DATA 225 – GROUP PROJECT

LARGE DATA SET IMPLEMENTATION AND EVALUATION

COVID-19 CASE SURVEILLANCE ANALYSIS

Group 3

Gayathri Sundareshwar

Keerthana Gopikrishnan

Deepasha Jenamani

Acknowledgement

Working on research like this has been a large help in preparing and equipping us with skills necessary to thrive in this industry. It isn't an exaggeration to say that this project work is a golden opportunity. It is vital to extend our gratitude to Professor Simon Shim, his constant support in clarifying the queries has helped us immensely during this whole semester especially while working on this project.

In addition, we would like to extend our heartfelt thanks to all the contributors listed in the references. Last but certainly not the least, we would like to thank our parents who gave us the wings to fly and the members of this team for their timely contribution and continued support.

Abstract

We live in an era of IT explosion. When confronted with a flood of information, people frequently fail to organize their data in an effective and efficient manner. Furthermore, what constitutes information in one system may be incomprehensible in another. However, when used correctly, this information can provide solutions to long-standing problems. The impact of information derived from raw data on decision-making is critical. The objective of this project is to design and optimize a storage solution for a large dataset based on the relational database model. This project relies on a monthly updated dataset for all COVID19 cases shared with the Centers for Disease Control and Prevention (CDC), which includes patient level data reported by states and autonomous reporting entities in the United States. Each record contains information about a patient's demographics, geography (state and county of residence), exposure status, case status and month, underlying medical conditions, and a few other details about the case and its outcomes. The data in the model can be used to analyze the case patterns in a patient based on some parameters such as age, gender, or any specific medical conditions for each. It can show the severity of any area based on its location. The model will also aid in determining the patterns of case spread and the underlying factors. Thus, tracking the pattern over time will help control the spread of this disease, and this model may be useful in providing significant steps to avert any future medical emergency. Along with this, we will also be exploring on optimizing this in a way that we shall end up with a more efficient solution which involves reducing the memory usage footprint and decreased fetching time.

List of Tables and Figures

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Objective

The objective of this project is to design a model for a large ongoing data set and explore options in relational database with optimization of schema to support large scale and domain specific qualities which can be used to analyses pattern based on the specified parameters. The major advantage of identifying the performance driving factor for database allows us to avoid over-provisioning and reduce cost. It also gives us insights into whether moving data storage or adding server capacity will bring improvement in performance or not, and if so, then how much will it be. Optimizing the solution and performing exploratory testing across with different JDBC and ORM connection will also help us in identifying which approach suits the best.

1.2. What is the Problem

The business world is more data driven than ever. With the constant flush of data that is being flooded into the environment every second , it is now more than necessary to create a conservatory of data . This conservatory should not only be able to store larger sets but also should enable us to access and derive optimal insights whenever required in shorter time span. Mysql has its own limitations when dealing with larger datasets. What we are going to explore in this project is, finding out the optimal ways to deal with these limitations and arrive at an efficient solution.

2. ABOUT THE DATASET

The CDC website from where the dataset is picked from, has the following description in the “About Dataset” Section. It is , “The COVID-19 case surveillance database includes patient-level data reported by U.S. states and autonomous reporting entities, including New York City and the District of Columbia (D.C.), as well as U.S. territories and affiliates. On April 5, 2020, COVID-19 was added to the Nationally Notifiable Condition List and classified as "immediately notifiable, urgent (within 24 hours)" by a Council of State and Territorial Epidemiologists (CSTE) Interim Position Statement (Interim-20-ID-01). CSTE updated the position statement on August 5, 2020 to clarify the interpretation of antigen detection tests and serologic test results within the case classification (Interim-20-ID-02). The statement also recommended that all states and territories enact laws to make COVID-19 reportable in their jurisdiction, and that jurisdictions conducting surveillance should submit case notifications to CDC. COVID-19 case surveillance data collected by jurisdictions are shared voluntarily with CDC.

COVID-19 case reports are routinely submitted to CDC by public health jurisdictions using nationally standardized case reporting forms. On April 5, 2020, CSTE released an Interim Position Statement with national surveillance case definitions for COVID-19. Current versions of these case definitions are available at wwwn.cdc.gov/nndss/conditions/coronavirus-disease-2019-covid-19/. All cases reported on or after were requested to be shared by public health departments to CDC using the standardized case definitions for lab-confirmed or probable cases. On May 5, 2020, the standardized case reporting form was revised. States and territories continue to use this form”.

Whereas Figure.1 depicts the sample of the dataset downloaded from CDC website. This dataset is regularly updated and contains tens of millions of records. For our project purpose, we

have extracted 3.4 million records which were present at the time of extraction specifically for the month 2021-12.

The dataset is extracted from the CDC website, and it consists of the columns case_month, res_state, state_fips_code, res_county, county_fips_code, age_group, sex, race, ethnicity, case_positive_specimen_interval, case_onset_interval, process, exposure_yn, current_status, symptoms_status, hosp_yn, icu_yn, death_yn, undelying_conditions_yn. The description of these columns is depicted in the Figure.2.

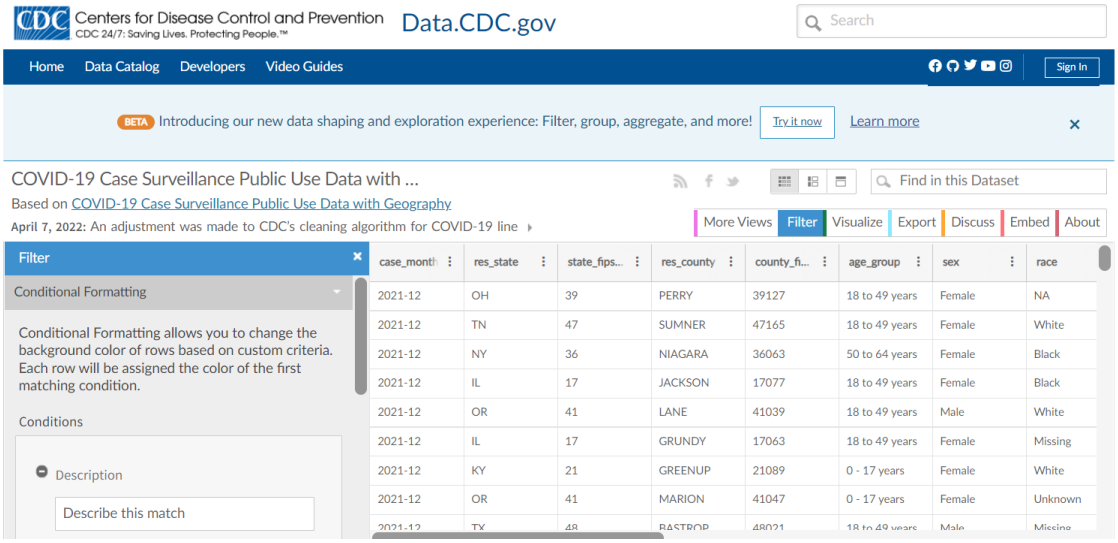


Figure.1 Data from the CDC Website

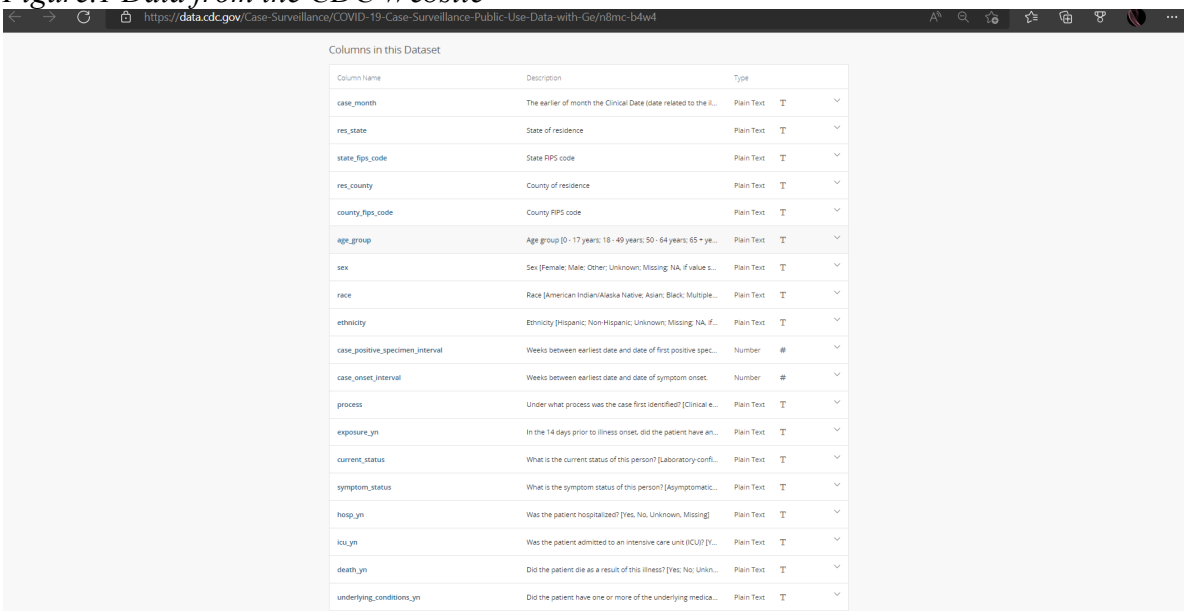


Figure.2 Column description present in CDC Website

3. MYSQL TABLE CREATION

In this section there is a description about the MySQL tables created for this model. Three different tables were created to cater to differ scenarios. Each of these tables will be helpful in finding out the optimal approach to reach an efficient working solution.

Figure 3,4 and 5 depicts the different tables that we have created to perform exploratory testing.

- Figure.3 depicts a MySQL table with partitions created in local instance.
- Figure.4 depicts a MySQL table without partitions created in local instance
- Figure.5 depicts a MySQL table created in AWS instance

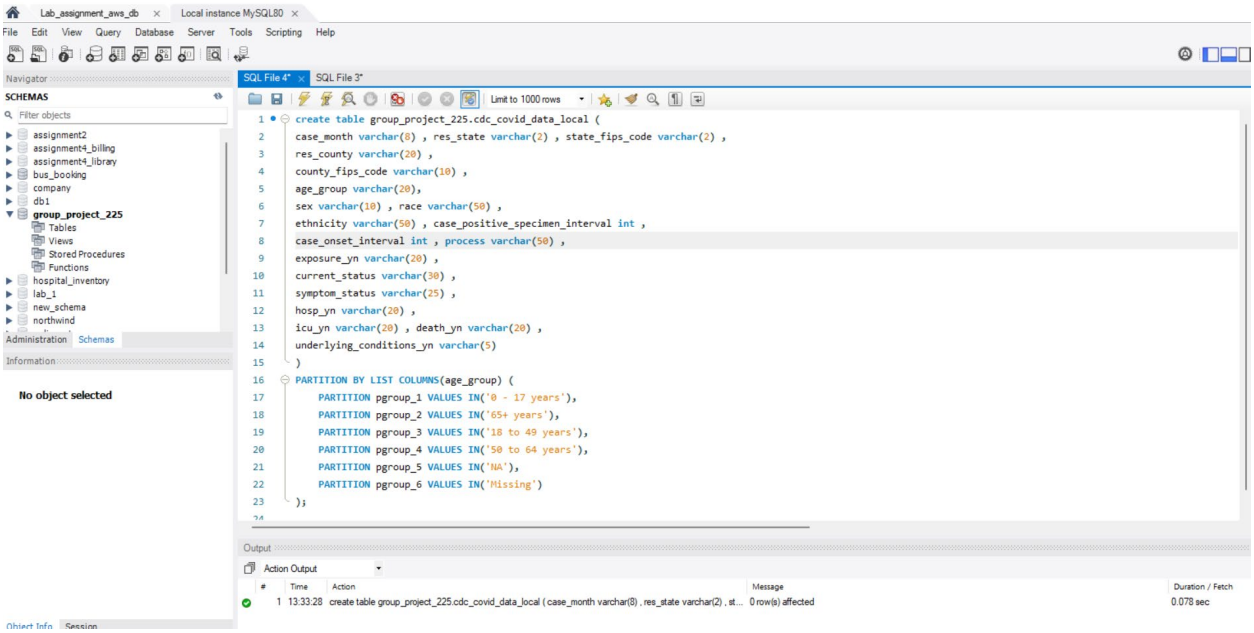


Figure.3 MySQL Table with partitions created in local instance

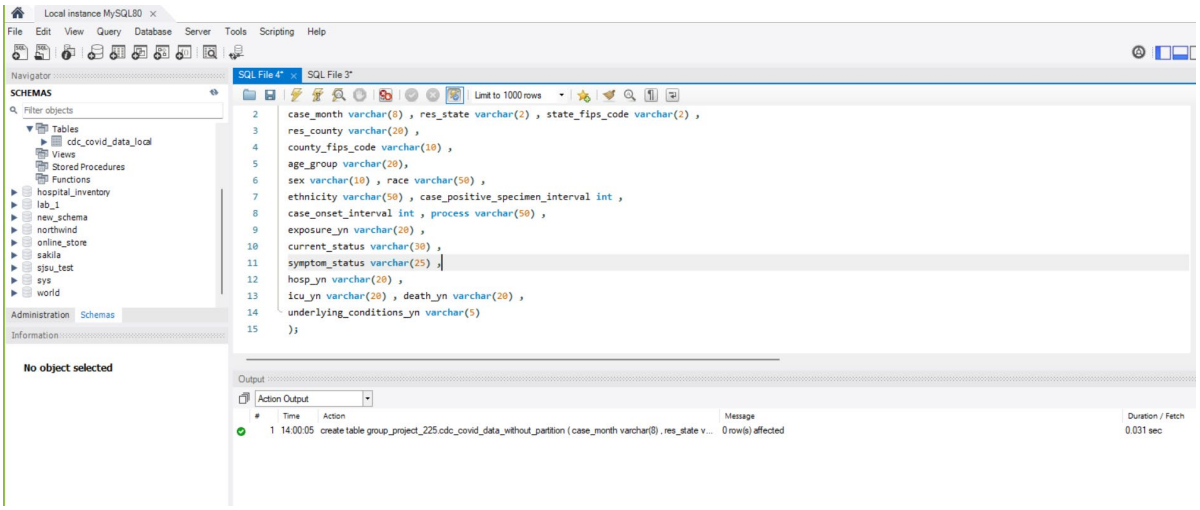


Figure.4 MySQL Table WITHOUT PARTITIONS created in local instance

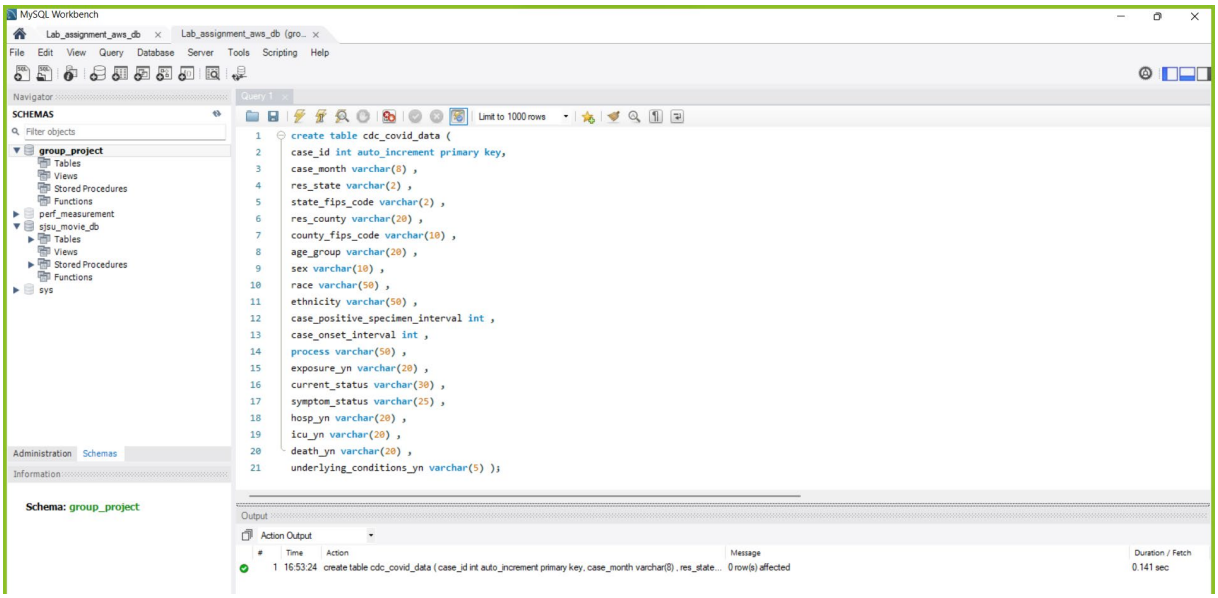


Figure.5 MySQL Table created in AWS

The usage of these different tables and how it comes to play when diving further deep into the project are explained in the sections following.

4. ACCESS CONTROL

To protect sensitive data from unauthorized access, it is necessary to implement a security feature. This is a mandatory step because unauthorized changes to the data cause a loss of trust in the system and demonstrate that the system is not robust. Assume, with regards to this model's data, a patient does not want to disclose his or her medical conditions and only shares the data for health analysis. However, if the system is hacked, the patient's data is compromised, resulting in a breach of privacy. As a result, adhering to highest level of security standards is a necessary step.

There are several types of access control, including discretionary access control (DAC), role-based access control (RBAC), mandatory access control (MAC), and attribute-based access control (ABAC). The strategy implemented for this model is Role Based Access Control. This approach incorporates three types of user roles, which are listed below.

Admin - Gayathri :

- A user with this role has complete access to the database. This role allows a user to perform read, modify, and delete operations. It also enables the user to perform administrative tasks such as database shutdown and backup. For this model, a user named 'gayathri' with this role is created to test this. Figures 7, 8 and 9 show how the user with this role can successfully insert, update, and delete.

