrypt the code to it's original form, from here we can directly match the decrypted string to the original string and if they'll match then we can conclude that the user is been authenticated.

Now let's move to the next module which is user console, from where the user can select the number of inputs to do the appropriate:

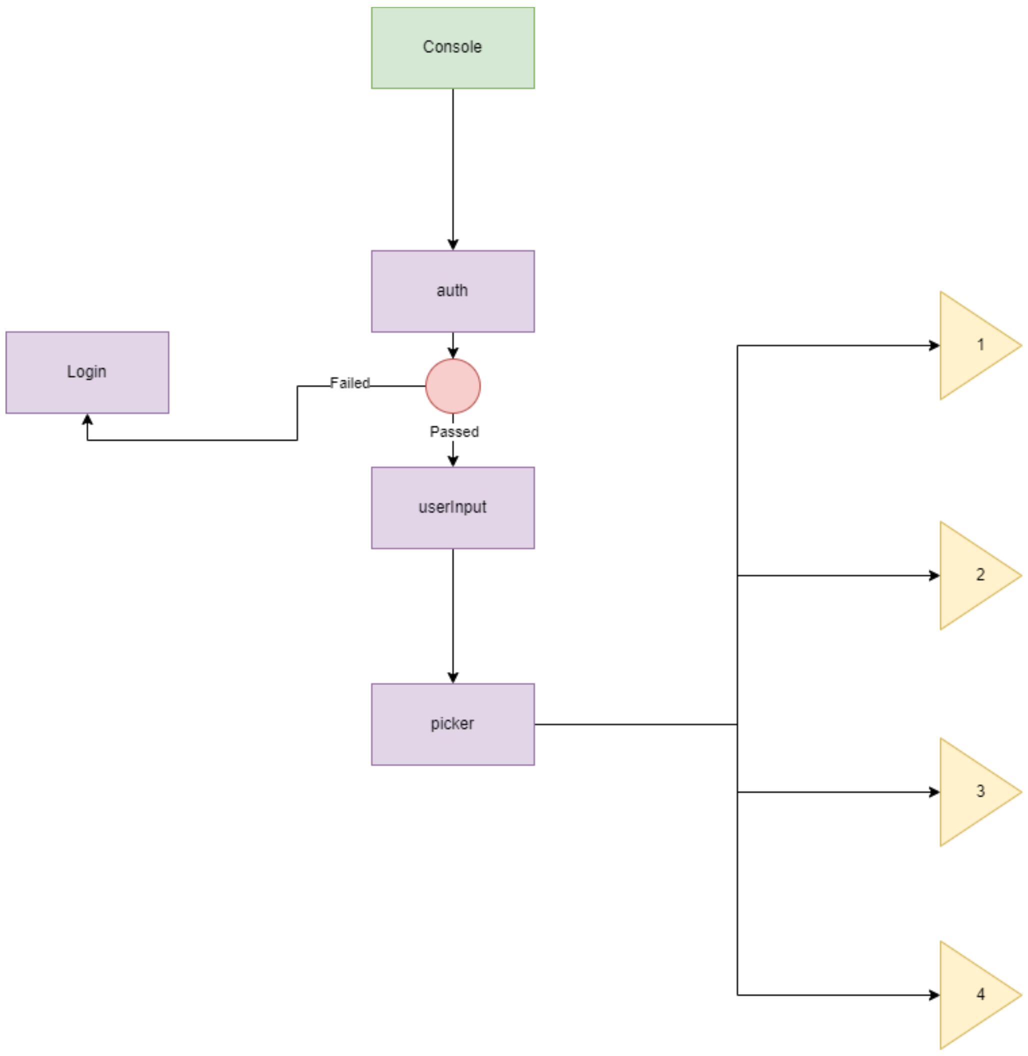
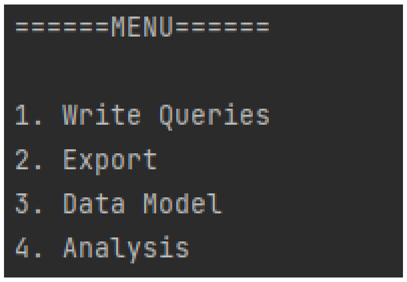


Figure 52: *Console ERD*

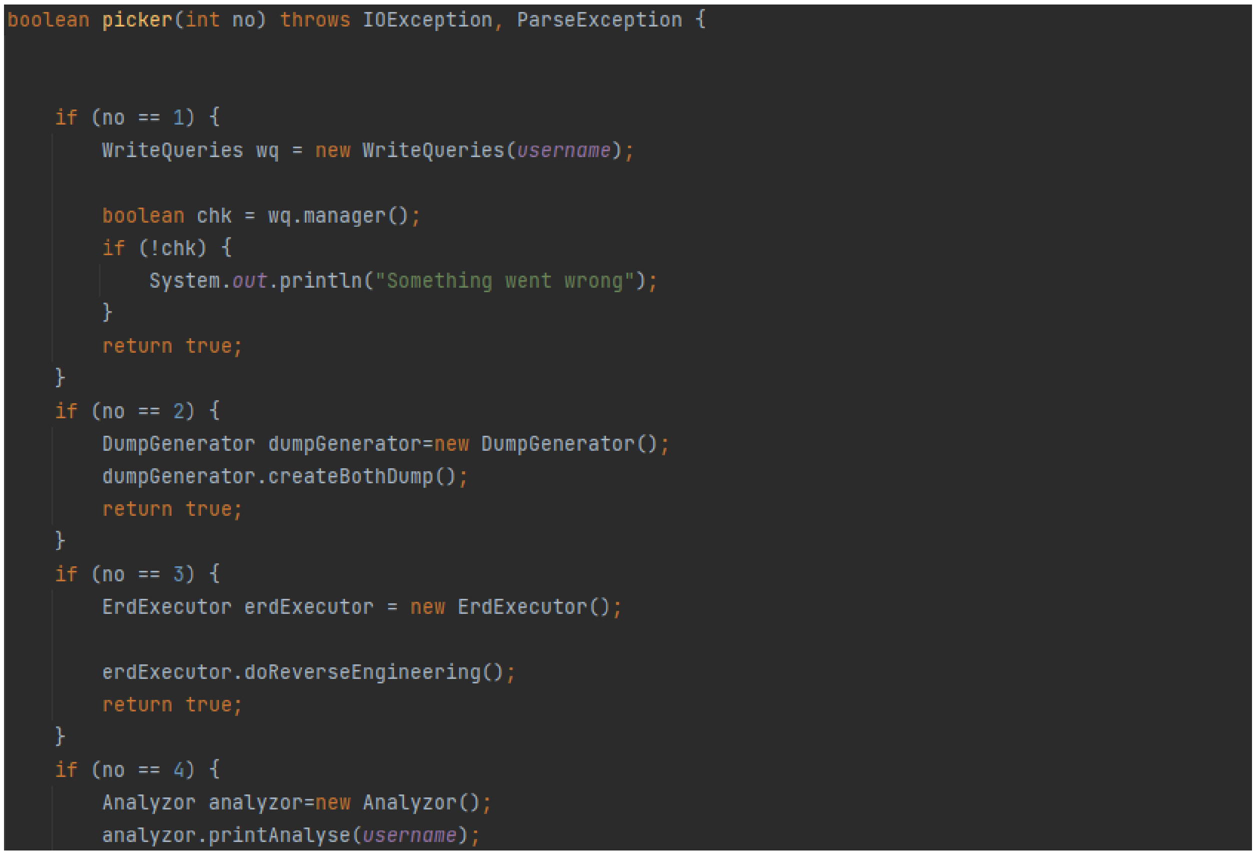
As we can see from the screenshot-6 and screenshot-1 the function moves to the console page, here first to provide security, it checks if the user is authintacated or not. If the user is then it'll show totally 4 options as shown below. They are a part of userInput() function.



```
======MENU======

1. Write Queries
2. Export
3. Data Model
4. Analysis
```