Manager – Keerthana :

- This role has fewer privileges than the admin role. A user with this role has limited read and write access, which allows the user to insert and access any record but not delete it. To test this scenario, a user named 'keerthana' with this role is created. Figure.10 shows that the user with this role can successfully insert a record, whereas Figure.11 shows how a delete operation is denied.

Other Employees – Deepasha :

This is the role with the fewest privileges. A user with this role can only access the records and cannot delete or update them. For validation, a user named 'deepasha' with this role is created. Figure.12 depicts a successful access operation of records with this role. However, permission is denied inserting any record is shown in Figure.13.

Table.1 lists the the three created users named “gayathri”, “keerthana” and “deepasha” with three different levels of privileges granted. This grants higher data security since most of the times these datasets deal with sensitive data.

User Name	Permission Granted	Permission tested	Result of the test
Gayathri	All Permissions	Insert	Data Inserted
		Update	Data Updated
		Delete	Data Deleted
Keerthana	Select, Insert	Insert	Data Inserted
		Delete	Permission Denied
Deepasha	Select	Insert	Permission Denied
		Select	Output displayed

Table. 1 User access control

Figure.6 presents the successful creation of the three types of users and their associated roles in the database instance using MySQL workbench.

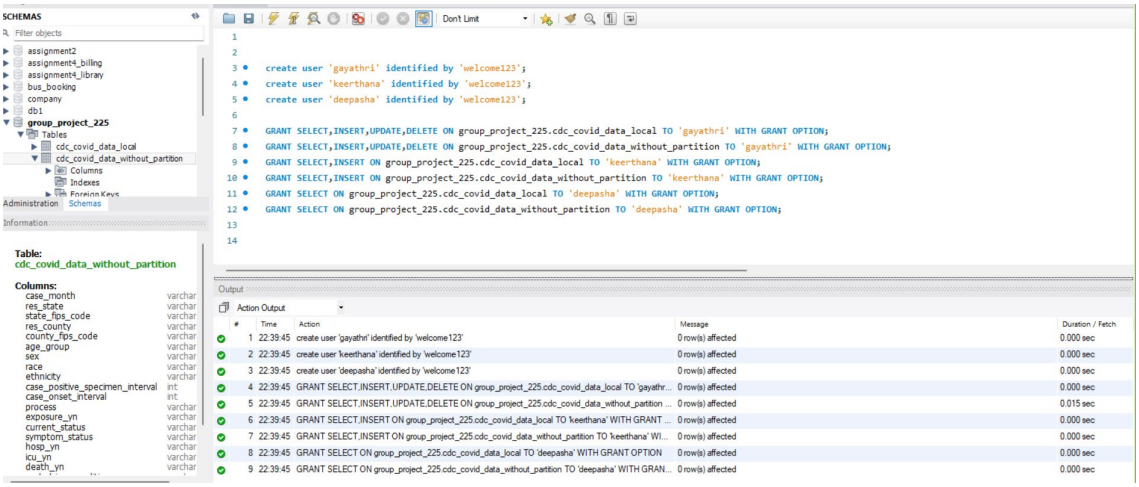


Figure.6 Users creation

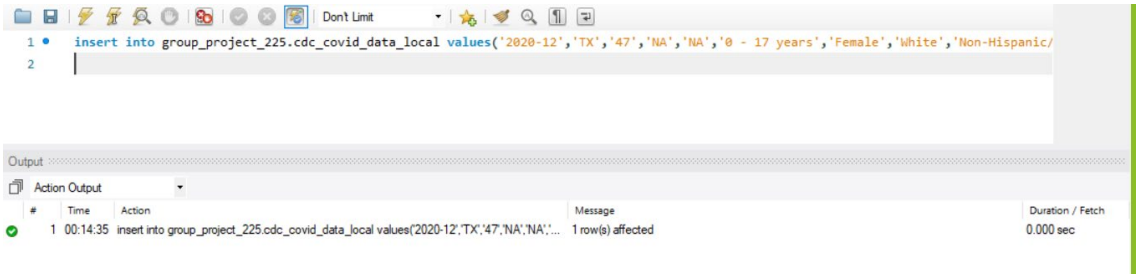


Figure. 7 Admin / Gayathri – Insert privilege check

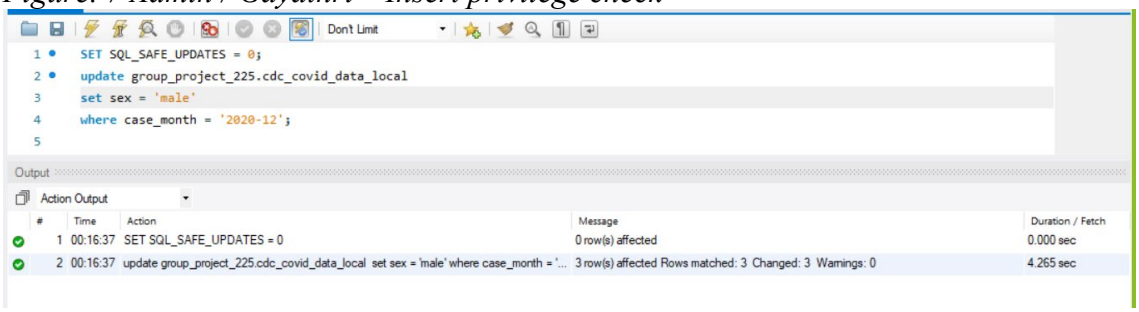


Figure.8 Admin / Gayathri – Update privilege check

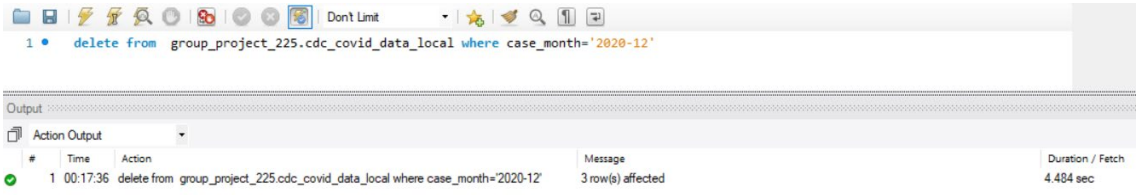


Figure.9 Admin / Gayathri – Delete privilege check

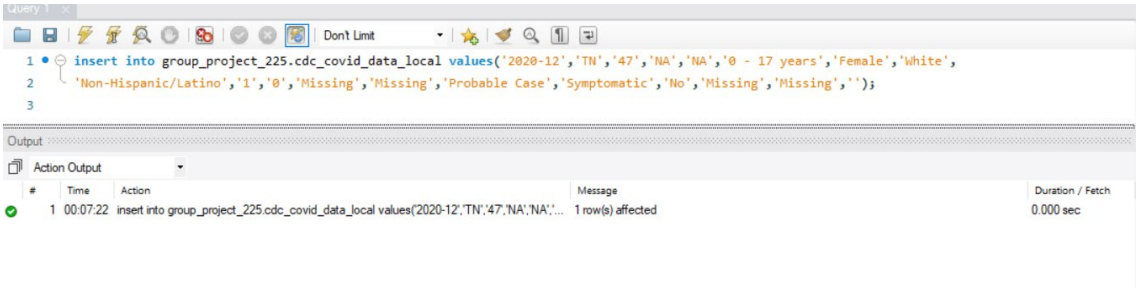


Figure.10 Manager / Keerthana – Insert privilege check

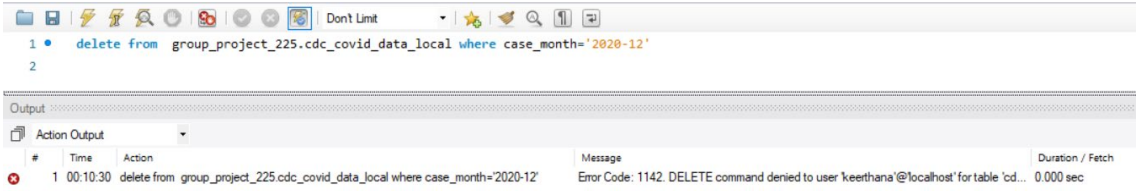


Figure.11 Manager / Keerthana – Delete privilege denied

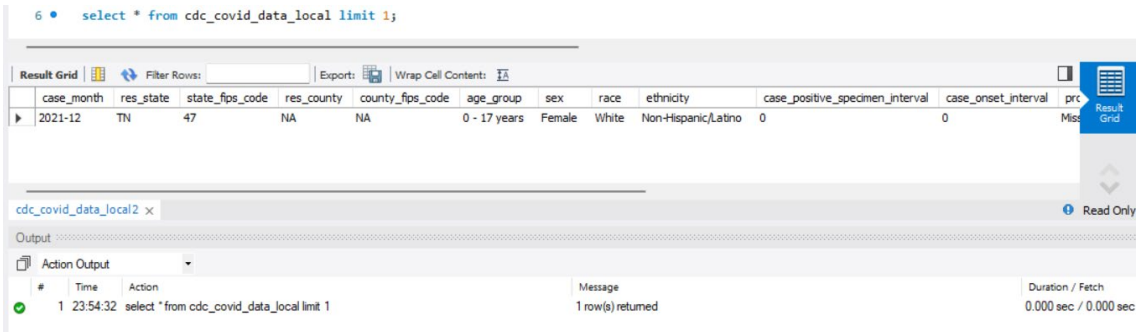


Figure.12 Employee / Deepasha – Read privilege check

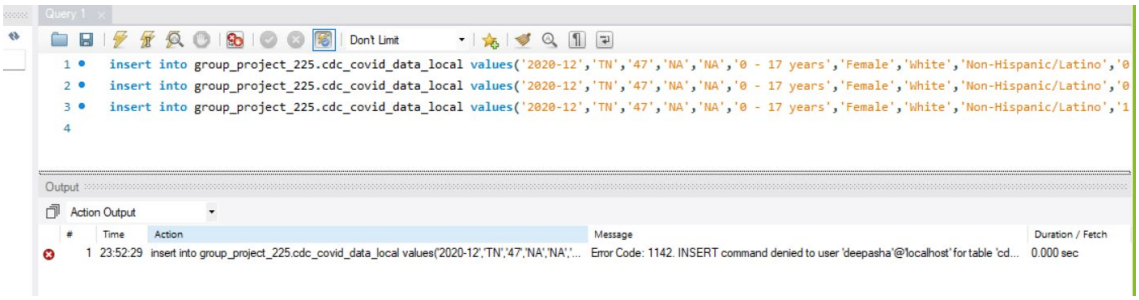


Figure.13 Employee / Deepasha – Insert privilege denied

5. AUDIT TABLE CREATION

Unauthorized access can be prevented with user access. But what happens if there is a SQL injection or if anyone with a low-privileged role performs a task that requires high-privileged access? As a result, all sensitive information can be compromised. Assume, for the purposes of this model, that a patient confirms that he or she was affected by covid and was traveling to other locations. In that case, it is necessary to track and record other people with whom the patient interacted to protect them from disease transmission and to warn them to remain isolated from others. However, if any user with manager access to this system logs in as an administrator and modifies some records. In that case, the data becomes contaminated and inconsistent. As a result, it is preferable if the database could keep a snapshot of every change to the data. This facilitates tracing back in time and validating for inconsistencies.

An audit table is maintained to keep track of user activities or any unauthorized access when using the audit logging strategy. This table records all user activities across all tables that have been created. If any changes are made to the original table, such as insertion, modification, or deletion, an entry will be created in the audit table. The created entry will also save details such as the user who made the change, the timestamp at which the change was made, and the action performed.

Figure.14 depicts the action output of all the audit tables and respective triggers created. Figure.15 shows the two create audit tables that have been created. ‘audit_cdc_covid_data_local’ able stores the audit data related the partitioned table. Whereas the other table which is, ‘audit_cdc_covid_data_without_partition’ records the audit logs related to the non-partitioned table.

Action Output

#	Time	Action	Message	Duration / Fetch
1	22:42:02	DROP TABLE IF EXISTS `group_project_225`.`audit_cdc_covid_data_local`	0 row(s) affected	0.031 sec
2	22:42:02	CREATE TABLE `group_project_225`.`audit_cdc_covid_data_local` (`auditAction` ENUM ('INSERT', 'U...	0 row(s) affected	0.047 sec
3	22:42:02	DROP TRIGGER IF EXISTS `group_project_225`.`cdc_covid_data_local_inserts`	0 row(s) affected	0.016 sec
4	22:42:02	CREATE TRIGGER `group_project_225`.`cdc_covid_data_local_inserts` AFTER INSERT ON `group_pr...	0 row(s) affected	0.015 sec
5	22:42:02	DROP TRIGGER IF EXISTS `group_project_225`.`cdc_covid_data_local_updates`	0 row(s) affected	0.000 sec
6	22:42:02	CREATE TRIGGER `group_project_225`.`cdc_covid_data_local_updates` AFTER UPDATE ON `group_...	0 row(s) affected	0.000 sec
7	22:42:02	DROP TRIGGER IF EXISTS `group_project_225`.`cdc_covid_data_local_deletes`	0 row(s) affected	0.032 sec
8	22:42:02	CREATE TRIGGER `group_project_225`.`cdc_covid_data_local_deletes` AFTER DELETE ON `group_p...	0 row(s) affected	0.015 sec
9	22:42:02	DROP TABLE IF EXISTS `group_project_225`.`audit_cdc_covid_data_without_partition`	0 row(s) affected	0.016 sec
10	22:42:02	CREATE TABLE `group_project_225`.`audit_cdc_covid_data_without_partition` (`auditAction` ENUM (1...	0 row(s) affected	0.047 sec
11	22:42:02	DROP TRIGGER IF EXISTS `group_project_225`.`cdc_covid_data_without_partition_inserts`	0 row(s) affected	0.015 sec
12	22:42:02	CREATE TRIGGER `group_project_225`.`cdc_covid_data_without_partition_inserts` AFTER INSERT O...	0 row(s) affected	0.016 sec
13	22:42:02	DROP TRIGGER IF EXISTS `group_project_225`.`cdc_covid_data_without_partition_updates`	0 row(s) affected	0.000 sec
14	22:42:02	CREATE TRIGGER `group_project_225`.`cdc_covid_data_without_partition_updates` AFTER UPDATE ...	0 row(s) affected	0.016 sec
15	22:42:02	DROP TRIGGER IF EXISTS `group_project_225`.`cdc_covid_data_without_partition_deletes`	0 row(s) affected	0.015 sec
16	22:42:03	CREATE TRIGGER `group_project_225`.`cdc_covid_data_without_partition_deletes` AFTER DELETE ...	0 row(s) affected	0.016 sec

Figure.14 Action output of Audit tables and trigger creation

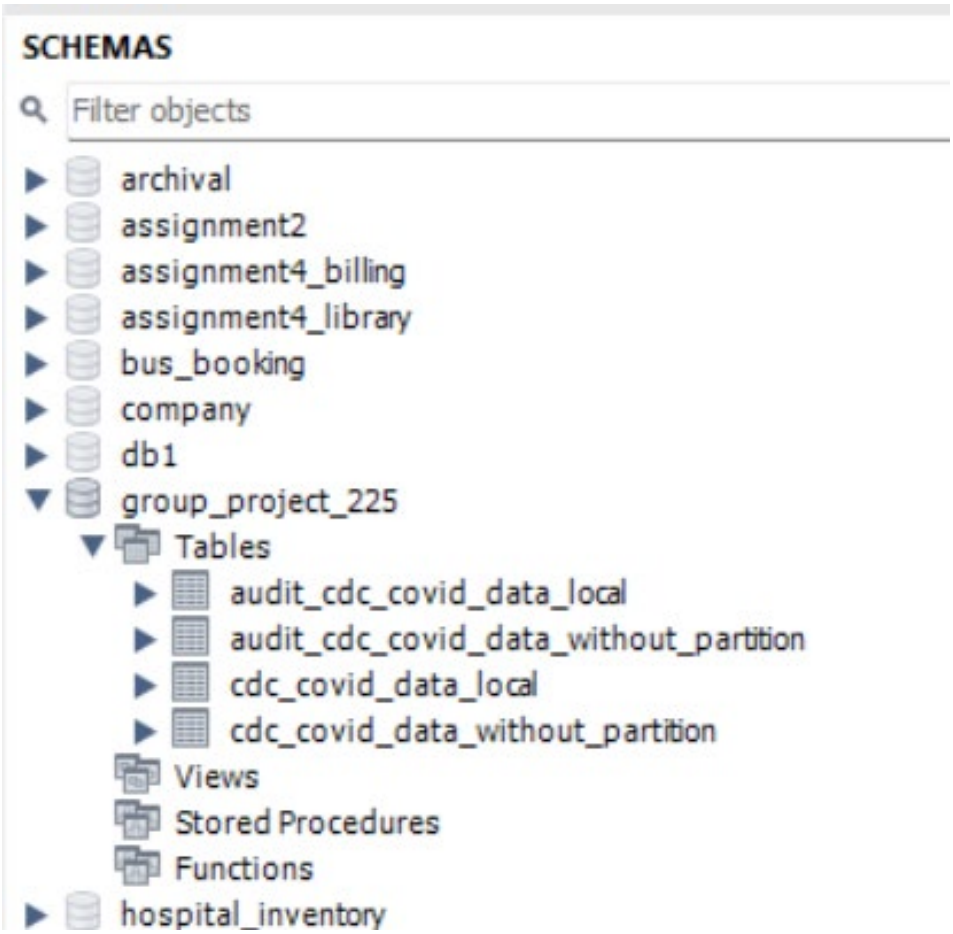


Figure.15 Tables present in database

Figure.16 depicts the audit table 'audit_cdc_covid_data_local' with entries logged with the user information and timestamp for each of the changes made to the original table 'cdc_covid_data_local'.

Figure.17 depicts the audit table ‘audit_cdc_covid_data_without_partition’ with entries logged with the user information and timestamp for each of the changes made to the original table ‘cdc_covid_data_without_partition’.

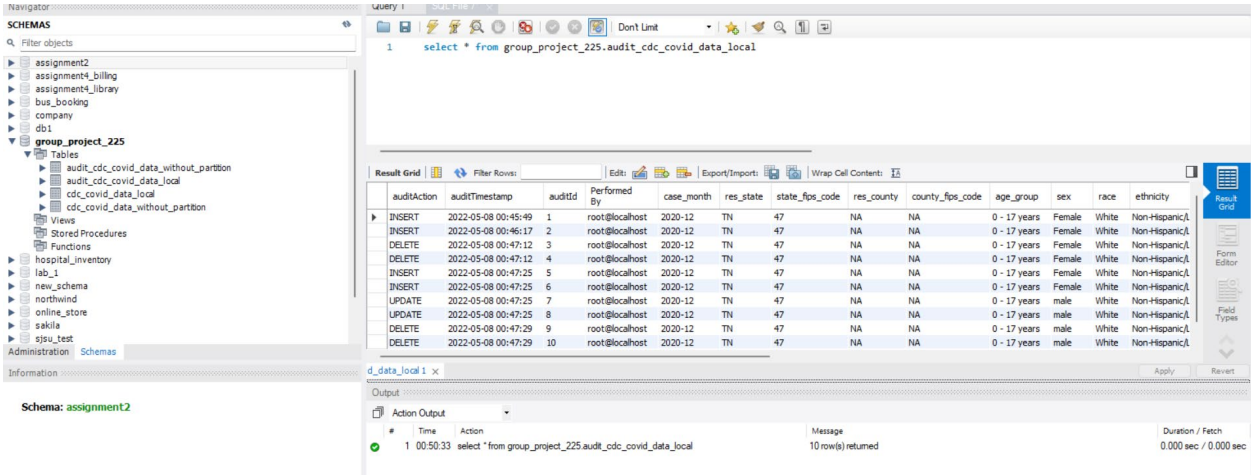


Figure.16. Audit table ‘audit_cdc_covid_data_local’ with logged entries

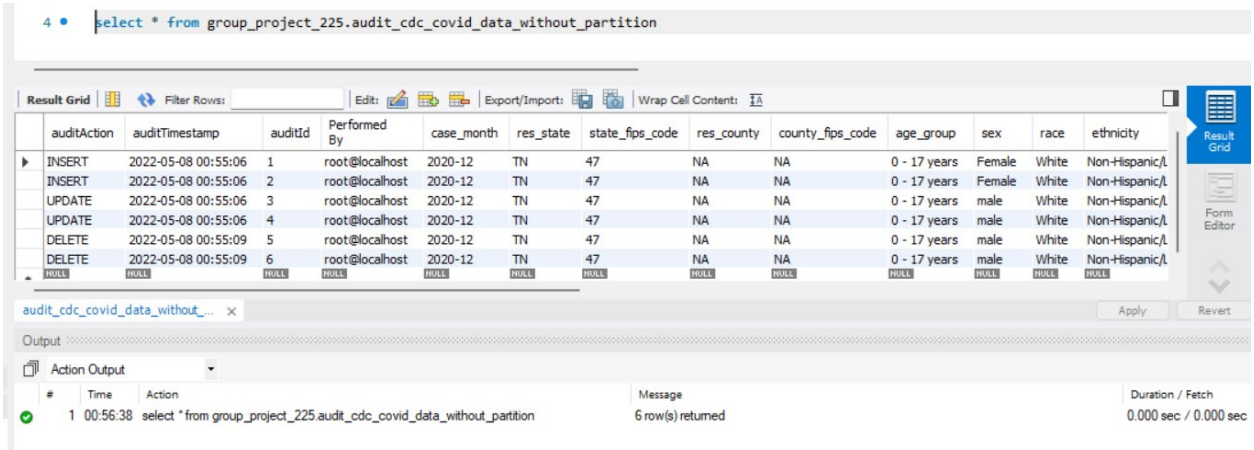


Figure.17 Audit table ‘audit_cdc_covid_data_without_partition’ with logged entries

From the above figures it is observed that, that the log has entries for various kinds of operations such as ‘INSERT’, ‘UPDATE’ and ‘DELETE’ are all recorded as expected along with the additional vital information such as , the user performing the change, the time at which the changes were made.

Table.2 shows the audit tables that are created for both the partitioned and non-partitioned instances. There are three types of triggers created for each instance that run when any inserting, updating, or deleting operation occurs.

Audit table name	Action	Description
audit_cdc_covid_data_local	Audit table creation	An audit table to log the changes made in the original table is created
	Insert trigger creation	This trigger will keep track of all the insert made in the table
	Update trigger creation	This trigger will keep track of all the updates made in the table
	Delete trigger creation	This trigger will keep track of all the deletes made in the table
audit_cdc_covid_data_without_partition	Audit table creation	An audit table to log the changes made in the original table is created
	Insert trigger creation	This trigger will keep track of all the insert made in the table
	Update trigger creation	This trigger will keep track of all the updates made in the table
	Delete trigger creation	This trigger will keep track of all the deletes made in the table

Table. 2 Audit tables

6. LOAD OPTIMIZATION

The dataset used for relational model design in this case is a massive, structured dataset with nineteen columns and 3.4 million records. If we import the dataset into mysql table directly all at once, the model's performance will suffer greatly. There are many ways to optimize the load

and most of them are confined to the hardware. There are a few approaches confined to software as well. The optimal solution used in this project is explained below.

6.1. Splitting the dataset into smaller chunks

Load optimization can be done by splitting the dataset into chunks of 100,000 records each. This way we will have files of smaller chunks each of similar size.

Figure 18 depicts the process of splitting the original file into chunks as described above. Splitting the files speeds up the data load when

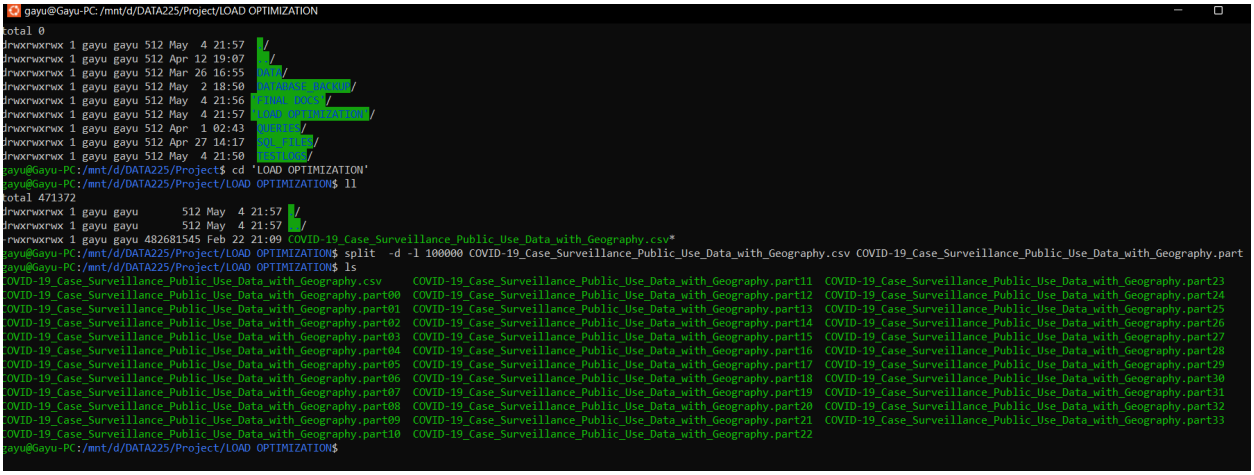


Figure.18 Splitting the file into chunks of 100,000 records each

The data is split using split command in Ubuntu. Since the original dataset has 3.4 million records. The split command ends up with 34 files each with 100,000 records in it.

6.2. Loading the split chunks into MySQL

The split files are now loaded into MySQL tables and the different possible approaches are explored in order to arrive at an optimal solution. Some of those explored approaches are listed below:

- Using insert command
- Using extended insert
- Load data local infile

Since the dataset has a significantly higher number of records, the insert and extended insert statements are lesser efficient ways to load. Although it can easily insert a couple of rows at once, it's ill-equipped to handle data sets more extensive than, say, a million rows. It is time consuming, and it is not optimal when dealing with memory usage.

Load data is more efficient than insert queries because it loads data in bulk. The server parses and interprets only one statement , instead of several. The other pro is that Load data infile consumes less memory IO disk space and CPU cycles. Since we have multiple files with hundred thousand records each, load data infile it is sensible to make use of load data local infile to optimize the data load.

Hence “load data local infile” command is used to load the split files into MySQL tables. The split files are now loaded into the three tables which are discussed in section 4.

- Figure 19 depicts the approximate time taken by Mysql Table WITHOUT partition to load 100,000 records.
- Similarly Figure 20 represents the time taken by Mysql table WITH partition to load the same chunk of 100,000 records.
- Apart from these two, we also have explored testing a third approach, which is AWS . Figure 21 represent the time taken by AWS Mysql table.

These three tables will come handy when analyzing the optimal solution for concurrent reads. This is the reason why different kinds of tables were created and loaded with same dataset.

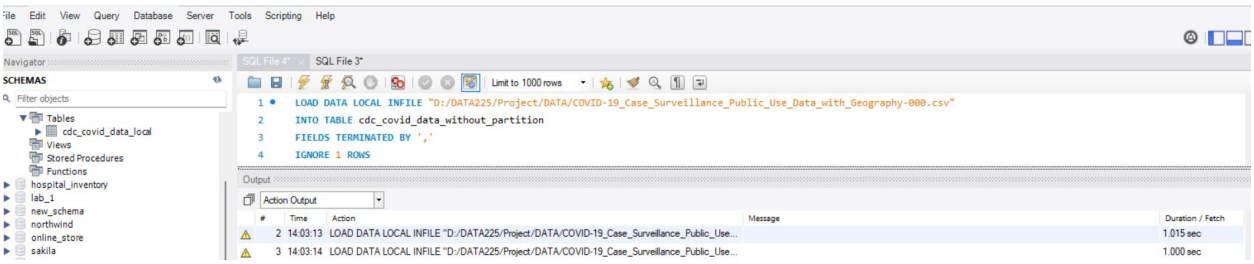


Figure.19 Loading files into Mysql table without partition

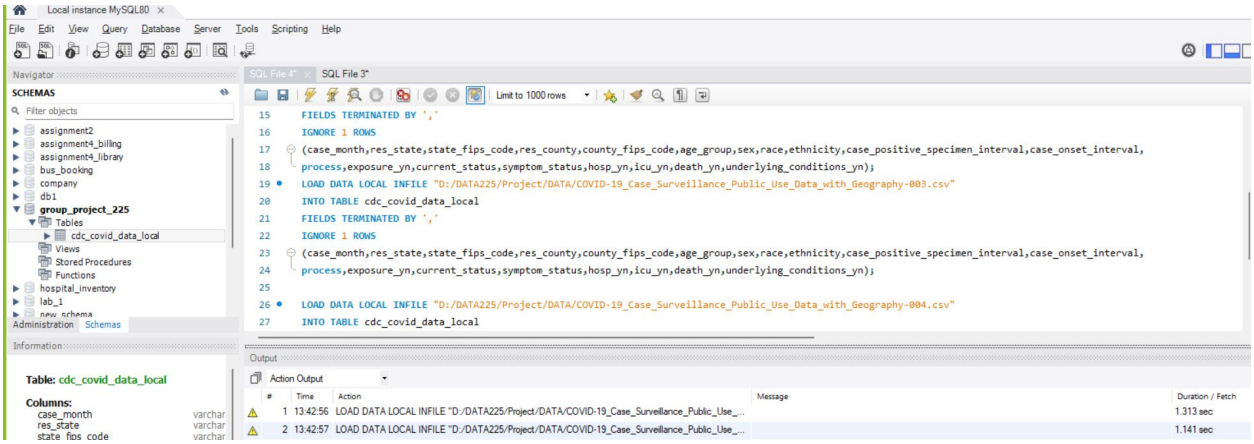


Figure.20 Loading files into Mysql table with partition

From figures 19 and 20, it is observed that there is not much difference in the time taken for a load query to run between the table with and without partition. This doesn't negatively impact the optimization of load in any way since the optimization part is covered by splitting the files.

Whereas Figure 21 depicts that the AWS MySQL table takes more time to load the data when compared with the local tables. This is an expected scenario since local loads always tend to work fasted when compared with the AWS loads due to the involvement of cloud.

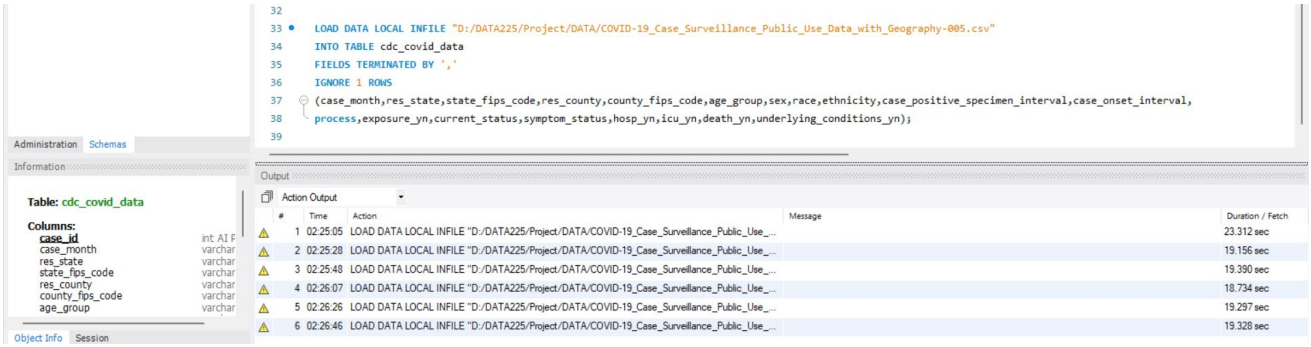


Figure.21 Loading files into AWS Mysql table

The advantage of loading into AWS is that it can be accessed from anywhere with the proper connection information and permissions. In the case of local, however, this is not the case. By weighing the benefits and drawbacks of the tested approaches, the best decision can be made. Because this project is run on a single server and by a limited number of users, a local solution appears to suffice. Another reason for choosing a local database for this project is that AWS is more expensive than a local instance.

7. CONNECTIVITY TO PYTHON

The key objective of this project is to explore the optimal solution that can be used to access the data from MySQL through other programs. To test the different possible scenarios, Python is used as a medium for this, and the implementation is carried out using a tool known as Jupyter notebook. This can come handy when additional analysis and representation are required apart from the normal query results attained from MySQL execution. Furthermore, the visualization feature present in these other programming languages where SQL is embedded can be used to curate the data into a more understandable format while highlighting trends and outliers. A MySQL adapter, such as mysqlclient, PyMySQL, or mysql-connector-python, is required to establish a database connection with the Python program. Mysql-connector-python, which is entirely written in Python and uses the MySQL API, is used for this model. For this project's purpose , Two approaches were tested. Detailed explanation regarding those two approaches can be found in detail in the following section.

7.1. JDBC Driver:

A database connection is established using the JDBC driver to transfer the query and result between the database and the client. MySQL Connector/Python allows users to connect to a MySQL database. It makes use of an API that is Python Database API compliant. It does not require any MySQL client libraries or Python modules other than those included with the standard library, making it a self-contained driver.

Figure.22 depicts the connector package that has been imported and the username and password read as input.

Enter the username and Password to connect to the MYSQL DB:

```
import mysql.connector
```

```
print(" Connecting to the localhost ")
global usrnm;
usrnm=input("Enter username to connect to the MYSQL DB: ")
global pwd;
pwd=getpass("Enter password  to connect to the MYSQL DB: ")
```

```
 Connecting to the localhost
Enter username to connect to the MYSQL DB: root
Enter password  to connect to the MYSQL DB: ·····
```

Figure.22 MySQL database connector package and input of username and password

Display the databases present in the connection:

```
from mysql.connector import connect, Error
try:
    with connect(
        host="localhost",user=usrnm, password=pwd ) as connection:
        show_db_query = "SHOW DATABASES"
        with connection.cursor() as cursor:
            cursor.execute(show_db_query)
            for db in cursor:
                print(db)
except Error as e:
    print("There is an error: {} while connecting to the database.".format(e))
```

```
('assignment2',)
('assignment4_billing',)
('assignment4_library',)
('bus_booking',)
('company',)
('db1',)
('group_project_225',)
('hospital_inventory',)
```

Figure.23 MySQL database connection and result displaying all tables.

Figure.23 depicts how, a successful connection is established and following that successful connection, how a SQL query is executed, and displays the result set retrieved in the output panel. The error message is logged in the event of an error.

The 'cursor' object is a MySQLCursor instance that performs the SQL statement execution. SQL statement execution can thus take place between the Python program and the database server. In this case, however, whenever a query is executed, it connects to the database. As a result, an alternative approach, Object Relational Mapping (ORM), is described in the following section.

7.2. Object Relational Mapper (ORM):

Object relational is a code library that automates the transfer of data from a relational database into application-specific objects. ORM allows the user to read, write, or update data

stored in a database using application-specific objects such as Python objects by providing a high-level abstraction over the relational database and thus removing any query parsing dependency with the database.

There are many ORM implementations for Python are available. Some of which are listed below.

- SQLAlchemy
- Peewee
- The Django ORM

SQLAlchemy is used in the proposed model. It provides a generalized interface for running database-compliant code without the need to write any SQL statements.