***Figure 53**: Console output*

For the actual selection the code is using the picker function. It takes the number as a input and redirects the page to appropriate place as we can see in the screenshot-6. The code example is given below:



```java
boolean picker(int no) throws IOException, ParseException {


    if (no == 1) {
        WriteQueries wq = new WriteQueries(username);

        boolean chk = wq.manager();
        if (!chk) {
            System.out.println("Something went wrong");
        }
        return true;
    }
    if (no == 2) {
        DumpGenerator dumpGenerator=new DumpGenerator();
        dumpGenerator.createBothDump();
        return true;
    }
    if (no == 3) {
        ErdExecutor erdExecutor = new ErdExecutor();

        erdExecutor.doReverseEngineering();
        return true;
    }
    if (no == 4) {
        Analyzor analyzor=new Analyzor();
        analyzor.printAnalyse(username);
```

***Figure 54:** Choosing the appropriate function for console input*

As we can see, according to the number chosen it'll redirect the user to the page. On number 1 it'll go to the WriteQueries, on selecting number 2, it'll redirect to the Dump generator, which will basically generates the dataDump for the data. On selecting number 3 it'll move to the ERD phase where, it'll create the ERDiagram for the code. And by selecting number 4 it'll move to the Analyse part, where it'll show all the logs and everything that has ever done according to the instances to the code, and by which user as well.

Now if the user selects 1 it'll move to the WriteQueries function, let's understand that in detail.

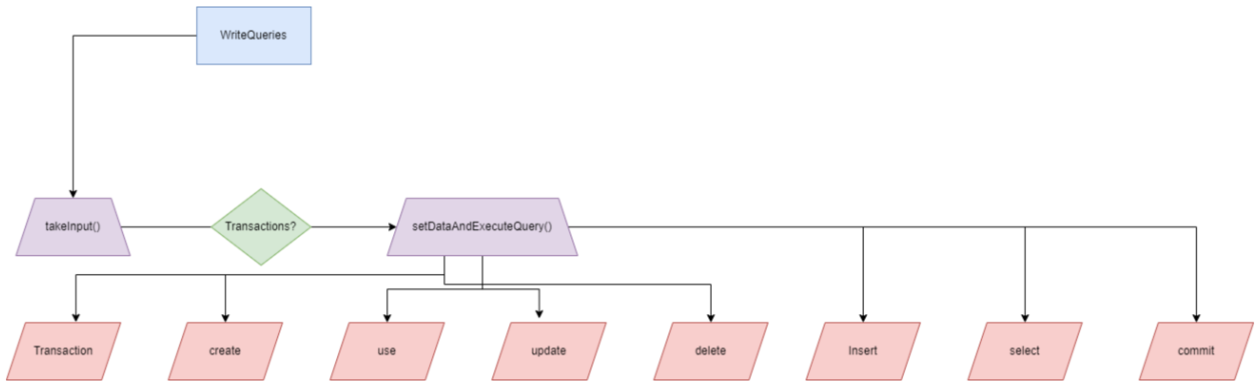
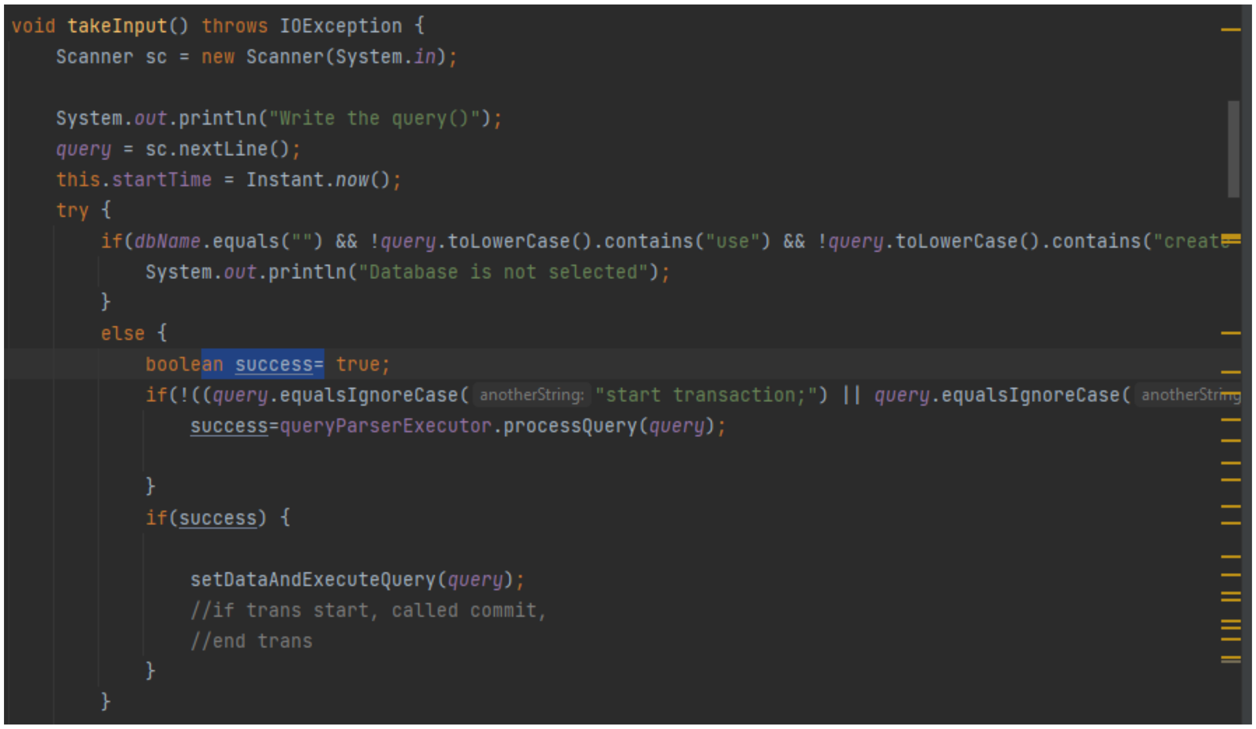


***Figure 55**: WriteQueries Flowchart*

As we can see from the high level architecture, first the WriteQuery constructor, takes the username as an input, as it's been used at number of places in the code.

The first function been called is takeInput(). The function first asks for the query string, then if the query is a take input or commit type then directly redirects it to the setDataAndExecuteQuery function, but if not the first it sends the query to processQuery() function, where basically it checks is the query is valid or not and if it is then it sents the required variables for the later use like the tablename, dbName, etc.
This function is been made to save some computation, if the query is wrong then there is no need to process it further and discard it in the first place.

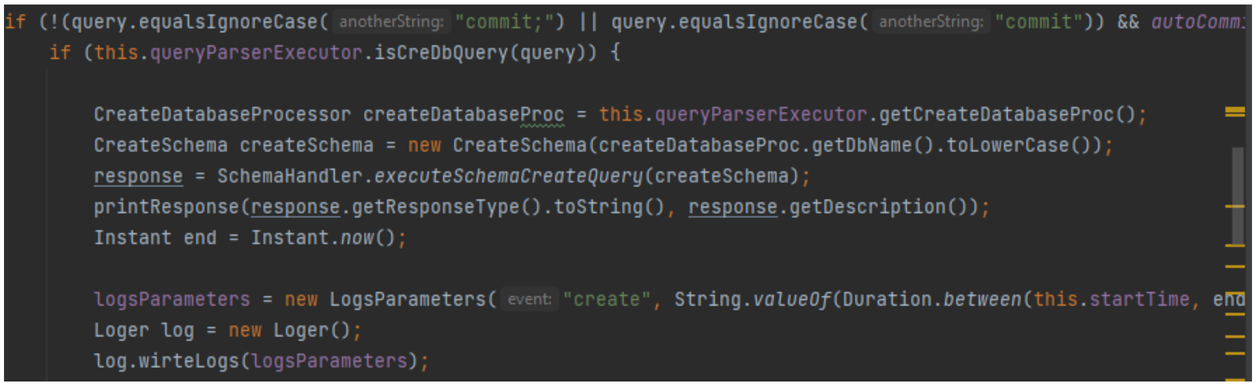
After the successful parsing the next step is as shown below:



```
void takeInput() throws IOException {
    Scanner sc = new Scanner(System.in);

    System.out.println("Write the query()");
    query = sc.nextLine();
    this.startTime = Instant.now();
    try {
        if(dbName.equals("") && !query.toLowerCase().contains("use") && !query.toLowerCase().contains("creat
            System.out.println("Database is not selected");
        }
        else {
            boolean success= true;
            if(!((query.equalsIgnoreCase( anotherString: "start transaction;") || query.equalsIgnoreCase( anotherStr
                success=queryParserExecutor.processQuery(query);

            }
            if(success) {

                setDataAndExecuteQuery(query);
                //if trans start, called commit,
                //end trans
            }
        }
```

Figure 56: *Code snippet for takeInput*

As we can see, the query ignores the step is it’s related to he transactions, as there’s no need to check the query, when we can directly match the string. And for the later part, if the query is correct then the success will biome true else false, if it’s true then it’ll be moved to the setDataandExecureQuery(String) part, which basically executes the query by using set of method and if else blocks. The transaction is been taken care of there as well.

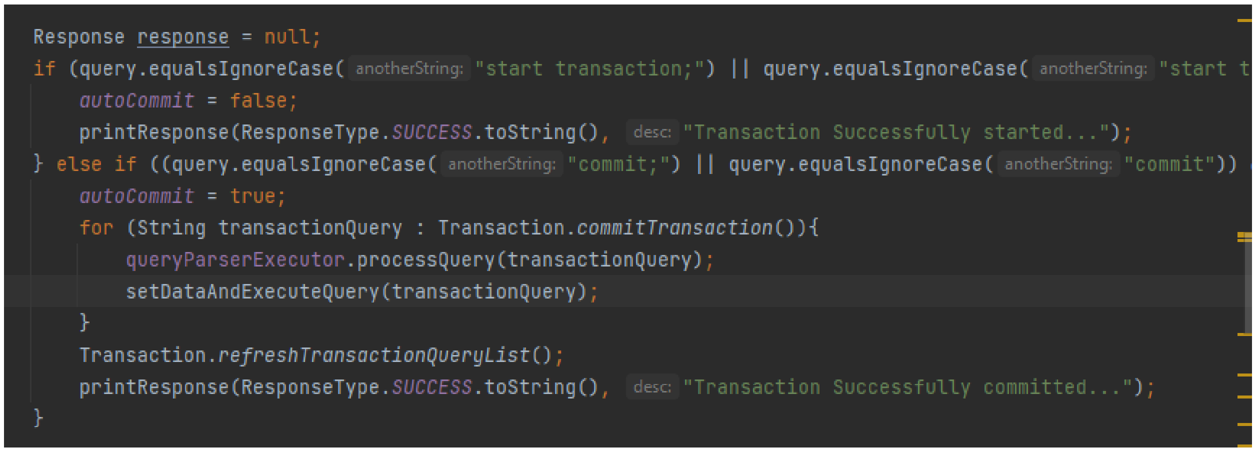


```
if (!(query.equalsIgnoreCase( anotherString: "commit;") || query.equalsIgnoreCase( anotherString: "commit")) && autoComm
    if (this.queryParserExecutor.isCreDbQuery(query)) {

        CreateDatabaseProcessor createDatabaseProc = this.queryParserExecutor.getCreateDatabaseProc();
        CreateSchema createSchema = new CreateSchema(createDatabaseProc.getDbName().toLowerCase());
        response = SchemaHandler.executeSchemaCreateQuery(createSchema);
        printResponse(response.getResponseType().toString(), response.getDescription());
        Instant end = Instant.now();

        logsParameters = new LogsParameters( event: "create", String.valueOf(Duration.between(this.startTime, end
        Loger log = new Loger();
        log.wirteLogs(logsParameters);
```

Figure 57*: Queries if else code snippet*

In the setDataAndExecuteQuery function, is made from number of if else blocks like in screenshot-11. It basically checks for the query type then, the response has all the data fields setted up and at the last step sets all the login parameters to the loginParameters object and for the every type of query there is a CreateQuery type of functions, like SelectQuery, UpdateQuery, etc. where the query is been processed and handles, and do the appropriate to the query. In the last step, it’ll return the response, which gives the status that if the query is been passed or not.

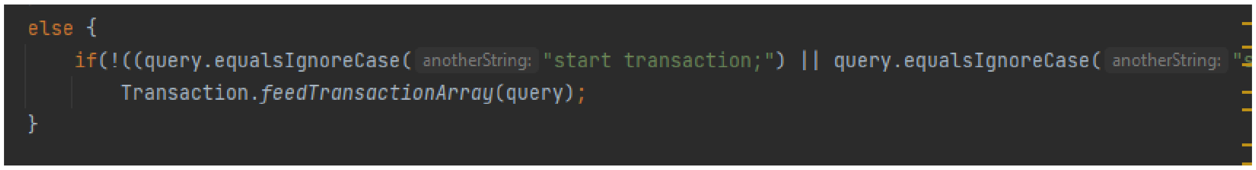
Meanwhile, in every step, the function is storing the step into the logs, and it gives the appropriate data values to the function as well. The class has been called is login parameters. The logs will be written by calling the log. writeLogs function by taking the loginParameters as an input.



```
Response response = null;
if (query.equalsIgnoreCase( anotherString: "start transaction;") || query.equalsIgnoreCase( anotherString: "start t
    autoCommit = false;
    printResponse(ResponseType.SUCCESS.toString(), desc: "Transaction Successfully started...");
} else if ((query.equalsIgnoreCase( anotherString: "commit;") || query.equalsIgnoreCase( anotherString: "commit"))
    autoCommit = true;
    for (String transactionQuery : Transaction.commitTransaction()){
        queryParserExecutor.processQuery(transactionQuery);
        setDataAndExecuteQuery(transactionQuery);
    }
    Transaction.refreshTransactionQueryList();
    printResponse(ResponseType.SUCCESS.toString(), desc: "Transaction Successfully committed...");
}
```

Figure 58: *Transactions code snippet*

To manage the transactions, as shown in the code block, it first checks the type of query, is its start transaction or commit type then, it won't go to the other normal query types, first it'll just set the auto-commit to false and it'll go to the else block of the remaining query. Please refer to the full code to get a better idea explained here.



```
else {
    if(!((query.equalsIgnoreCase( anotherString: "start transaction;") || query.equalsIgnoreCase( anotherString: "s
        Transaction.feedTransactionArray(query);
}
```

Figure 59*: Check transactions condition*

In the else block as we can see it basically just stores the query into the arraylist. The function Transaction.feedTransactionArray has a ArrayList built inside. And after calling the commit as we can see in screenshot-12, the function will set the auto-commit to true, and then one by one, by using the for loop as shown in the shot-12 it'll pass the queries to the if-else blocks as shown into the screenshot-11. After executing every block it'll print the response of every query and add the logs as well.