Figure.24 shows that after importing the SQLAlchemy library, an Engine object named 'engine' is created by calling the create engine() method with a URL that can include the hostname, database, username, and password. Because the connection URL in this model does not explicitly mention any dialect, mysql-python is used as the default DBAPI. An Engine instance represents the core database interface. However, calling create engine indicates a lazy connection, which means that the Engine will not attempt to connect to the database until it is asked to perform a database-related task.

```
import sqlalchemy as db
engine = db.create_engine('mysql://root:GaYu6793@localhost:3306/group_project_225')
```

Figure.24 ORM Engine instance creation for MySQL Database using SQLAlchemy

Figure.25 depicts how the data stored in the database tables can be retrieved after connecting. In the figure we shall observe that, the connection has been established, table can be accessed, and the metadata of the table is also displayed as output.

```
connection = engine.connect()
metadata = db.MetaData()
covid_data_partition = db.Table('cdcengine_covid_data_local', metadata, autoload=True, autoload_with=engine)
```

```
print(repr(metadata.tables['cdc_covid_data_local']))
```

```
Table('cdc_covid_data_local', MetaData(), Column('case_month', VARCHAR(length=8), table=<cdc_covid_data_local>), Co
lumn('res_state', VARCHAR(length=2), table=<cdc_covid_data_local>), Column('state_fips_code', VARCHAR(length=2), ta
ble=<cdc_covid_data_local>), Column('res_county', VARCHAR(length=20), table=<cdc_covid_data_local>), Column('county
_fips_code', VARCHAR(length=10), table=<cdc_covid_data_local>), Column('age_group', VARCHAR(length=20), table=<cdc_
covid_data_local>), Column('sex', VARCHAR(length=10), table=<cdc_covid_data_local>), Column('race', VARCHAR(length=
50), table=<cdc_covid_data_local>), Column('ethnicity', VARCHAR(length=50), table=<cdc_covid_data_local>), Column('
case_positive_specimen_interval', INTEGER(), table=<cdc_covid_data_local>), Column('case_onset_interval', INTEGER()
, table=<cdc_covid_data_local>), Column('process', VARCHAR(length=50), table=<cdc_covid_data_local>), Column('expos
ure_yn', VARCHAR(length=20), table=<cdc_covid_data_local>), Column('current_status', VARCHAR(length=30), table=<cdc
_covid_data_local>), Column('symptom_status', VARCHAR(length=25), table=<cdc_covid_data_local>), Column('hosp_yn',
VARCHAR(length=20), table=<cdc_covid_data_local>), Column('icu_yn', VARCHAR(length=20), table=<cdc_covid_data_local
>), Column('death_yn', VARCHAR(length=20), table=<cdc_covid_data_local>), Column('underlying_conditions_yn', VARCHA
R(length=5), table=<cdc_covid_data_local>), schema=None)
```

```
covid_data_without_partition = db.Table('cdc_covid_data_without_partition', metadata, autoload=True, autoload_with=e
```

```
print(repr(metadata.tables['cdc_covid_data_without_partition']))
```

```
Table('cdc_covid_data_without_partition', MetaData(), Column('case_month', VARCHAR(length=8), table=<cdc_covid_data
_without_partition>), Column('res_state', VARCHAR(length=2), table=<cdc_covid_data_without_partition>), Column('sta
te_fips_code', VARCHAR(length=2), table=<cdc_covid_data_without_partition>), Column('res_county', VARCHAR(length=20
), table=<cdc_covid_data_without_partition>), Column('county_fips_code', VARCHAR(length=10), table=<cdc_covid_data_
without_partition>), Column('age_group', VARCHAR(length=20), table=<cdc_covid_data_without_partition>), Column('sex
', VARCHAR(length=10), table=<cdc_covid_data_without_partition>), Column('race', VARCHAR(length=50), table=<cdc_cov
id_data_without_partition>), Column('ethnicity', VARCHAR(length=50), table=<cdc_covid_data_without_partition>), Col
umn('case_positive_specimen_interval', INTEGER(), table=<cdc_covid_data_without_partition>), Column('case_onset_int
erval', INTEGER(), table=<cdc_covid_data_without_partition>), Column('process', VARCHAR(length=50), table=<cdc_covi
d_data_without_partition>), Column('exposure_yn', VARCHAR(length=20), table=<cdc_covid_data_without_partition>), Co
lumn('current_status', VARCHAR(length=30), table=<cdc_covid_data_without_partition>), Column('symptom_status', VARC
HAR(length=25), table=<cdc_covid_data_without_partition>), Column('hosp_yn', VARCHAR(length=20), table=<cdc_covid_d
ata_without_partition>), Column('icu_yn', VARCHAR(length=20), table=<cdc_covid_data_without_partition>), Column('de
ath_yn', VARCHAR(length=20), table=<cdc_covid_data_without_partition>), Column('underlying_conditions_yn', VARCHAR(
length=5), table=<cdc_covid_data_without_partition>), schema=None)
```

Figure.25 ORM connection to MySQL Database using SQLAlchemy

8. LARGE / SMALL READS

The model includes data from the Centers for Disease Control and Prevention (CDC). The server is expected to send data repeatedly without requiring each piece of information to be provided in a separate transaction. As a result, this model is expected to be available and accessible to multiple users on a demand basis. Users from different demographic regions may access the data, but they may not always require the same number of records. The model employs the InnoDB storage engine, which is the default storage engine for MySQL. The primary advantage of this storage engine in terms of memory for high-volume reads is the buffer pool, which caches table and index data as it is accessed.

This caching allows the read to occur from main memory, reducing processing time. In some cases, the reads may exceed the available memory. As a result, it raises the question of how to find an optimal solution for handling large volumes of data when it is required to fetch different number of records in iterative fashion. Different approaches were put into test to find the most fitting solution. This is also one of the reasons why Different SQL tables (With, without partition and AWS) were created in the initial steps.

The SQL tables created with and without partition in the initial stage can now be put into use. The first approach is the MySQL table with partition,

Some of the types of MySQL partitioning are:

- RANGE
- LIST
- HASH
- COLUMNS
- KEY

For the proposed model, a 'LIST' type of partitioning is used, with the age group defining how the data should be partitioned. While performing reads and retrieving records of different size, insights can be attained on which is ideal for what size of data. It will also be useful to find the difference between the approaches of JDBC and ORM. These insights can be helpful when making decision on business requirements with similar sized or larger datasets. Figure.26 presents the partitioned tables created based on age group.

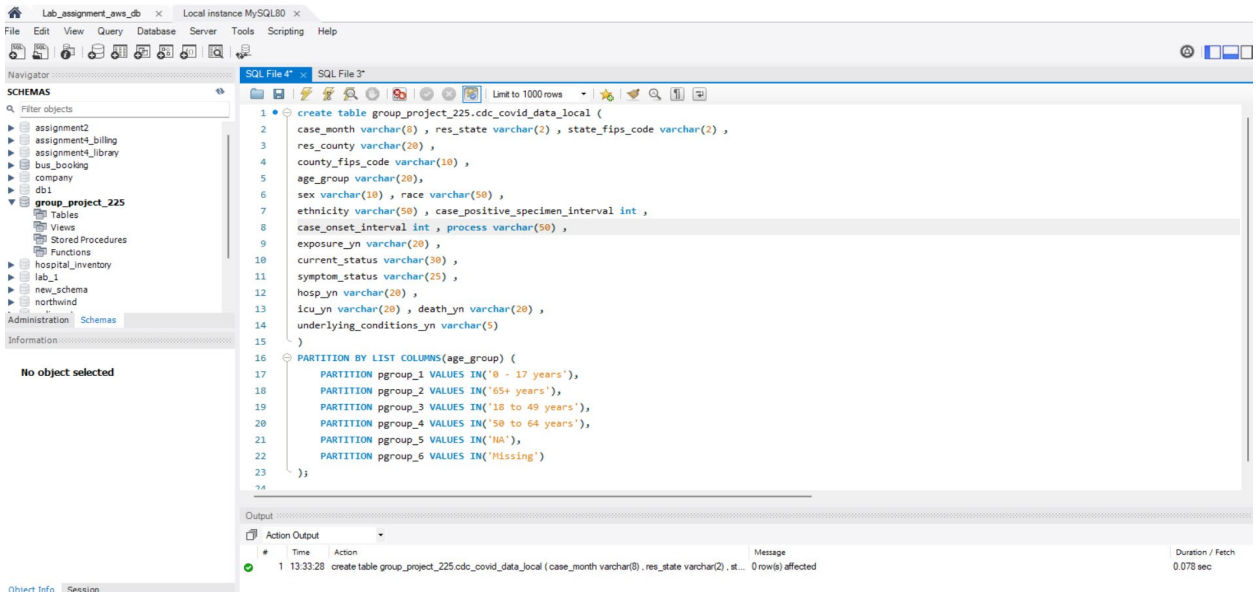


Figure.26 Table partitioning based on 'age_group'

The execution time of data retrieval operations from partitioned and non-partitioned tables is compared to find a best suited solution. The pre-requisite before proceeding with execution of these ideology is that the parameters that could influence the result of these exploratory tests should be the same for these tests , For example, if the partitioned table with an age group of less than 16 years is compared to the non-partitioned table, the records from the latter should only contain those records with an age group of less than 16 years. Insights attained from testing this way on both partitioned and non-partitioned instances with large and small read volumes holds more accuracy

than testing where the parameters mismatch. Furthermore, this comparison will also help in determining the performance differences between the JDBC and ORM approaches.

8.1. Large Reads

8.1.1. Large Reads Using JDBC

Following the reads ideology, large reads are performed in iterative manner with varied sizes . Time taken by each of these iterations will be measured for each of the approaches. These recorded timestamps will be valuable when deciding on which approach to use. To begin, a connection to both partitioned and non-partitioned tables is established using JDBC. As previously discussed, in order to arrive at a bias-free conclusion, the parameters are set the same for both tables. As for the parameter, a specific age group is selected and through iterative loop, data with incremental sizes is fetched. The age group of 18 to 49 years has the most records, with nearly 1.8 million. For privacy reasons, the age is not available in some scenarios, and there are 19,740 records in this category, which is displayed as 'NA'. Reads from both partitioned and non-partitioned tables will be performed and time taken by the connection to fetch the results is recorded for each attempt. The scenario tested here is, the iteration read value increments by 100,000 records on each attempt, and this continues until it reaches the limit of 1,80,000 records which is nothing, but the total number of records present in the selected age group. The insights attained from these tests are discussed in detail in the coming sections.

Figure.27 contains 3 columns; first one is count representing the number of records fetched in each of the run. Second one is, time taken by JDBC to fetch the records from a table without partition. And the final column is similarly the time taken by JDBC to fetch the records from a table with partition. Analyzing this data can be helpful in attaining useful insights . An observation

is made how a partitioned table fetches the data faster than a table without partition. There is also significant difference of approximately four seconds when reading one million records from the partitioned table versus the other. Also, as the record count increases, this time gap is still evident

Count	Time Taken - WO Partition	Time Taken - W Partition
100000	0.280434	0.266755
200000	0.732781	0.631928
300000	1.217474	0.863831
400000	1.618266	1.160356
500000	2.019552	1.643749
600000	2.551742	2.010968
700000	2.982881	2.426870
800000	3.501579	2.870222
900000	3.809089	3.230571
1000000	4.578667	3.778391
1100000	4.994658	4.136639
1200000	5.898068	4.829008
1300000	7.162713	5.656934
1400000	7.732898	5.648630
1500000	8.979501	5.971123
1600000	7.875977	6.333595
1700000	8.519996	6.904070
1800000	10.294916	7.211643

Figure.27 JDBC large reads execution time – Non partitioned vs Partitioned instance

Visualization has always proven to be more effective in conveying the data across to even the end users or executives in layman terms. Figure.28 is a line plot of the result in Figure.27. The partitioned instance takes less time to execute, as shown in the figure.

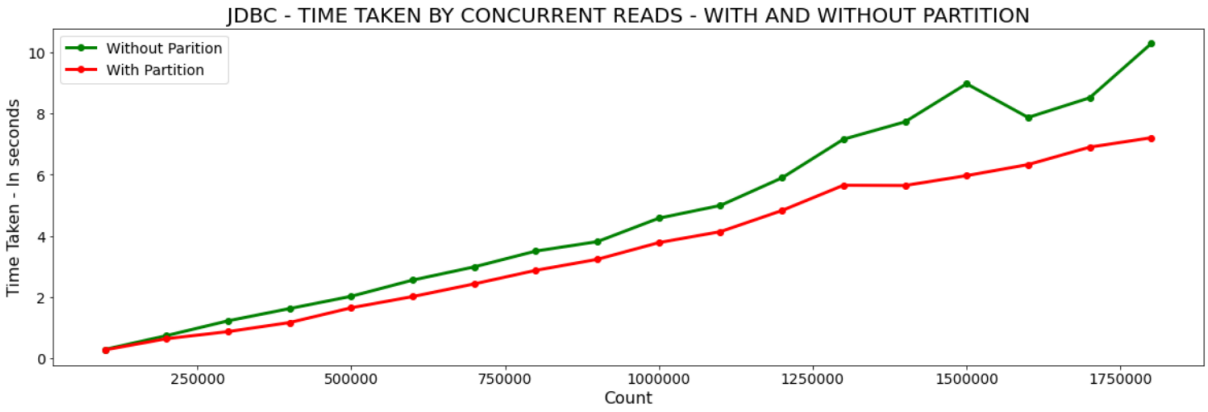


Figure.28 Plot for - JDBC large reads execution time – Non partitioned vs Partitioned instance

8.1.2. Large reads Using ORM

The reads of high-volume data with respect to the ORM approach are explained in this section. Like the process described in JDBC larger reads, one more set of iterative read operation is performed, now with the connection being ORM. To achieve a bias-free conclusion, the same parameters used in JDBC are used here as well.

The same age range of 18 to 49 years is considered. However, unlike JDBC, ORM uses SQLAlchemy to retrieve data from a database. Unlike JDBC, where a cursor is used to read the data by directly establishing a connection with the database each time a read is required. ORM works in a completely different manner. Here , the data fetched can be mapped using its metadata. The metadata of the MySQL table will be replicated into the respective target system which can hold a dataset, for instance it can be a pandas dataframe or spark dataframe.

This leads to the question of why iterative reads is performed when direct mapping is possible in this project? The reason is, in this case the data is large but not exponentially. Direct mapping of all the records into target can work best in case of limited record size. Suppose for

instance, if the database table has 1 billion records, mapping them directly will increase the digital footprint exponentially. This, in turn, will be a complete waste of resources that could have been used for other necessary tasks. As a result, instead of reading from the target after direct mapping, the approach is with specified reads.

Figure.29 shows that the execution time increases as the number of records increases. Furthermore, fetching records from the partitioned instance takes less time than from the non-partitioned one.

Count	Time Taken - WO Partition	Time Taken - W Partition
100000	0.542045	0.505698
200000	0.962232	0.964312
300000	1.850563	1.263844
400000	2.500932	1.503446
500000	3.031992	1.997281
600000	4.871644	2.108282
700000	4.662550	3.350403
800000	4.582515	3.672059
900000	4.664297	3.839507
1000000	5.012995	4.874027
1100000	5.279776	5.131148
1200000	5.592312	5.603186
1300000	8.314085	6.464975
1400000	8.534524	7.361800
1500000	8.497806	6.943384
1600000	9.076284	7.476461
1700000	8.615573	8.005115
1800000	8.761989	8.546330

Figure.29 ORM large reads execution time – Non partitioned vs Partitioned instance

Figure.30 depicts the visualization of the table showcased on Figure.29. In ORM as well, we can see that the query ran on table with partition takes lesser time to execute, than the other.

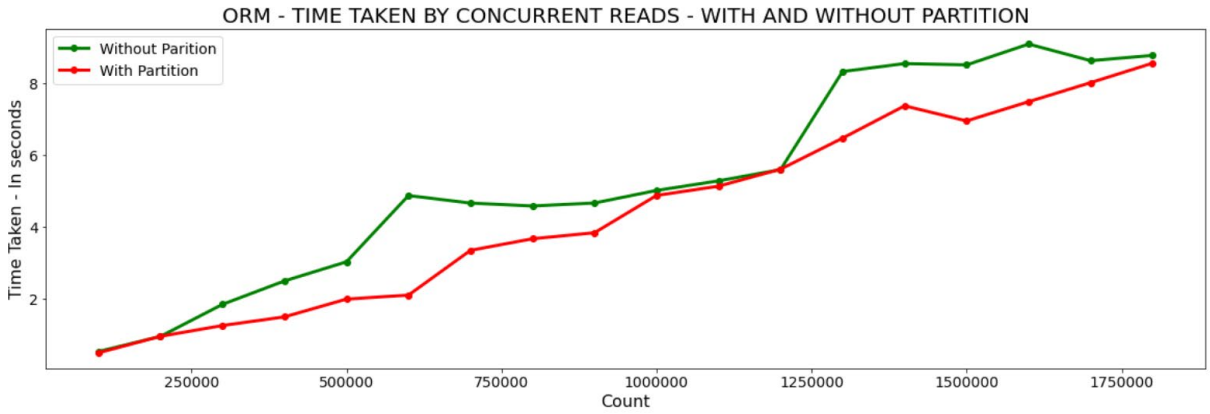


Figure.30 Plot for - ORM large reads execution time – Non partitioned vs Partitioned instance

8.1.3. Comparison of JDBC vs ORM – Large Reads

Figure.31 depicts the comparison of time taken by all the 4 scenarios JDBC and ORM) to fetch a certain no of records.