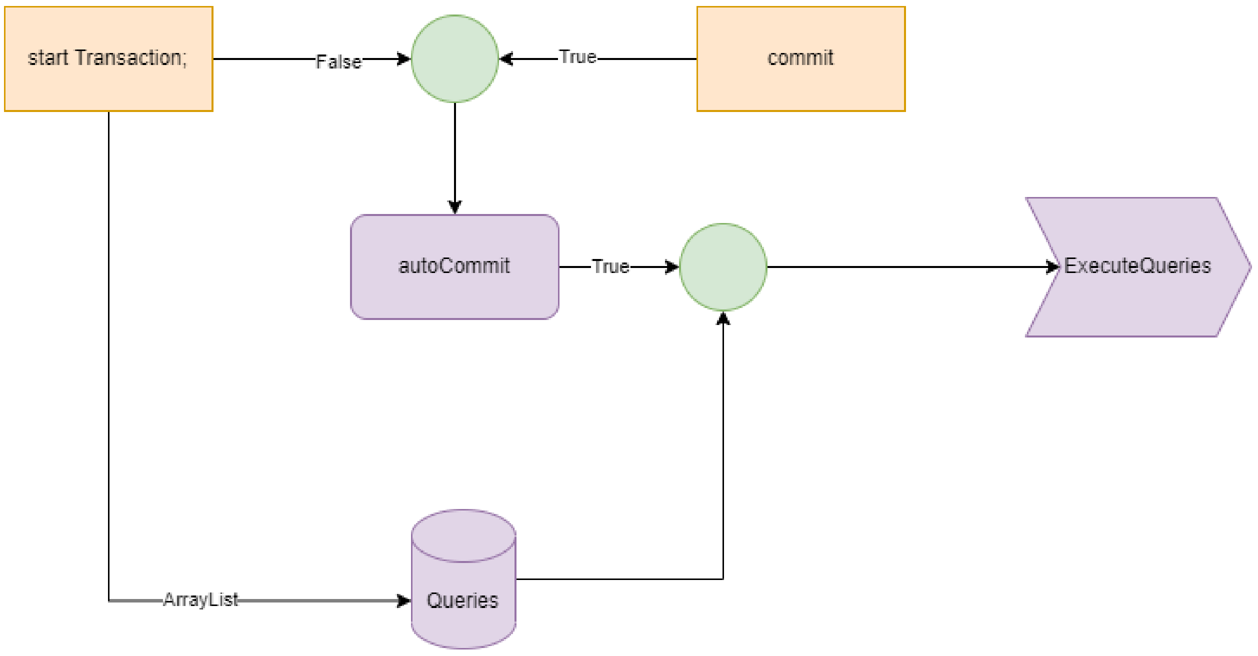


Figure 60: *Transactions Flowchart*

As we can see from the ER diagram, first if the query is started transaction type, then it'll set the autoCommit to the false, and if it's a commit then to the true. Then after the start transaction, every query called has been stored into the ArrayList and after the commit is been called and at the same time if the autoCommit is been called true then if both the conditions been satisfied then the function will be moved to the executeQuery part where each query is been executed as explained in the above part.

15. Query Parser

Use case:

Query parser is a stateless layer of the distributed system. It does not have any idea about the processing logic of another layer of a distributed system. The responsibility of the query parser is to parse the SQL query and extract query data from the SQL statement and send it to the query handler layer. Whenever the user receives SQL query input, it is first validated by the query parser that it is a valid SQL query or not. If it is not a valid SQL statement, then the query parser throws an invalid SQL statement exception. If the given SQL statement is valid then the query parser parses the statement and extracts data such as a table, database name, columns name, column values, and constraints such as foreign key and primary key.

We have implemented a query parser by abiding by the best practices of java development. For each type of SQL statement such as select, create, delete so on, a separate parser is created. Depending on the kind of statement individual parser is called. Then that parser parses the SQL statement and extracts data from the statement and sends extracted data in form of a java object query handler for execution.

Implementation classes for query parser:

1. WriteQueries.java
2. CreateDatabaseProcessor.java
3. CreateQueryProcessor.java
4. DeleteQueryProcessor.java
5. InsertQueryProcessor.java
6. SelectQueryProcessor.java
7. UpdateQueryProcessor.java
8. UseDatabaseQueryProcessor.java
9. QueryParser.java
10. QueryParserExecutor.java

Pseudocode for SQL parser:

```
1. START
2. Receive input from the user console.
3. Check the validity of user input.
   3.1. Throw exception if user input is not valid.
4. Call query parser executor if user input is valid and extract data from the SQL statement
5. Select the associated query processor based on the type of SQL query.
6. Send data to the query executor layer.
7. Get a response after query execution
8. Print response to the console.
9. END
```

Types of SQL statements supported:

- Create table
  - In create query int, varchar and float data types are supported.
  - Create a table with a primary key.
  - Create a table with a foreign key.
- Insert Query
  - Insert query without column names.
  - Insert query with column names.
- Normal query execution and as a transaction.
  - Start transaction
  - Commit.
- Select Query
  - Select query with single column and multiple columns.
  - Select query with * operator.
  - Select query with and without where clause.

    - Select query with <, =, <,>=, <= operator in where clause.
- Update query
    - Update query with and without where clause.
    - Update query with <, =,> operator in where clause.
- Delete Query
    - Delete query with and without where clause.
    - Delete query with <, =,> operator in where clause.
- Use database query.
- Create database query.

SQL statements supported:

Create table:

- CREATE TABLE professor(professorID varchar , name varchar , designation varchar);
- CREATE TABLE courses(courseId int PRIMARY KEY,name varchar,professorID int FOREIGN KEY REFERENCES professor(professorID));
- CREATE TABLE students(courseId int FOREIGN KEY REFERENCES courses(courseId),name varchar,marks int);

Insert query:

- INSERT INTO professor(professorID,name,designation) VALUES (P006204,Saurabh Dey,Professor);
- INSERT INTO students VALUES (CSCI5308,vatsal,81);

Insert query:

- INSERT INTO professor(professorID,name,designation) VALUES (P006204,Saurabh Dey,Professor);
- INSERT INTO students VALUES (CSCI5308,vatsal,81);

Process flow:

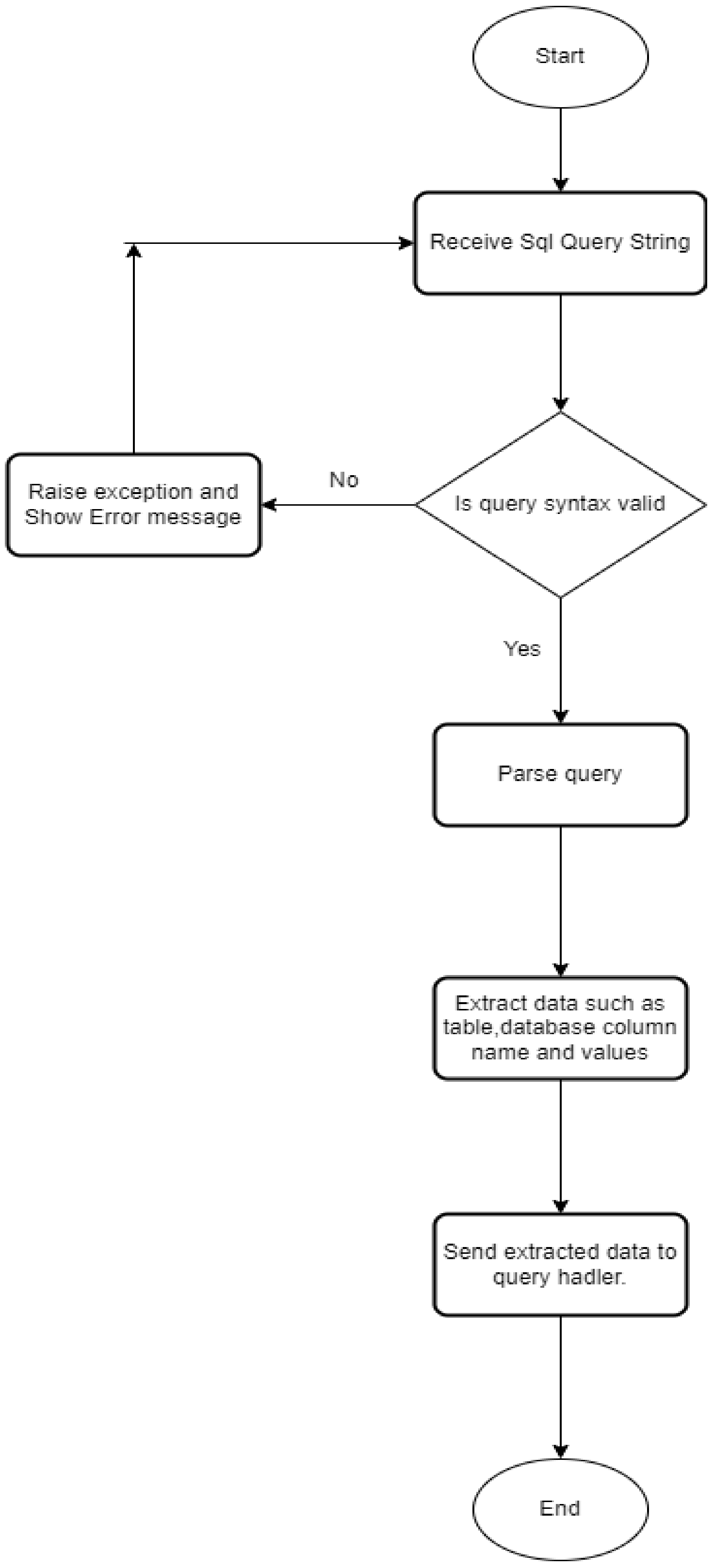
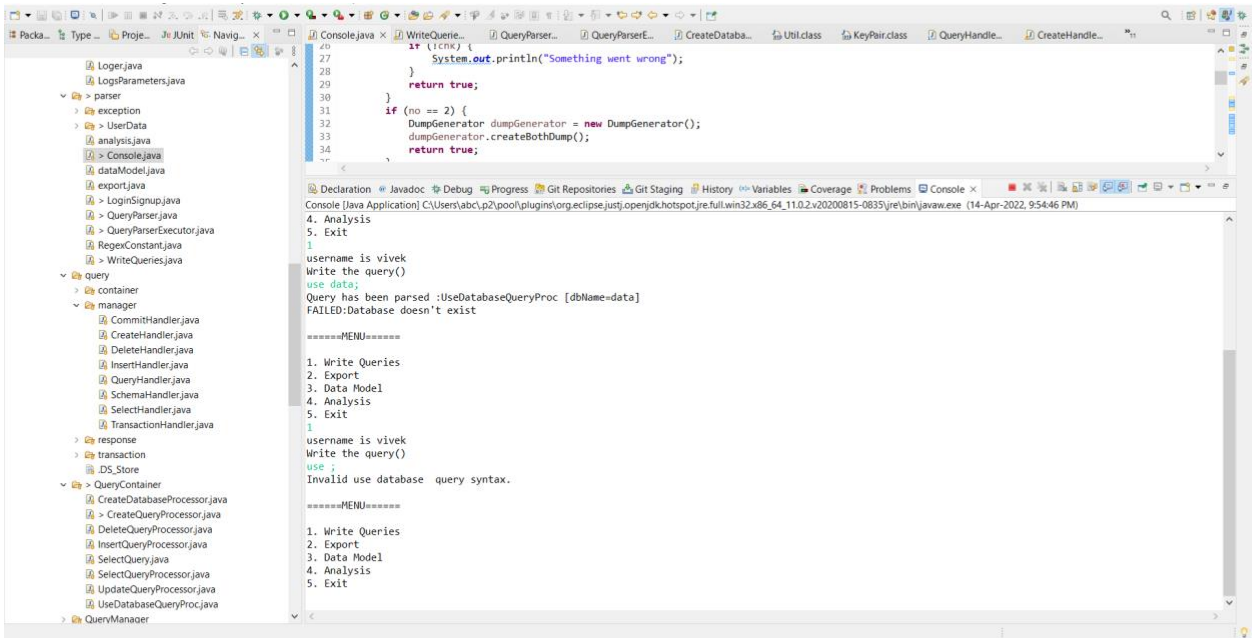

Start
Receive Sql Query String
Is query syntax valid
No
Raise exception and Show Error message
Yes
Parse query
Extract data such as table,database column name and values
Send extracted data to query hadler.
End