Count	JDBC Time Taken - WO Partition	JDBC Time Taken - W Partition	ORM Time Taken - W0 Partition	ORM Time Taken - W Partition
100000	0.280434	0.266755	0.542045	0.505698
200000	0.732781	0.631928	0.962232	0.964312
300000	1.217474	0.863831	1.850563	1.263844
400000	1.618266	1.160356	2.500932	1.503446
500000	2.019552	1.643749	3.031992	1.997281
600000	2.551742	2.010968	4.871644	2.108282
700000	2.982881	2.426870	4.662550	3.350403
800000	3.501579	2.870222	4.582515	3.672059
900000	3.809089	3.230571	4.664297	3.839507
1000000	4.578667	3.778391	5.012995	4.874027
1100000	4.994658	4.136639	5.279776	5.131148
1200000	5.898068	4.829008	5.592312	5.603186
1300000	7.162713	5.656934	8.314085	6.464975
1400000	7.732898	5.648630	8.534524	7.361800
1500000	8.979501	5.971123	8.497806	6.943384
1600000	7.875977	6.333595	9.076284	7.476461
1700000	8.519996	6.904070	8.615573	8.005115
1800000	10.294916	7.211643	8.761989	8.546330

Figure.31 Time taken to fetch records – ORM and JDBC combined

In order to attain high scalability, it is required to optimistically enhance the database to handle reads. As from the explored scenarios, there is a stark difference between the time taken by JDBC connection to fetch data from a partitioned table , when compared with the other scenarios.

Figure.32 depicts the visualization of this difference through a line plot. The red line which is the least time consuming one is “JDBC connection to MySQL table without partitions”.

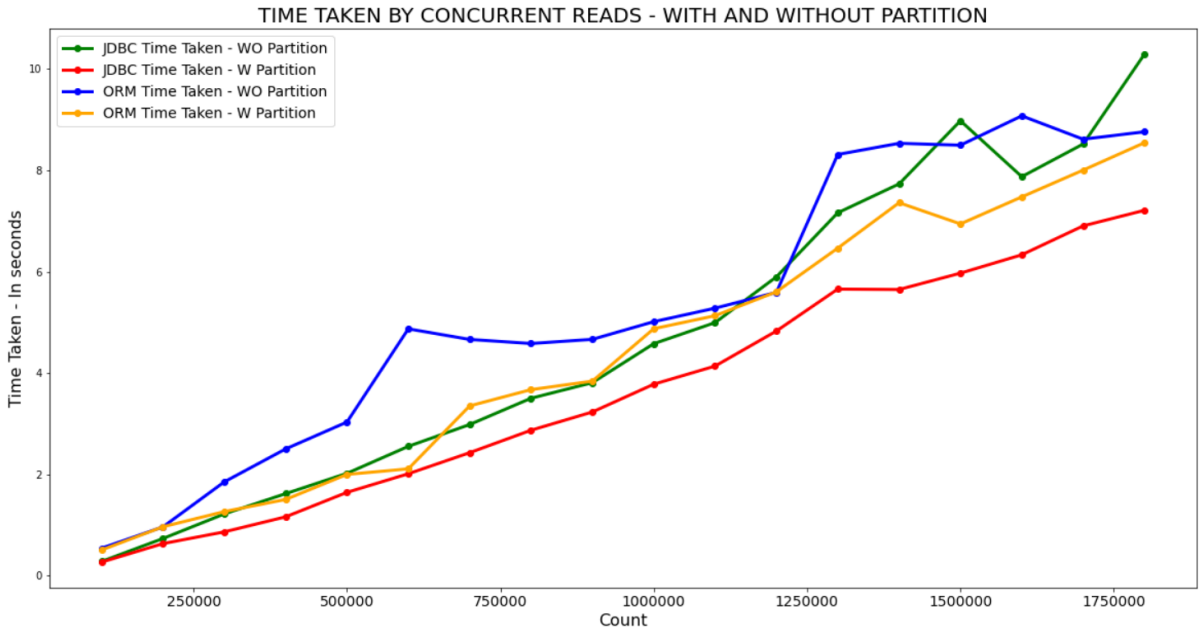


Figure.32 Time taken to fetch records – ORM and JDBC combined

The dataset originally picked has 3.4 million records, and the scenario tested scales up to 1.8 million records starting from 100,000. This test can also be replicated by increasing the record count limit up to the maximum capacity. The partitioning logic that was used while creating the MySQL table with partitions, can also be changed into different possible combination, and tested. But the undeniable insight attained is that the JDBC connection fetches data faster than ORM. Figure.33 Depicts the bar graph representation of the same is a line plot.

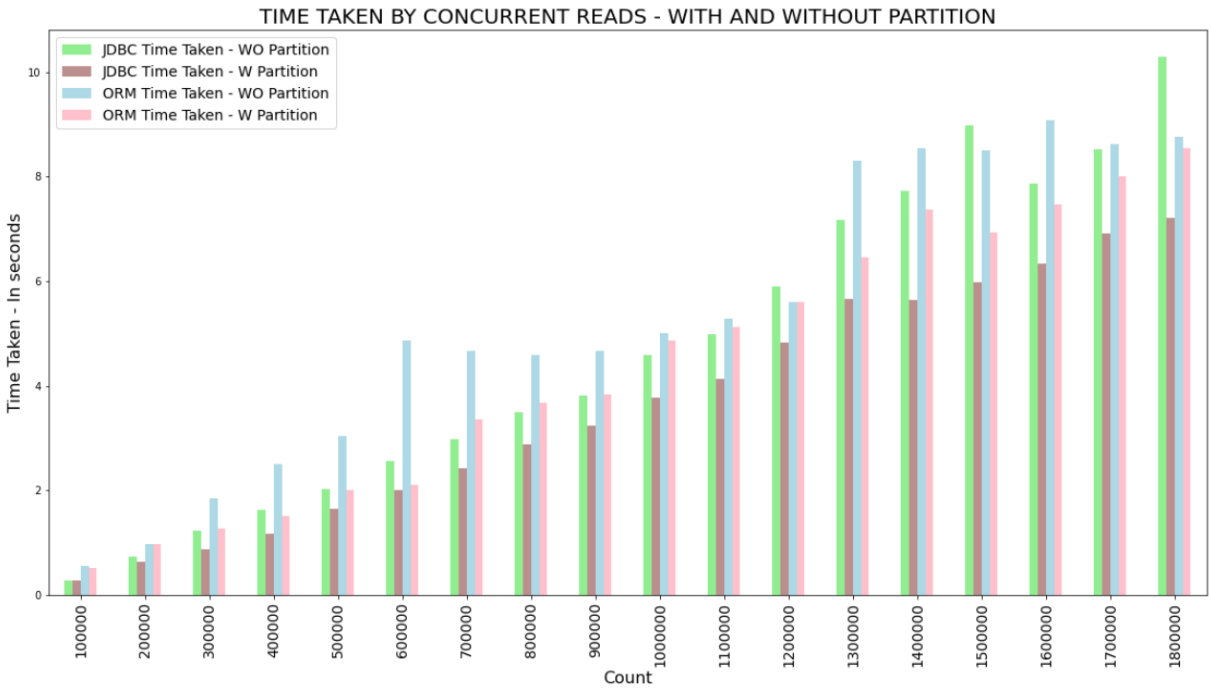


Figure.33 Time taken to fetch records – ORM and JDBC combined

8.2. Small Reads

In order to cover all bases of reads, the small reads operations are performed following the larger reads. Smaller reads are explained in detail in the following section. When it comes to small reads, the number of records requested is smaller as the name suggests. Smaller reads come in handy when trying to pick records based on specific conditions.

This type of read operation is required when analyzing a result for only a limited and specific set of values. Consider, in relation to the covid dataset used in this model, if a specific state county wants to find a pattern of hospitalization of senior citizens over the course of a year. In this case, the number of senior citizens could number in the thousands. As a result, to better analyze the results, the state county can only study the pattern of a few hundred people rather than the entire population, and in this case, a small reads help.

But, if the large reads approach has been successfully tested and a best solution has been found, will that not cover the scenarios for small reads? The answer is that an ideal solution for

large reads may not always be suitable for small reads. Sometimes it is feasible that, the optimal solution for smaller reads is complete opposite of the ideal solution of larger reads. Choosing what's best must be done after carefully analyzing all these different possible outcomes, taking the factors influencing it also into consideration.

Hence to cover all bases, small read operation based on a partitioned, non-partitioned instance is implemented and tested using both JDBC and an ORM approach. The following sections go over the implementation in greater detail.

8.2.1. Small Reads Using JDBC

As aforementioned, an iterative approach like large read is performed. But unlike large reads, the data limit is exponentially less. The initial run of the iteration starts from fetching 5000 records . Along with each incremental run, the record count is incremented by 5000. This process continues until the maximum limit of 200000 is reached. Since the limit set for smaller reads is 200000 records.

Unlike large reads, where a specific age group is picked, the data is fetched without any filter conditions. The code is given complete reign over the entire dataset and the entire table is queried with a limit as specified above. Each iteration returns the number of records in each limit range for both partitioned and non-partitioned instances. And for each of these, the execution time is recorded.

Figure.34 and 35 depicts a first and last 5 samples of the time difference between the partitioned and the other. In this case, the partitioned one outperforms the unpartitioned one, even though the difference is small.

As discussed previously, the optimal solution for the large read is not always the best possible solution for all scenarios and vice versa. But despite that, its observed here that the difference in time here is nearly negligible due to its smaller difference.

Count	Time Taken - WO Partition	Time Taken - W Partition
5000	0.018293	0.018357
10000	0.020717	0.022139
15000	0.025031	0.028361
20000	0.035914	0.037235
25000	0.043689	0.047004

Figure.34 JDBC small reads (first 5 sample) Time taken by Non partitioned vs Partitioned instance

175000	0.501084	0.541642
180000	0.467553	0.600653
185000	0.461734	0.770973
190000	0.511156	0.638062
195000	0.525138	0.589880

Figure.35 JDBC small reads (last 5 sample) Time taken by Non partitioned vs Partitioned instance

Figure.36 represents the visualization of the outcome of the JDBC Small reads in the line plot format.

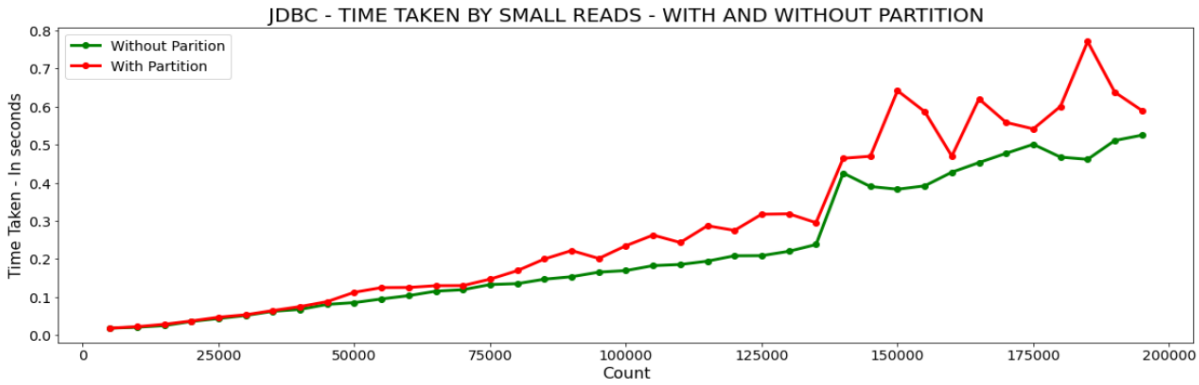


Figure.36 Plot for - JDBC small reads execution time – Non partitioned vs Partitioned instance

This overhead isn't large in comparison with the overhead where partitioning data can reduce reads. But if, the comparison is being done with smaller objects that are queried more frequently, with more varying queries that require building new execution plans – then the overhead of partitioning starts to add up which results in reduced performance.

8.2.2. Small Reads Using ORM

In this case, the same set of operations is performed as in the JDBC approach, but instead of querying the database, the result is obtained by using a mapped object. The methodology used here is like the one performed in large reads for retrieving data from partitioned and non-partitioned tables.

The iterations are set to run with the same exact parameters which is starting with 5000 records and incrementing by 5000 till the maximum record count of 200000 is reached. Since the parameters are same as the ones used in JDBC, the insights attained will be free of bias.

These iterative runs are done with SQLAlchemy embedded in python as done for the larger reads. The total number of iterations executed for the smaller reads is more than what has been done to test large reads.

Figure.37 and 38 depicts a first and last 5 samples of the time difference between the partitioned and the other. With these results, it is possible to observe which one outperforms the other. This is helpful in identifying the ideal strategy that can be used.

As stated before, the optimal solution for the large read is not always the best possible solution for all scenarios and vice versa. The scenarios which will be most frequently used from the business perspective should be taken into consideration when deciding on the optimal one. Sometimes, it is possible that the ideal solution of the test scenarios may incur other challenges when exposed to the outer business requirements. It is always a gamble, but the gamble can turn to a win if all these influencing factors align up perfectly with the suggested solution and also the business requirements.

Count	Time Taken - WO Partition	Time Taken - W Partition
5000	0.012319	0.019670
10000	0.030741	0.039421
15000	0.118356	0.059560
20000	0.056728	0.077650
25000	0.141397	0.092569

Figure.37 ORM small reads (first 5 sample) Time taken by Non partitioned vs Partitioned instance

175000	0.673741	0.722350
180000	0.810526	0.811160
185000	0.722066	0.715858
190000	0.813126	0.797754
195000	0.765715	0.768471

Figure.38 ORM small reads (last 5 sample) Time taken by Non partitioned vs Partitioned instance

Figure 39 demonstrates the above results in a line plot with the green line representing the execution time of data without partition and the red one is the one with partition.

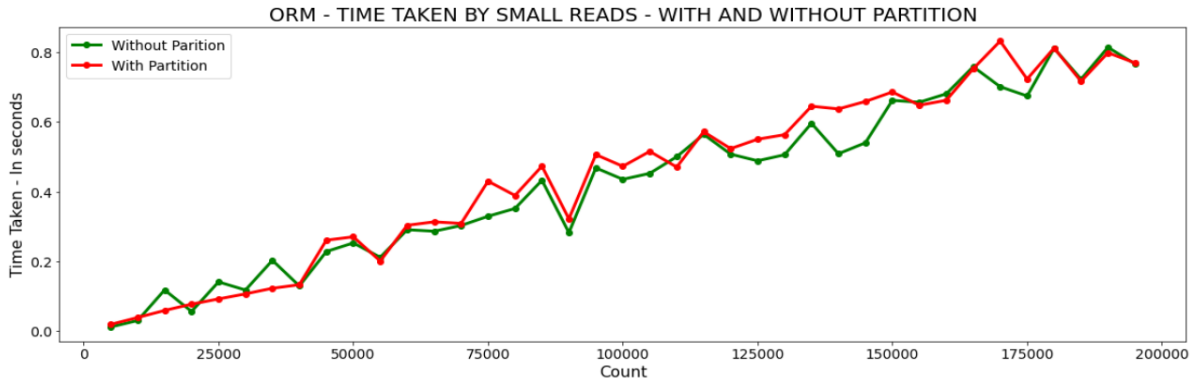


Figure.39 Plot for - ORM small reads execution time – Non partitioned vs Partitioned instance

8.2.3. Comparison of JDBC vs ORM – Small Reads

The results of JDBC and ORM is compared in this section. Insights gained from this comparison is, there is a significant difference. Querying a database with the JDBC driver outperforms the other method. This is because ORM consumes more CPU cycles. In addition to this, ORM fetches the data and stores it into a mapped object in the target. An additional overhead is added because of this.

The number of records in each fetch of the iteration gradually increases, but the JDBC execution time outperforms the ORM one. And these results can be portrayed in layman terms using the visualization options. It is also concluded from these observations that retrieving the result in a environment using the JDBC strategy is preferable. Figure.40 represents the graph of the time taken by each of the approach. The result is also illustrated in Bar graph format which can be observed in Figure.41.