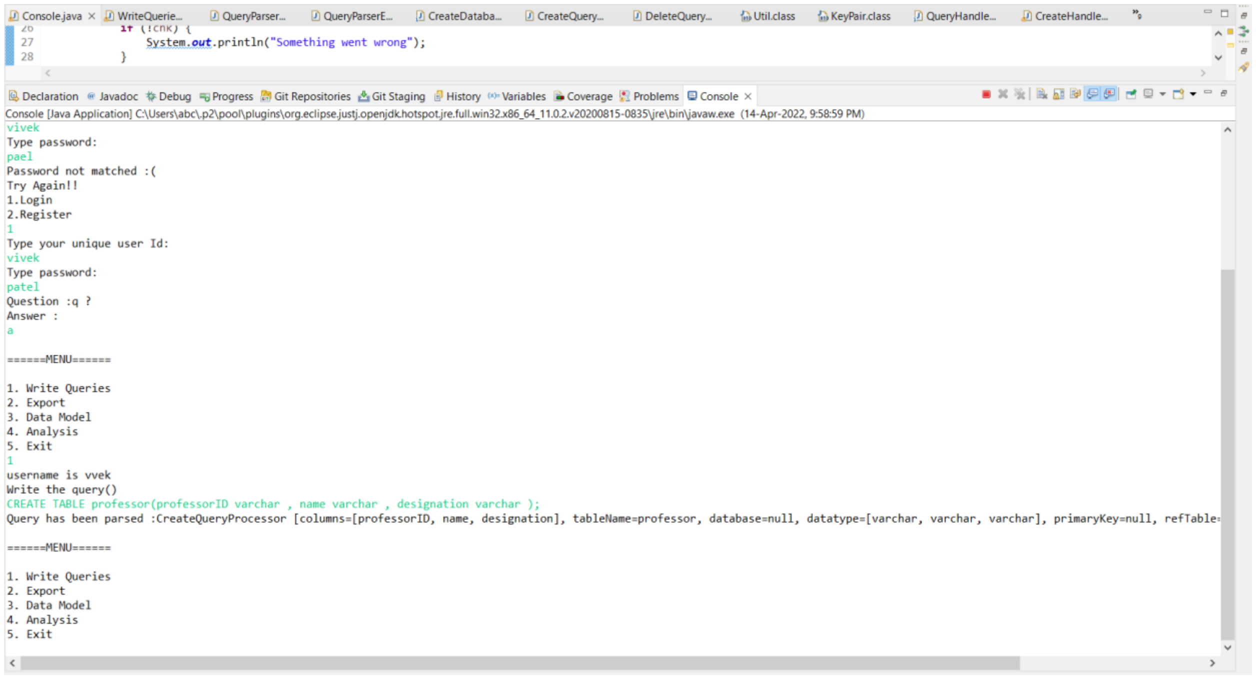
Figure 61: *Flow chart for query parser.*



```
        if (!chk) {
            System.out.println("Something went wrong");
        }
        return true;
    }
    if (no == 2) {
        DumpGenerator dumpGenerator = new DumpGenerator();
        dumpGenerator.createBothDump();
        return true;
```

```
Console [Java Application] C:\Users\abc\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_11.0.2.v20200815-0835\jre\bin\javaw.exe (14-Apr-2022, 9:54:46 PM)
4. Analysis
5. Exit
1
username is vivek
Write the query()
use data;
Query has been parsed :UseDatabaseQueryProc [dbName=data]
FAILED:Database doesn't exist

======MENU======

1. Write Queries
2. Export
3. Data Model
4. Analysis
5. Exit
1
username is vivek
Write the query()
use ;
Invalid use database  query syntax.

======MENU======

1. Write Queries
2. Export
3. Data Model
4. Analysis
5. Exit
```

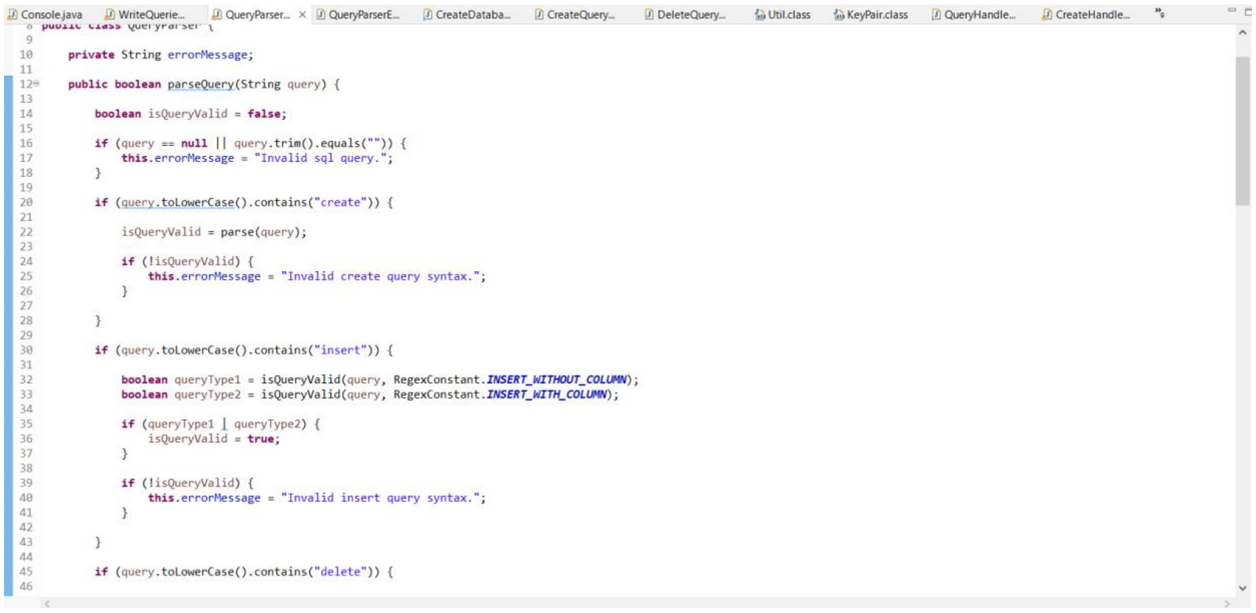
Figure 62: *Invalid use database query.*



```
        if (!chk) {
            System.out.println("Something went wrong");
        }
```

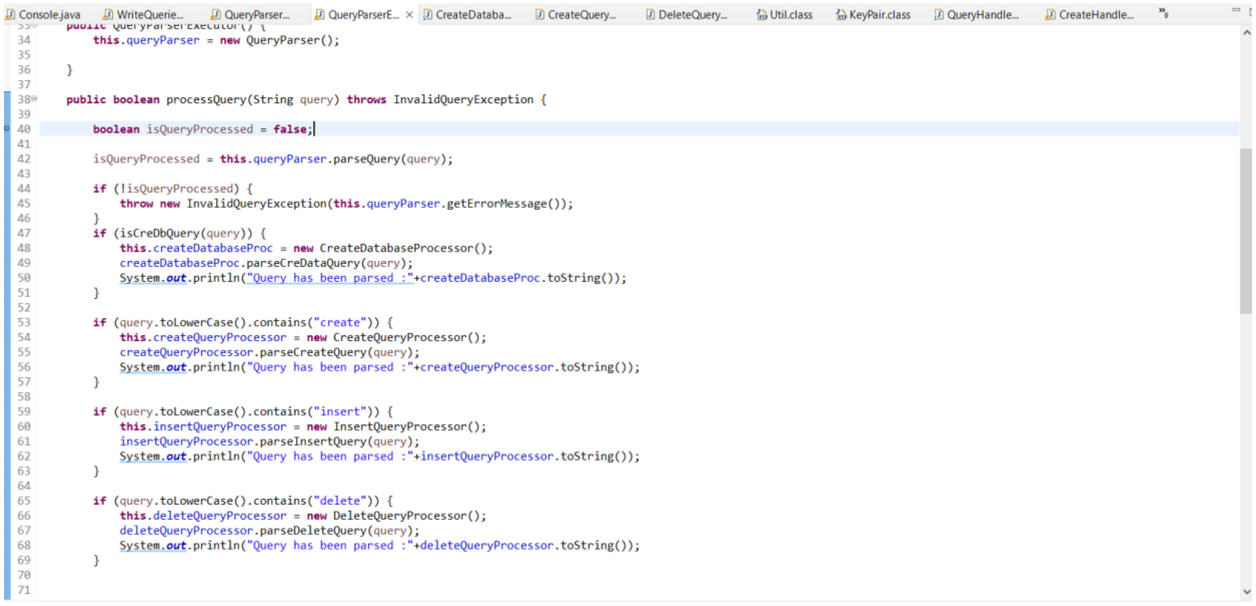
```
Console [Java Application] C:\Users\abc\.p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full.win32.x86_64_11.0.2.v20200815-0835\jre\bin\javaw.exe (14-Apr-2022, 9:58:59 PM)
vivek
Type password:
pael
Password not matched :(
Try Again!!
1.Login
2.Register
1
Type your unique user Id:
vivek
Type password:
patel
Question :q ?
Answer :
a

======MENU======

1. Write Queries
2. Export
3. Data Model
4. Analysis
5. Exit
1
username is vvek
Write the query()
CREATE TABLE professor(professorID varchar , name varchar , designation varchar );
Query has been parsed :CreateQueryProcessor [columns=[professorID, name, designation], tableName=professor, database=null, datatype=[varchar, varchar, varchar], primaryKey=null, refTable=

======MENU======

1. Write Queries
2. Export
3. Data Model
4. Analysis
5. Exit
```

Figure 63: *Valid create table query*



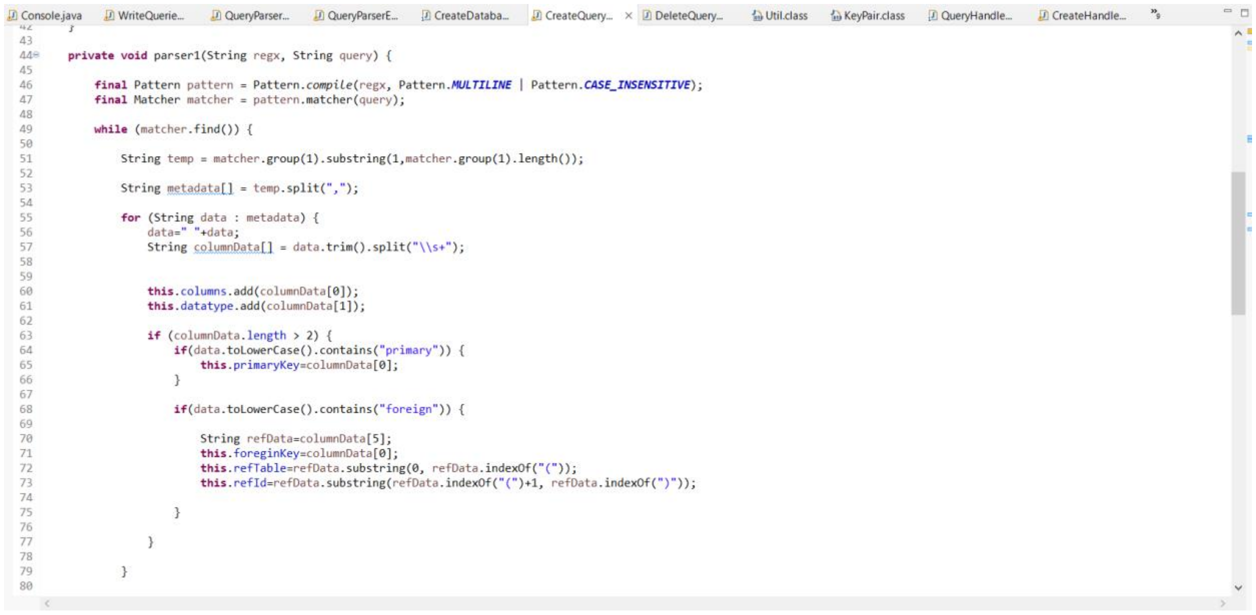
```java
public class QueryParser {

    private String errorMessage;

    public boolean parseQuery(String query) {

        boolean isQueryValid = false;

        if (query == null || query.trim().equals("")) {
            this.errorMessage = "Invalid sql query.";
        }

        if (query.toLowerCase().contains("create")) {

            isQueryValid = parse(query);

            if (!isQueryValid) {
                this.errorMessage = "Invalid create query syntax.";
            }

        }

        if (query.toLowerCase().contains("insert")) {

            boolean queryType1 = isQueryValid(query, RegexConstant.INSERT_WITHOUT_COLUMN);
            boolean queryType2 = isQueryValid(query, RegexConstant.INSERT_WITH_COLUMN);

            if (queryType1 | queryType2) {
                isQueryValid = true;
            }

            if (!isQueryValid) {
                this.errorMessage = "Invalid insert query syntax.";
            }

        }

        if (query.toLowerCase().contains("delete")) {

```

Figure 64: *Parse query code snippet*



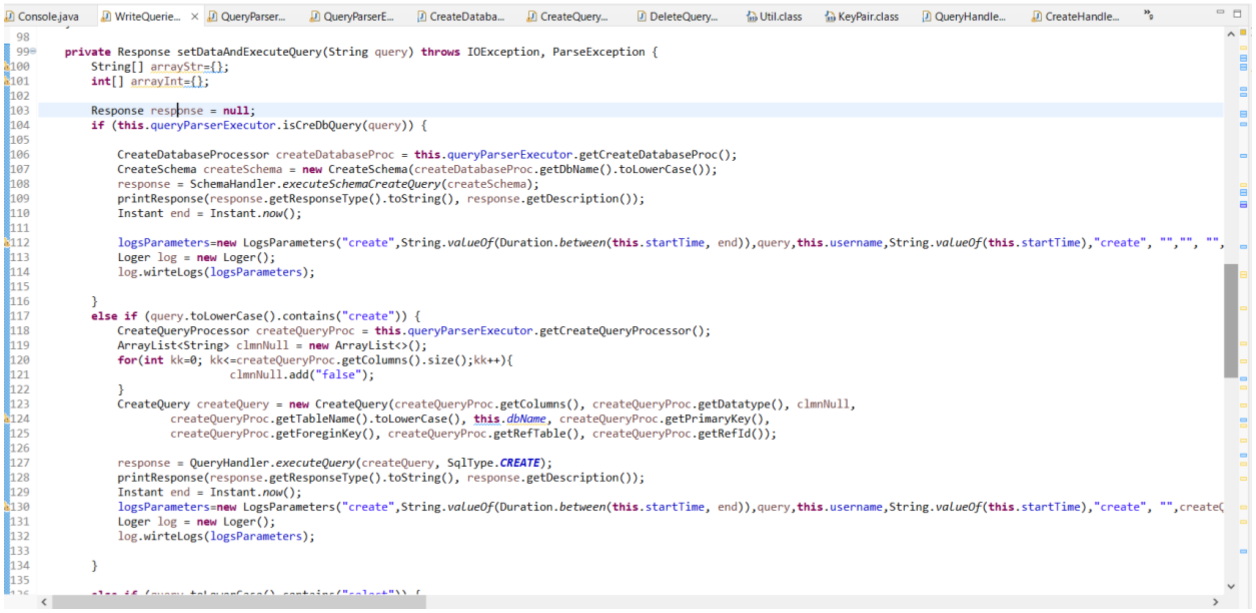
```java
    public QueryParserExecutor() {
        this.queryParser = new QueryParser();

    }

    public boolean processQuery(String query) throws InvalidQueryException {

        boolean isQueryProcessed = false;

        isQueryProcessed = this.queryParser.parseQuery(query);

        if (!isQueryProcessed) {
            throw new InvalidQueryException(this.queryParser.getErrorMessage());
        }
        if (isCreDbQuery(query)) {
            this.createDatabaseProc = new CreateDatabaseProcessor();
            createDatabaseProc.parseCreDataQuery(query);
            System.out.println("Query has been parsed :"+createDatabaseProc.toString());
        }

        if (query.toLowerCase().contains("create")) {
            this.createQueryProcessor = new CreateQueryProcessor();
            createQueryProcessor.parseCreateQuery(query);
            System.out.println("Query has been parsed :"+createQueryProcessor.toString());
        }

        if (query.toLowerCase().contains("insert")) {
            this.insertQueryProcessor = new InsertQueryProcessor();
            insertQueryProcessor.parseInsertQuery(query);
            System.out.println("Query has been parsed :"+insertQueryProcessor.toString());
        }

        if (query.toLowerCase().contains("delete")) {
            this.deleteQueryProcessor = new DeleteQueryProcessor();
            deleteQueryProcessor.parseDeleteQuery(query);
            System.out.println("Query has been parsed :"+deleteQueryProcessor.toString());
        }