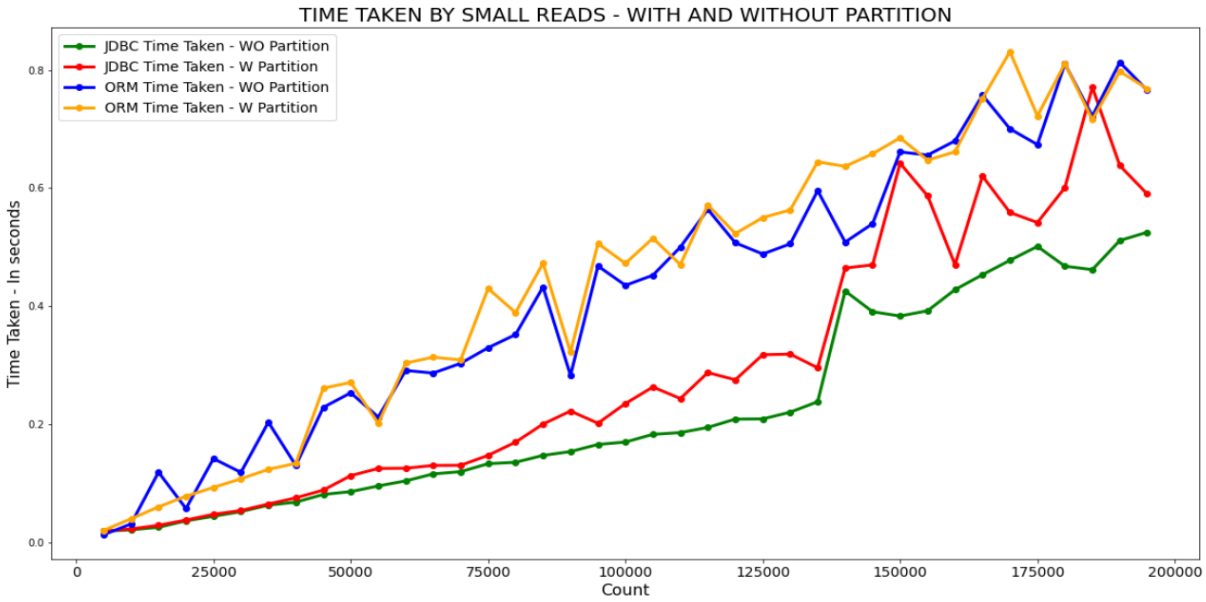


Figure.40 Line plot representing the Small reads comparison between JDBC and ORM

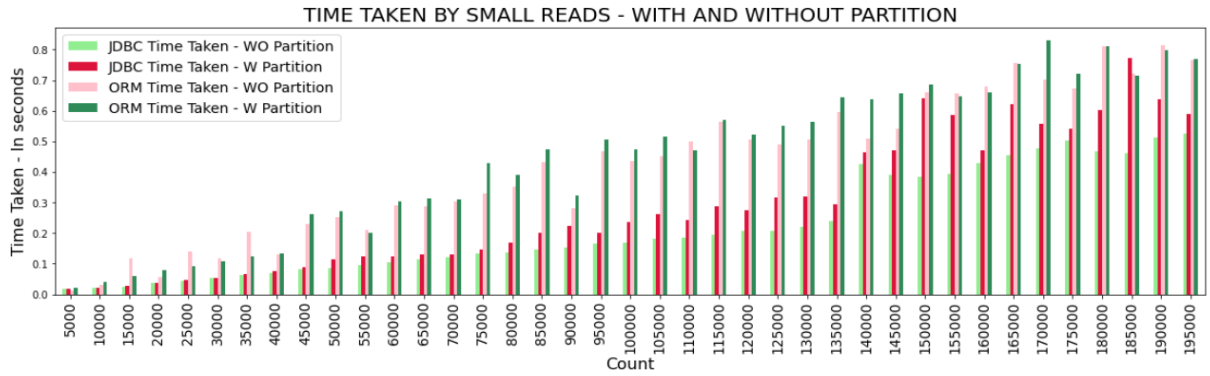


Figure.41 Bar Chart representing the Small reads comparison between JDBC and ORM

Partitioning is a powerful optimization technique, which will help you in improving your query performance. In order to properly utilize this technique, it is recommended that first you analyze your data and properly choose the key columns on which partitioned is to be done, as well as a suitable number of partitions based on volume of your data.

True, ORM is a simpler approach to implementation, but when dealing with a large dataset, it is always better to focus on the model's performance, and thus JDBC with partitioned table is an optimal solution.

9. EXPLORATORY ANALYSIS

Knowledge about the data is required to effectively design the model. Understanding the data involves performing exploratory data analysis, which refers to the discovery of data patterns through preliminary investigations. Assumptions can thus be verified or validated using summary statistics and graphical representation. This is an important step because it allows the factors to be identified and categorized as significant or insignificant in relation to a specific scenario. Furthermore, some data can be linked to others. Additionally, some data can be correlated with others. Assume there is a need to determine whether there is a relationship between age and the rate of hospitalization. If this is the case, identifying this pattern beforehand allows a precautionary step to be taken and any sort of undesirable conditions to be avoided.

To understand the performance of SQL query execution. The time taken by each of these scenarios with respect to the connection type and table type is recorded. Partitioning doesn't always necessarily mean that the query execution time will be lesser compared to non-partitioned tables. It solely depends on how the partitions are utilized while querying. Depending on the business requirements, it can be observed that some columns are more frequently used to generate patterns and attain knowledge, it is always recommended to make use of that frequency factor and partition accordingly. Some of the scenarios tested here doesn't make use of that factor , hence it is observed that some queries run faster in table without partition than in table with partitions. Apart from this, the performance of JDBC vs ORM is also tested, a vast difference in execution time can be noticed here. And as expected, JDBC executes faster in comparison with ORM.

9.1 Scenarios Executed

Various scenarios are executed to perform exploratory analysis. These scenarios results were also plotted in charts which can be found in the respective execution logs. These analyses are performed for both partitioned and non-partitioned instances using both JDBC and the ORM approach.

Table.3 depicts the list of scenarios tested in different connections respective of MySQL table type.

Scenario No	Description	Connection	Table type
1	No of cases recorded per age group	JDBC	My SQL - Table without partition
		JDBC	My SQL - Partitioned table
		ORM	My SQL - Table without partition
		ORM	My SQL - Partitioned table
2	No of deaths based on sex	JDBC	My SQL - Table without partition
		JDBC	My SQL - Partitioned table
		ORM	My SQL - Table without partition
		ORM	My SQL - Partitioned table
3	No of cases per Ethnicity	JDBC	My SQL - Table without partition
		JDBC	My SQL - Partitioned table
		ORM	My SQL - Table without partition
		ORM	My SQL - Partitioned table
4	No of positive hospitalized cases	JDBC	My SQL - Table without partition
		JDBC	My SQL - Partitioned table
		ORM	My SQL - Table without partition
		ORM	My SQL - Partitioned table
5	Number of cases and respective current_status of alive people	JDBC	My SQL - Table without partition
		JDBC	My SQL - Partitioned table
		ORM	My SQL - Table without partition
		ORM	My SQL - Partitioned table

Table.3 Scenarios executed for Exploratory Analysis

Figure.42 depicts the time taken by each of the above listed scenario to execute with different connection and table type.

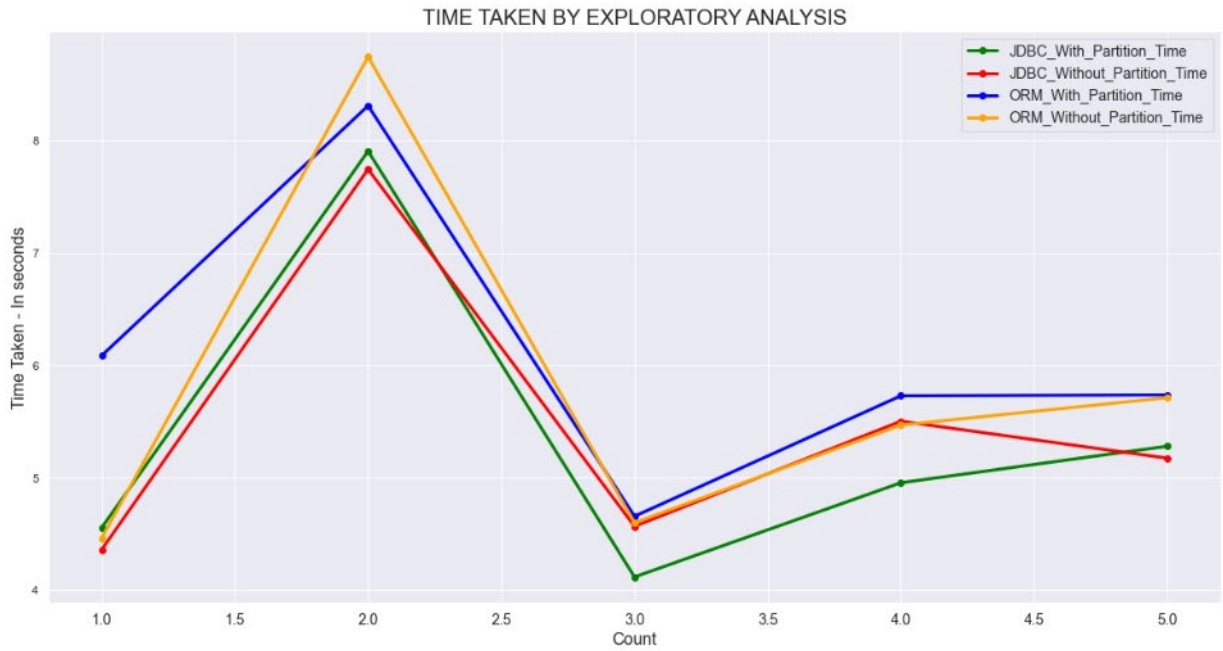


Figure.42 Line chart representing the execution time taken by the scenarios

10. CONCURRENT BURST READS

Concurrent burst reads can be defined as the task of running multiple queries parallelly at the same time. The impact to the execution time of the queries while performing concurrent reads is monitored in this section and the insights derived from this can be helpful in understanding how this impacts the performance.

To test this scenario, a bash file is created with 10 different python files execution listed inside it. When this bash file is executed, it in turn executes all the python files listed inside it and it all starts parallelly. The start time and end time of the connection to SQL inside each of the file is recorded and printed as output. Upon analyzing the output we shall see that, they are run parallelly.

Figure 43 depicts the bash script that is used to execute the python files parallelly and figure 44 depicts the sample python file. Figure 45 depicts the execution results.

```
#!/bin/bash
python3 script1.py &
python3 script2.py &
python3 script3.py &
python3 script4.py &
python3 script5.py &
python3 script6.py &
python3 script7.py &
python3 script8.py &
python3 script9.py &
python3 script10.py &
```

Figure.43 bash script

jupyter script2.py a few seconds ago

File Edit View Language

```python
import time
import mysql.connector
from getpass import getpass
from mysql.connector import connect, Error
from getpass import getpass
from datetime import datetime

start_dt = datetime.now()
start_ts=datetime.timestamp(start_dt)
try:
    with connect(host="lab-assignment-225.cibzfcia066j.us-east-1.rds.amazonaws.com",
    user='admin',password='welcome123',database="group_project"
    ) as connection:
        create_states_table_query = "Select count(*) from (select * from cdc_covid_data LIMIT 10000) as t1"
        with connection.cursor() as cursor:
            cursor.execute(create_states_table_query)
            result = cursor.fetchall()
except Error as e:
    print(e)
end_dt=datetime.now()
end_ts=datetime.timestamp(end_dt)
res = end_dt - start_dt
print('Start time: ',start_dt,' End time: ',end_dt,' Time taken: ',format(res))
```

Figure.44 sample python file

```
Start time:  2022-05-15 23:10:53.139423  End time:  2022-05-15 23:10:54.126189  Time taken:  0:00:00.986766
Start time:  2022-05-15 23:10:53.122627  End time:  2022-05-15 23:10:54.170836  Time taken:  0:00:01.048209
Start time:  2022-05-15 23:10:53.156755  End time:  2022-05-15 23:10:54.171865  Time taken:  0:00:01.015110
Start time:  2022-05-15 23:10:53.092700  End time:  2022-05-15 23:10:54.377878  Time taken:  0:00:01.285178
Start time:  2022-05-15 23:10:53.180146  End time:  2022-05-15 23:10:54.385667  Time taken:  0:00:01.205521
Start time:  2022-05-15 23:10:53.144343  End time:  2022-05-15 23:10:54.395335  Time taken:  0:00:01.250992
Start time:  2022-05-15 23:10:53.208036  End time:  2022-05-15 23:10:54.399458  Time taken:  0:00:01.191422
Start time:  2022-05-15 23:10:53.227479  End time:  2022-05-15 23:10:54.406867  Time taken:  0:00:01.179388
Start time:  2022-05-15 23:10:53.154524  End time:  2022-05-15 23:10:54.410284  Time taken:  0:00:01.255760
Start time:  2022-05-15 23:10:53.217215  End time:  2022-05-15 23:10:54.411417  Time taken:  0:00:01.194202
```

Figure.45 execution results

11. DATA LOCALITY

When it comes to building a robust model, an important factor to consider is maintaining a balanced throughput by designing an effective infrastructure. A properly balanced infrastructure ensures an improved throughput. And for building this there is a need to avoid any kind of slow functioning components which can hamper the system. If the data is stored in multiple Hosts. If in any case the virtual machine (VM) of one node migrates to another one within the same cluster, then there is a chance of performance drop. It is because a VM uses the resources associated with the current host. In this scenario, it can still use the remote host for processing. This adds up network traffic. Fabric interconnections may be slower than storage. In this way, the impacted performance of the VM can affect the whole system's efficiency. Therefore, data locality strategy is used where the actual data resides near to the computation. The approaches implemented for data locality are described below.

11. 1. InnoDB Clusters

For the design of this model InnoDB cluster approach is implemented in the host machine. Using MySQL shell, the administration of at least three server instances to function as a cluster is possible. Each server instance runs MySQL group replication, and this provides a mechanism for data replication within the cluster and built-in automatic failover feature. MySQL router is setup as a proxy to hide multiple instances behind a single TCP port. The router helps any applications to connect to a cluster member without any code change.

For this model a InnoDB cluster named as 'covidCluster' is created with three sandbox instances. It needs three instances to support failover mechanism. Figure 46 demonstrates the initialization and creation of the of the cluster. One of the three created server instance is selected

as a seed instance during initialization. This instance holds the initial state of the database and when other instances are added this state is replicated to those instances. Creating the cluster does some of the key actions such as deploying the metadata schema, starting the group replication after verifying, validating the configuration for it, registering the seed instance, and creating the necessary administrative accounts.

In the next step, the other two created instances are added to the cluster. At this stage both do not have any data. They need to sync with the current state of the seed instance. Figure 47 represents the state of the cluster after adding those two instances. The status of each instance can be ONLINE or RECOVERING. A RECOVERING status shows that currently it is receiving updates and it switches to ONLINE once it is complete. This ensures data consistency.

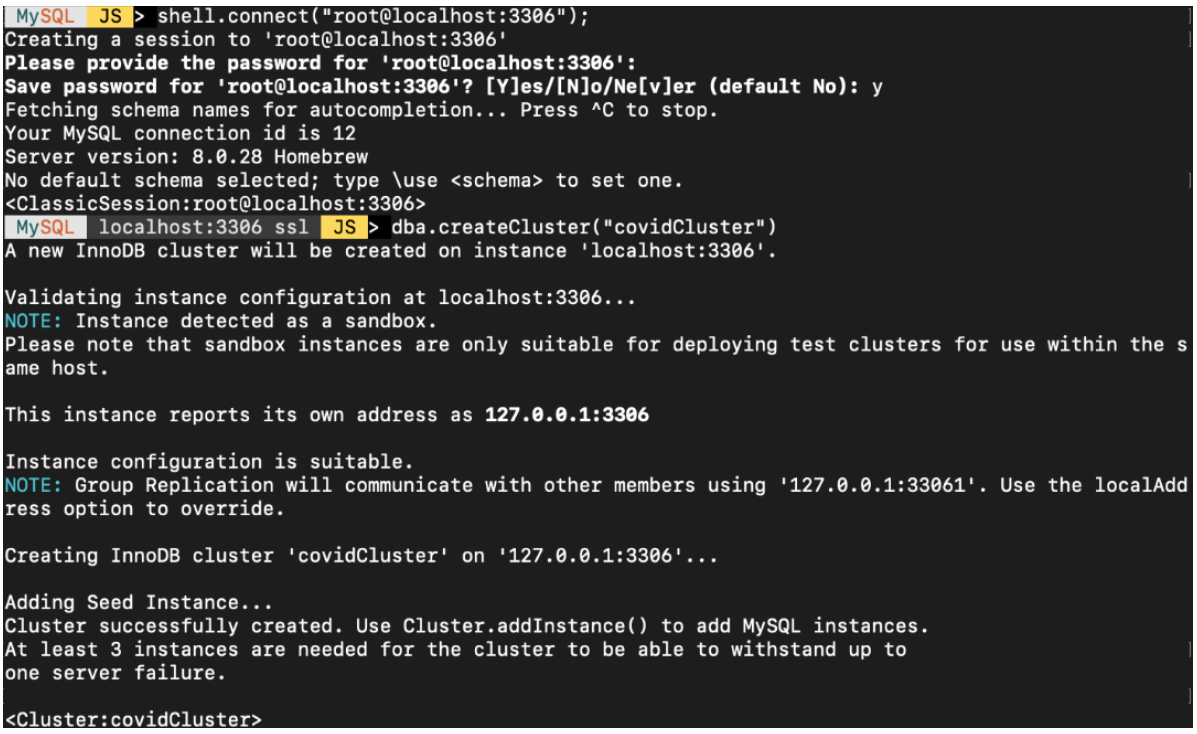


Figure 46. InnoDB cluster initialization and creation

Figure.47 Cluster status

Figure.48 portrays the deployment of MySQL router which can handle failover, where as Figure 49 shows how the connection is established between the client and the router port. It connects to the cluster and configures itself for use after fetching metadata. There are two session created by the configuration :

- one which redirects connections to the Primary instance and it is knowna as a read-write session.
- other one is a read session, redirecting connections in a round-robin fashion to one of the secondary instances.

```
(base) deepss-MacBook-Air:~ deeps$ mysqlrouter --bootstrap localhost:3306 --directory covidrouter
Please enter MySQL password for root:
# Bootstrapping MySQL Router instance at '/Users/deeps/covidrouter'...

- Creating account(s) (only those that are needed, if any)
- Verifying account (using it to run SQL queries that would be run by Router)
- Storing account in keyring
- Adjusting permissions of generated files
- Creating configuration /Users/deeps/covidrouter/mysqlrouter.conf

# MySQL Router configured for the InnoDB Cluster 'covidCluster'

After this MySQL Router has been started with the generated configuration

    $ mysqlrouter -c /Users/deeps/covidrouter/mysqlrouter.conf

InnoDB Cluster 'covidCluster' can be reached by connecting to:

## MySQL Classic protocol

- Read/Write Connections: localhost:6446
- Read/Only Connections:  localhost:6447

## MySQL X protocol

- Read/Write Connections: localhost:6448
- Read/Only Connections:  localhost:6449
```

Figure.48 MySQL router deployment

```
MySQL  localhost:6446 ssl  SQL > select @@port;
+--------+
| @@port |
+--------+
|   3306 |
+--------+
1 row in set (0.0018 sec)
```

Figure.49 MySQL router connection

The next approach that has been explored on for the locality part is, using AWS. This can be discussed down the road.

11.2 AWS RDS

Data storage can be done in a local host or in a cloud environment. AWS RDS(Relational Database Service) supports relational database models to store and organize data in an efficient manner. It assists in data managemnet task such as backup,recovery and data migration. RDS is a service which is used to manage relational databaes. Pre-congfigured parameters are provided for the selected engine by RDS databae instances. Multiple database instances can be managed by a single service.

Based on the features provided by AWS RDS the data used in this model is hosted in a cloud environment using this service. A connection between the relational database MySQL and the created cloud service is established. Then, the data is uploaded into MySQL database by using the approach as mentioned in section 7. The same data is stored in the locahost. And, it is observed that data fetched from the service takes more time to execute than localhost. High latency always hampers the efficiency of a model. But there is an other factor that also needs to be considered in this. Data present in the local host, can be accessed and manipulated by that specific machine only. Where are, the data in the AWS RDS can be accessed from anywhere as long as the connection details are provided.

Figure 50 depicts the AWS Connection that has been created .

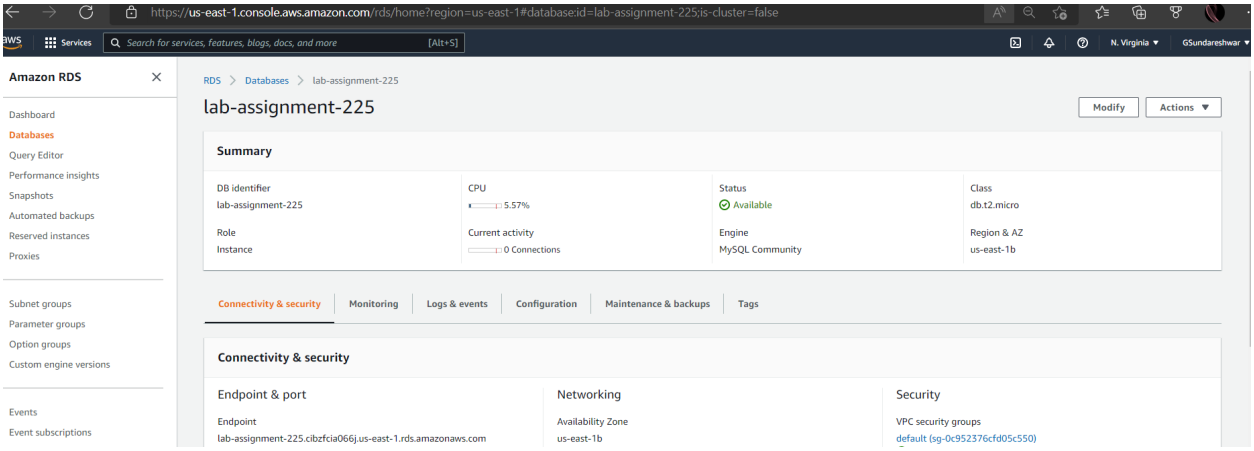


Figure.50 AWS DB connection

12. MEMORY PRESSURE

The performance of a database server depends on how much memory is allocated to the server and how well it is utilized. Memory plays a significant resource for speed and efficiency when handling concurrent transactions and running big queries. Each thread in MySQL shares the same base memory and it demands memory which is used to manage the client connections. When a thread is no longer needed, the memory allocated to it is released and returned to the system unless the thread goes back into the thread cache. In that case, the memory remains allocated.

In most cases, the memory-specific variables set for configuration are targeted on a storage-based specific configuration such as MyISAM or InnoDB. When a mysqld instance spawns within the host system, MySQL allocates buffers and caches to improve performance of database operations based on the set values set on a specific configuration. There are numerous simple by crucial steps mandatory in reducing the memory pressure. Some of them are listed below

12.1 Techniques to reduce Memory Pressure

12.1.1 Schema Design

Memory pressure can be significantly reduced while creating the table itself. Since fetching the data utilized a lot of memory, it is always advisable to start refining the data from table and schema creation itself, so that all the overhead of blank spaces and unnecessary values can be handled before the querying phase. There are some key points to be kept in mind while creating the schema. Some of them are listed below:

- Using the right datatype plays a major part in reducing the memory pressure, for example, using Smallint instead of Bigint as type for columns which will handle smaller integer numbers. Another example could be DATETIME vs TIMESTAMP.

- Adding NOT NULL to mandatory columns will reduce the memory usage, because even the NULL value requires 1 or 2 bytes of storage, so it is advisable to optimize it by adding NOT NULL to the records.
- When defining varchar type variables , it is sensible to assign the length of the variable in such as way that it matches the maximum length required for that field. In this model for instance, there is a field which is related to the residential state of a patient . This column contains the state code represented with 2 characters. Assigning the length of varchar variable according to the length of the data is more sensible than having the length specified as 100 or 1000 for this specific column. This way, all the empty memory allocated for this column (which is the additional 8 bytes assigned) can be utilized in other required spaces.
- Floating-point types typically use less space than DECIMAL to store the same range of values. A FLOAT column uses four bytes of storage. DOUBLE consumes eight bytes and has greater precision and a larger range of values than FLOAT. As with integers, you're choosing only the storage type; MySQL uses DOUBLE for its internal calculations on floating-point types.

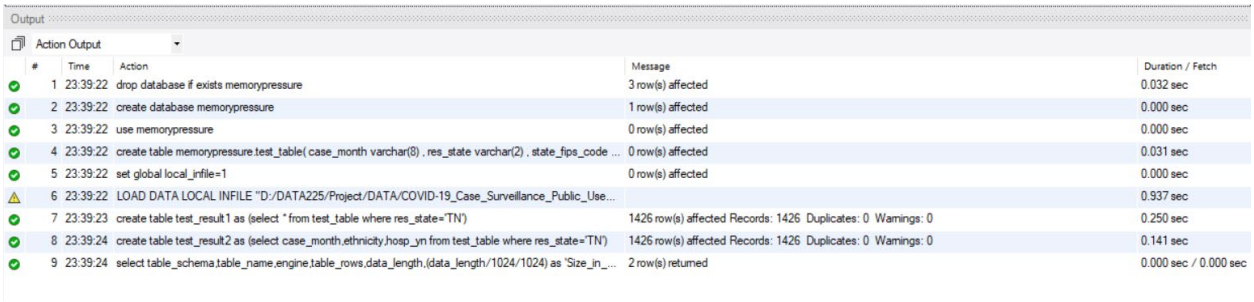
12.1.2 Optimizing the Query:

Every field accounts for some space and when select * commands are used, all the columns are fetched as result whether we use it or not. This means that the size of the resultant set will be directly proportional to the number of columns. In Order to reduce the digital footprint or relieve some memory usage, it is always recommended to pick only whats necessary.

For instance, the dataset used in this project has 19 columns and 3.4 million records. If the requirement is to list the cases of a specific ethnicity belonging to a specific region to determine

the hospitalization status, it is not sensible to fetch all the columns. It makes perfect sense to fetch only the 4 listed columns since those are enough to obtain the required insight. Selecting the specific columns means, the system is freed from scanning all the other unnecessary rows . This in turn will significantly reduce the memory pressure.

It might not seem significant in case of smaller database with lesser columns. Whereas when the size of dataset is larger (as in, GB,TB), selecting the specific columns plays a crucial part in reducing the memory pressure



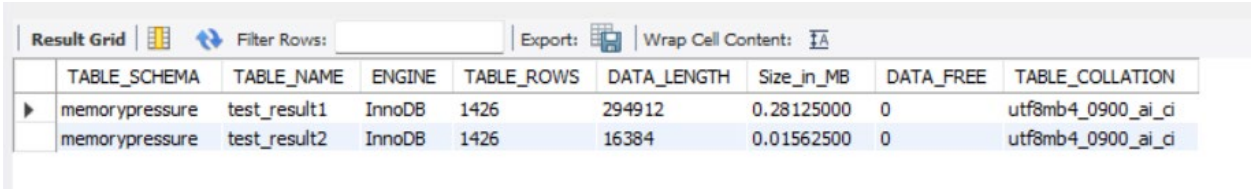
Output

Action Output

#	Time	Action	Message	Duration / Fetch
1	23:39:22	drop database if exists memorypressure	3 row(s) affected	0.032 sec
2	23:39:22	create database memorypressure	1 row(s) affected	0.000 sec
3	23:39:22	use memorypressure	0 row(s) affected	0.000 sec
4	23:39:22	create table memorypressure.test_table(case_month varchar(8) , res_state varchar(2) , state_fips_code ...	0 row(s) affected	0.031 sec
5	23:39:22	set global local_infile=1	0 row(s) affected	0.000 sec
6	23:39:22	LOAD DATA LOCAL INFILE "D:/DATA225/Project/DATA/COVID-19_Case_Surveillance_Public_Use...		0.937 sec
7	23:39:23	create table test_result1 as (select * from test_table where res_state='TN')	1426 row(s) affected Records: 1426 Duplicates: 0 Warnings: 0	0.250 sec
8	23:39:24	create table test_result2 as (select case_month,ethnicity,hosp_yn from test_table where res_state='TN')	1426 row(s) affected Records: 1426 Duplicates: 0 Warnings: 0	0.141 sec
9	23:39:24	select table_schema,table_name,engine,table_rows,data_length,(data_length/1024/1024) as 'Size_in_...	2 row(s) returned	0.000 sec / 0.000 sec

Figure.51 Action output of memory pressure test

Figure 51 depicts action output of the operations performed to test the memory usage difference between the select query with specific columns vs all the columns.



Result Grid | Filter Rows: | Export: | Wrap Cell Content:

TABLE_SCHEMA	TABLE_NAME	ENGINE	TABLE_ROWS	DATA_LENGTH	Size_in_MB	DATA_FREE	TABLE_COLLATION
memorypressure	test_result1	InnoDB	1426	294912	0.28125000	0	utf8mb4_0900_ai_ci
memorypressure	test_result2	InnoDB	1426	16384	0.01562500	0	utf8mb4_0900_ai_ci

Figure.52 Difference in Size recorded

Figure 52 depicts the difference in memory between the two different select queries. We shall see that the size is exponentially less when specific columns are picked rather than the entire list of columns. This might be a simple but crucial way to reduce the memory pressure.

12.1.3 Handling the Historical Data:

The other way to reduce memory usage while retrieving query results is archival of data. Suppose lets say, there exists a system where historical records are stored, the updates happening to any of the data is stored as a new entry with the current flag set to 1. Some systems may make use of the older versions, where as some others might not. When querying the results using the id, the overhead of the query results tend to be higher because of the different version of records stored in it.

So it is recommended that, when the requirement is only the current record, the query should be optimized in the way, that only the records with current flag set to 1 is picked. The length of the resultant set tend to be way lesser compared to when all the versions are fetched.

If there is no frequent use of this historical versions, a trigger shall be created which will move the older version to a new table with the engine set to Archive and store only the current version in the main table. The historical records can be accessed from the archive table when necessary.

13. CAPACITY

Database implementations are not static. It is always growing in response to demand. Once deployed, there is always incoming or outgoing data traffic where querying, loading, updating, and all other database operations take place. As a result, capacity planning becomes an integral part of database design. Capacity planning is the process of analyzing a system's storage requirements. As a result, the resources can be adjusted as needed. While planning for this, both the hardware and software infrastructures are considered. Measuring these aids in forecasting the need for additional storage. This can be accomplished in a variety of ways, such as by adding more disk devices or expanding the computing infrastructure.

The data stored in this model has a minimum of more than 3 million records for one specific month which was picked. This data grows exponentially every month. As a result, when there is an update each month, millions of new records will enter the database. In this scenario, long-term storage on a single server becomes inconvenient. And the read and write operations will have an impact on the system's performance as well.

A cluster-based approach is used to deal with this. Database clustering occurs when more than one server or instance connects to a single database. Because one server cannot handle the volume of data, another server or node is added to assist with load balancing, data redundancy, high availability, or scalability. Below mentioned are some features provided by database clustering.

- Data Redundancy: In terms of data redundancy, all connected nodes have the same set of data, allowing for data redundancy. Although data redundancy is not ideal, it is advantageous in the case of clustering because data is synchronized among other nodes assures availability of data.

- Load balancing: This is the process of distributing workload across multiple instances connected to a cluster. If there is an increase in traffic in any scenario, the performance impact can be minimized when other nodes or instances can serve the new incoming requests.
- High availability: The availability of a database is determined by the number of transactions or analytics performed on it. If the main server fails or malfunctions, other data nodes can store the data and continue to provide service. In such way, end users receive consistent data as the data is synchronized.

13.1 NDB cluster:

NDB cluster, where NDB stands for Network Database, is the distributed database system underlying MySQL Cluster. It is an in-memory storage engine which offers high availability and data persistence features. The cluster is configured independently of the MySQL server. Some of the key features of MySQL Cluster are a scalability, ACID-compliant transactions, autosharding, online operations and 99.999% availability. Data is partitioned across multiple nodes, and each can accept write operations. There are three types of nodes available.

- Management node: It manages other nodes within a cluster provides configuration of data and starting and stopping a node.
- Data node: It stores cluster data. And it depends on the number of fragment replicas and fragments.
- SQL node: It access the cluster data. It is a specialized type of node. This node designates any application which accesses the cluster data.

NBD Clusters were connected using DigitalOcean Droplets which are **Linux-based virtual machines** (VMs) that run on top of virtualized hardware. Each Droplet that is created is a new server you can use, either standalone or as part of a larger, cloud-based infrastructure

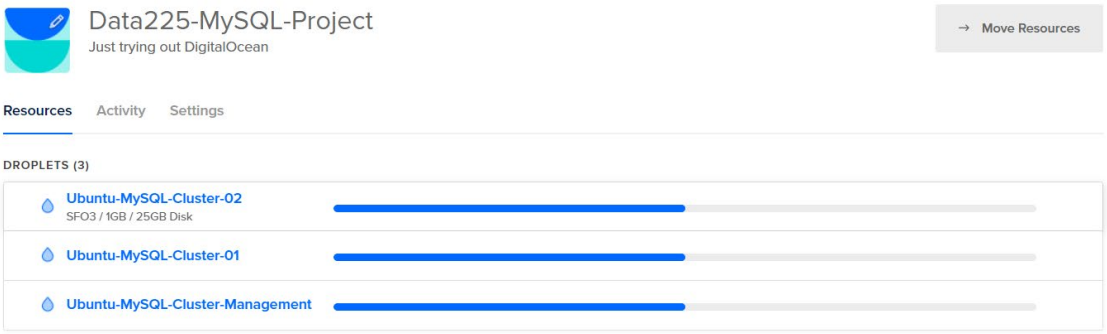


Figure.53 Clusters created

Digital ocean is a paid software with different plans and different number of clusters offered per plan. The one chosen here is a standard plan that limits to a maximum of 75 clusters. Figure53 depicts the three servers that were created.

SQL can be configured in each of those and one can be assigned as the master.

Figure 54 Cluster 1 bandwidth status

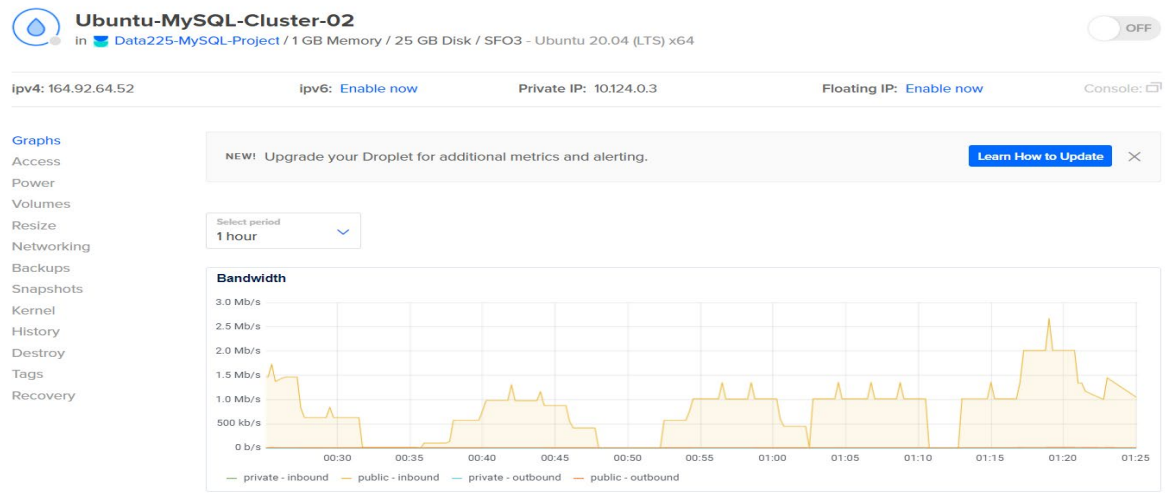


Figure.55 Cluster 2 Bandwidth

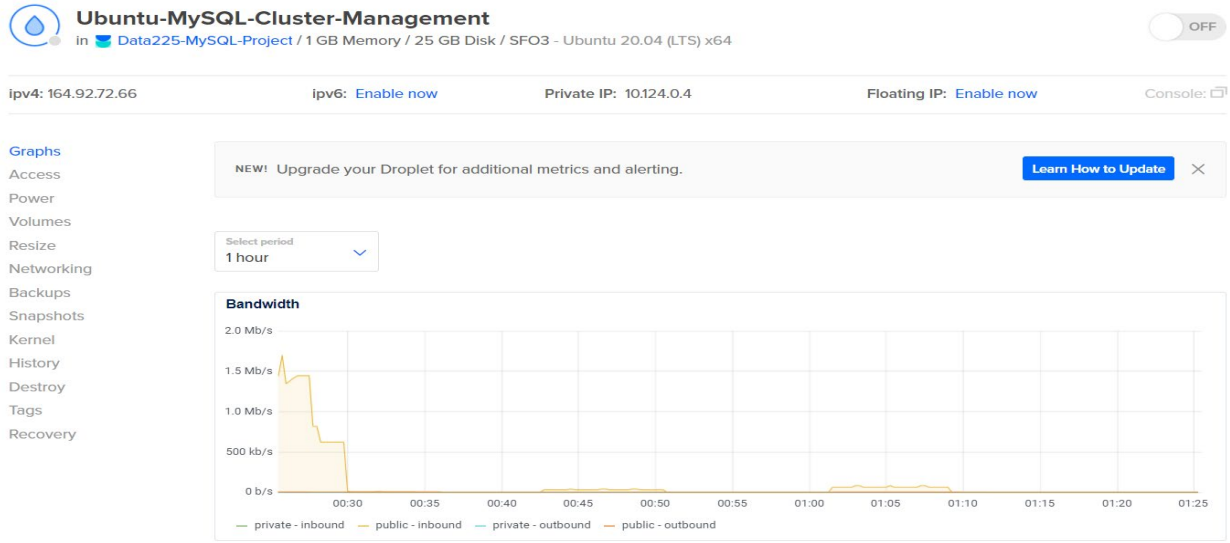


Figure.56 Cluster Management Bandwidth

SQL Can be installed in these servers and using NBD Cluster management different nodes can be created.