```

Figure 65: *Process query code snippet*



```
    }

    private void parser1(String regx, String query) {

        final Pattern pattern = Pattern.compile(regx, Pattern.MULTILINE | Pattern.CASE_INSENSITIVE);
        final Matcher matcher = pattern.matcher(query);

        while (matcher.find()) {

            String temp = matcher.group(1).substring(1,matcher.group(1).length());

            String metadata[] = temp.split(",");

            for (String data : metadata) {
                data=" "+data;
                String columnData[] = data.trim().split("\\s+");


                this.columns.add(columnData[0]);
                this.datatype.add(columnData[1]);

                if (columnData.length > 2) {
                    if(data.toLowerCase().contains("primary")) {
                        this.primaryKey=columnData[0];
                    }

                    if(data.toLowerCase().contains("foreign")) {

                        String refData=columnData[5];
                        this.foreginKey=columnData[0];
                        this.refTable=refData.substring(0, refData.indexOf("("));
                        this.refId=refData.substring(refData.indexOf("(")+1, refData.indexOf(")"));

                    }

                }

            }

```

Figure 66: *Code to extract query details*



```

    private Response setDataAndExecuteQuery(String query) throws IOException, ParseException {
        String[] arrayStr={};
        int[] arrayInt={};

        Response response = null;
        if (this.queryParserExecutor.isCreDbQuery(query)) {

            CreateDatabaseProcessor createDatabaseProc = this.queryParserExecutor.getCreateDatabaseProc();
            CreateSchema createSchema = new CreateSchema(createDatabaseProc.getDbName().toLowerCase());
            response = SchemaHandler.executeSchemaCreateQuery(createSchema);
            printResponse(response.getResponseType().toString(), response.getDescription());
            Instant end = Instant.now();

            logsParameters=new LogsParameters("create",String.valueOf(Duration.between(this.startTime, end)),query,this.username,String.valueOf(this.startTime),"create", "","", "",
            Loger log = new Loger();
            log.wirteLogs(logsParameters);

        }
        else if (query.toLowerCase().contains("create")) {
            CreateQueryProcessor createQueryProc = this.queryParserExecutor.getCreateQueryProcessor();
            ArrayList<String> clmnNull = new ArrayList<>();
            for(int kk=0; kk<=createQueryProc.getColumns().size();kk++){
                            clmnNull.add("false");
            }
            CreateQuery createQuery = new CreateQuery(createQueryProc.getColumns(), createQueryProc.getDatatype(), clmnNull,
                    createQueryProc.getTableName().toLowerCase(), this.dbName, createQueryProc.getPrimaryKey(),
                    createQueryProc.getForeginKey(), createQueryProc.getRefTable(), createQueryProc.getRefId());

            response = QueryHandler.executeQuery(createQuery, SqlType.CREATE);
            printResponse(response.getResponseType().toString(), response.getDescription());
            Instant end = Instant.now();
            logsParameters=new LogsParameters("create",String.valueOf(Duration.between(this.startTime, end)),query,this.username,String.valueOf(this.startTime),"create", "",createQ
            Loger log = new Loger();
            log.wirteLogs(logsParameters);

        }

```

Figure 67: *Set data method for further execution*

Individual Contribution

Name	Domain worked
Aman Singh Bhandari	Query Execution, Distribution layer, VM instances setup, Distribute Database Design
Amankumar Patel	User Interface & Login Security, Query Parser, Query Processing, Distribute Database Design
Vivekkumar Patel	User Interface & Login Security, Query Parser, Query Processing, Distribute Database Design
Qiwei Sun	Analytics, Log management, Data Dump, Distribute Database Design
Vatsal Yadav	Data Modelling – Reverse Engineering, Transaction Processing, Distribute Database Design

Meeting Logs

Meeting Number	Date	Duration	Attendees	Reason	Location
1	2022-02-03 9:00	53m	All member	Brainstorming and discussion on the understanding the problem better. Successfully achieved.	Teams
2	2022-02-07 10:00	1h37m	All member	Dividing problem in to subproblems.	Teams

3	10-Feb	1h47	All member	Decided on having an in-person meeting/discussion on coming Monday to finalise our design.	Teams
4	2022-03-15	1h	All member	Discuss about the design and architecture.	Teams
5	2022-04-07	Entire Day	All member	Working on project	Goldberg
6	2022-04-08	Entire Day	All member	Working on project	Goldberg
7	2022-04-09	Entire Day	All member	Working on project	Goldberg
8	2022-04-10	Entire Day	All member	Integration and fix bugs	Goldberg
9	11-Apr-22	12m	All member	Prepare for the group presentation	Teams

References

[1] "JCraft," *JSch - Java Secure Channel*. [Online]. Available: http://www.jcraft.com/jsch/. [Accessed: 14-Apr-2022].

[2] "Java JSCH example to run shell commands on SSH unix server," *JournalDev*, 03-Mar-2021. [Online]. Available: https://www.journaldev.com/246/jsch-example-java-ssh-unix-server. [Accessed: 14-Apr-2022].

[3] "Topological Sorting." *Wikipedia*, 27-May-2021. [Online]. Available: https://en.wikipedia.org/wiki/Topological_sorting [Accessed: 14-Apr-2022].

[4] "Topological Sort with Good Explanation to Understand," *S, Vignesh.*, *Leetcode.com*, 4-Apr-2022. https://leetcode.com/problems/course-schedule/discuss/1912984/Topological-sort-with-good-explanation-to-understand [Accessed: 14-Apr-2022].

[5] F. Dib, "regex101: build, test, and debug regex," *regex101*, 2022. [Online]. Available: https://regex101.com/r/RrVdAx/1. [Accessed: Apr. 14, 2022]

[6] http://www.facebook.com/HowToDoInJAVA, "Java AES Encryption Decryption Example - HowToDoInJava," *HowToDoInJava*, Jun. 10, 2016. [Online]. Available: https://howtodoinjava.com/java/java-security/java-aes-encryption-example/. [Accessed: Apr. 14, 2022]