The status of the cluster droplets created in digitalocean is depicts in the figures 54,55 and 56. Whereas, Figure 57,58 and 59 depicts the IP address and configurations of the servers created.

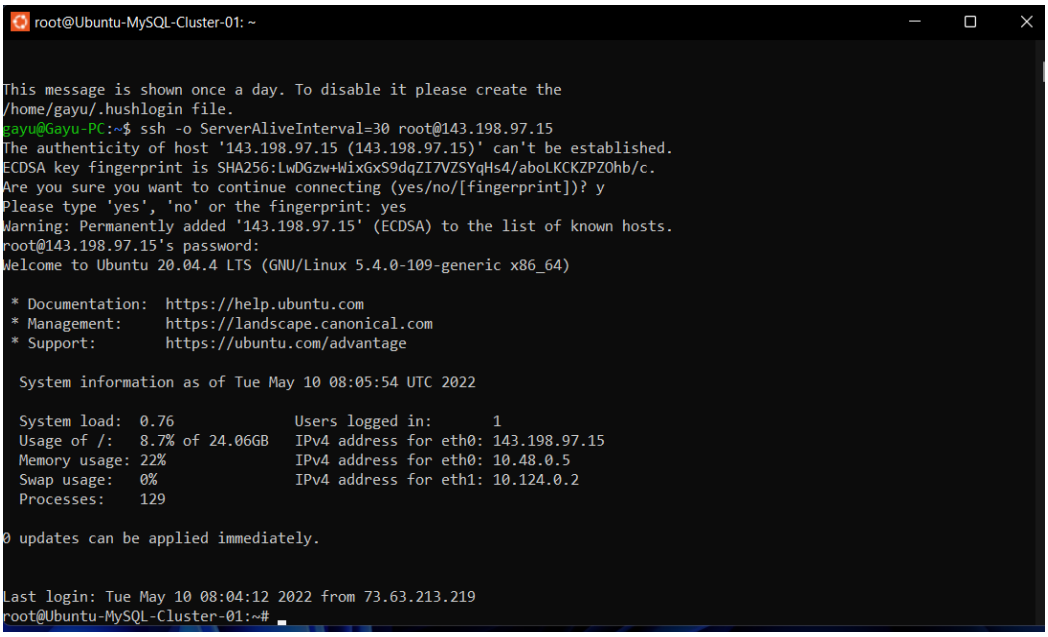


Figure.57 Cluster 1 bandwidth status

Figure.58 Cluster 2 bandwidth status

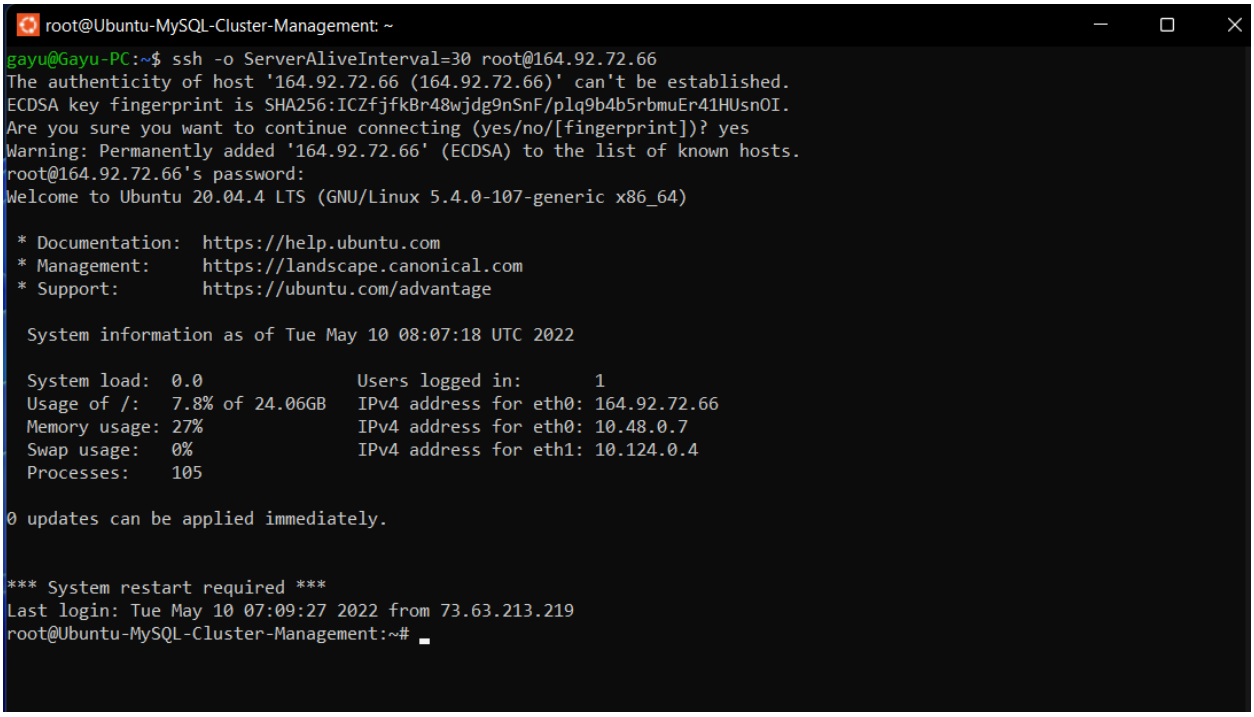


Figure 59 Cluster Management bandwidth status

14. FAILURE, BACKUP AND RECOVERY

What is the one of the most essential parts of an efficient database model? There exist many factors which can be essential in a way, but one that is universally accepted as a crucial part is Failure and Recovery. The ability of the system to recover from crashes with minimal or zero impact is vital.

For instance, when a transaction is submitted to a DBMS for execution, it is essential for the system to ensure that all the operations which need to be in performed in the transaction have completed successfully and the changes must be committed to the database. In case of any interruptions, the data should be remaining same as how it was before the transaction happened. Any changes made must be rolled back to original form. This ensures the data is consistent and intermediate results doesn't affect the original data until completion. This is explained as the ACID property in relational databases.

14.1. Failure

Failures are bound to happen often. And there can be many ways on how it happens. Some of the types are listed below,

1. Transaction failure
2. System failure
3. Media failure and so on.

Let us try to understand the different types of failures that may occur during the transaction.

1. System crash:

   A hardware, software or network error occurs during the execution of the transaction.

2. System error:

   Some operation that is performed during the transaction because of erroneous parameter values or because of a logical programming error.

3. Local error:

   This basically happens due to certain conditions that may lead to cancellation of the transaction.

4. Disk failure:

   This type of failure basically occurs due to data lose in the disk due to a read or write malfunction or because of a disk read/write head crash.

5. Catastrophic error:

   These include the failures caused due to power failure or air-conditioning failure, fire, theft sabotage overwriting disk or tapes by mistake and mounting of the wrong tape by the operator.

14.2. Backup

Failure can happen anytime as it is unpredictable. It is always necessary to have a system in place to ensure that there is no data loss in case of such failures. A periodical or timely backup is necessary so that even if there occurs any serious failure, the loss of data can be minimal or in fortunate cases none. This guarantees the consistency and integrity of the stored data. Furthermore, this also ensures that the data is secure.

Since the dataset used here in this model is related to a public health crisis, it is mandatory to protect the data without any significant loss. As the data gets updated in a timely manner, a scheduled backup system must be kept in place.

Various options are considered before deciding on a best possible solution. Some of these are listed below,

14.2.1 SQLBackupAndFTP Tool

SQLBackupAndFTP is a software that backups SQL Server, MySQL, and PostgreSQL Server databases, performs regular full, differential, and transaction log backups, runs file/folder backup, zips and encrypts the backups, stores them on a network or on an FTP server or in the cloud (Amazon S3 and others - we're constantly adding more), removes old backups, and sends an e-mail confirmation on the job's success or failure.

This tool is used to create a scheduled backup job. This job runs at a predetermined time, backs up the data, and sends a success or failure email to a specified email address.

Figure.60 depicts the backup job that has been created to run daily with the tool. Whereas Figure. 61 and 62 represent a successful backup run and the mail confirmation received after completion of the job.

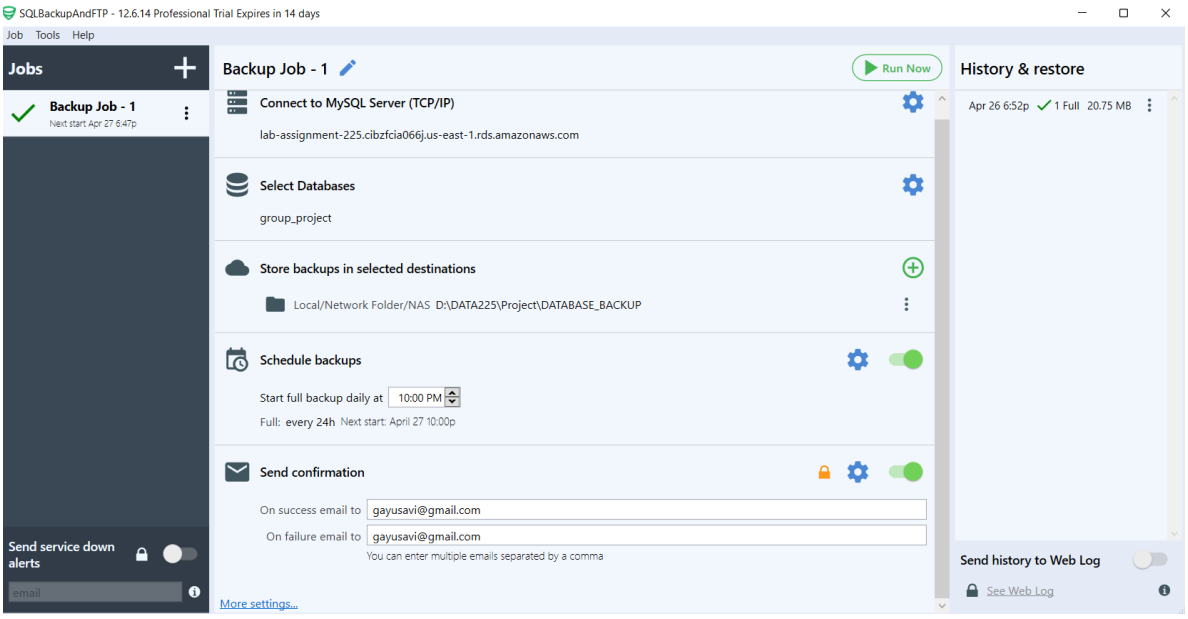


Figure.60 Settings of the backup job

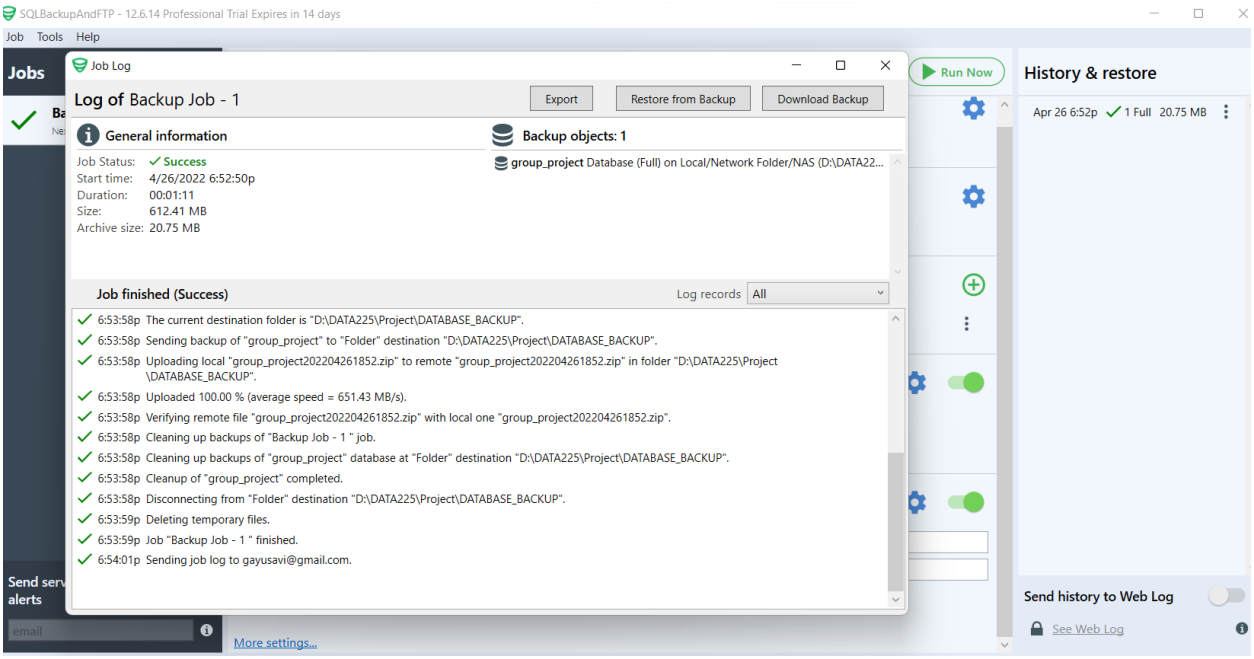


Figure.61 Success of the backup job

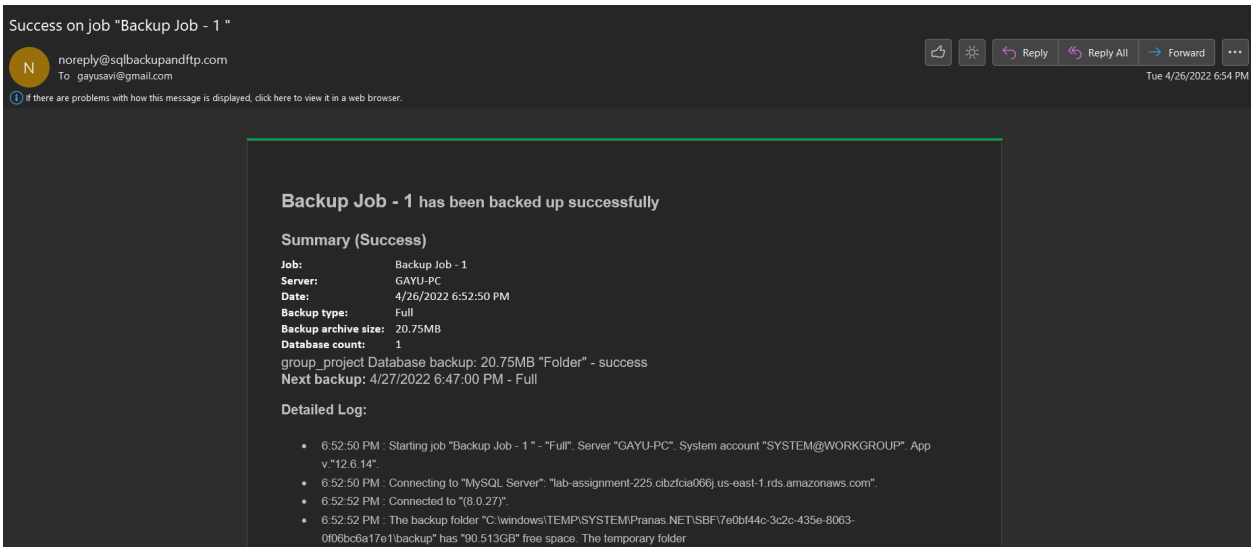


Figure.62 Mail confirmation after successful backup

14.2.2 AWS RDS

Apart from the partitioned and not partitioned tables created in local host, Another MySQL table is also created connected to the Amazon RDS. The original dataset is imported into that as well. Since operating costs for the AWS RDS table is exponentially high, this option was not

explored when dealing with the concurrent reads. The main reason behind these table's creation is, to explore more towards the direction of Locality as well as Backup and Recovery. Hence, this is used in this section.

Unlike in localhost, AWS RDS comes with an automated backup job runs at a predefined time regularly and stores the backups for a prescribed retention period. The backup process doesn't cost much, and it will be charged only in case of any recovery operation performed. Although it has a lot of pros, it is not devoid of cons. The main drawback of AWS RDS Backup is, only InnoDB storage engines are supported for automated backups. As a result, if the data needs to be stored using a different storage engine, such as Archive storage for historical data storage, it is not feasible.

The user specifies a backup window which will be used by the Amazon RDS Backup job to create and save automated backups of the DB instance, which in this case is daily. By backing up the entire DB instance, RDS creates a storage volume snapshot. There is a retention period that the user specifies for which RDS saves backups. Here, the retention period is set to 7 days. And recovery can be performed as needed by the user during the retention period. There are two major rules to follow when considering this automated backup strategy. The first requirement is that the DB instance be available, and that the AWS region of the DB instance and the DB snapshot copy not be the same.

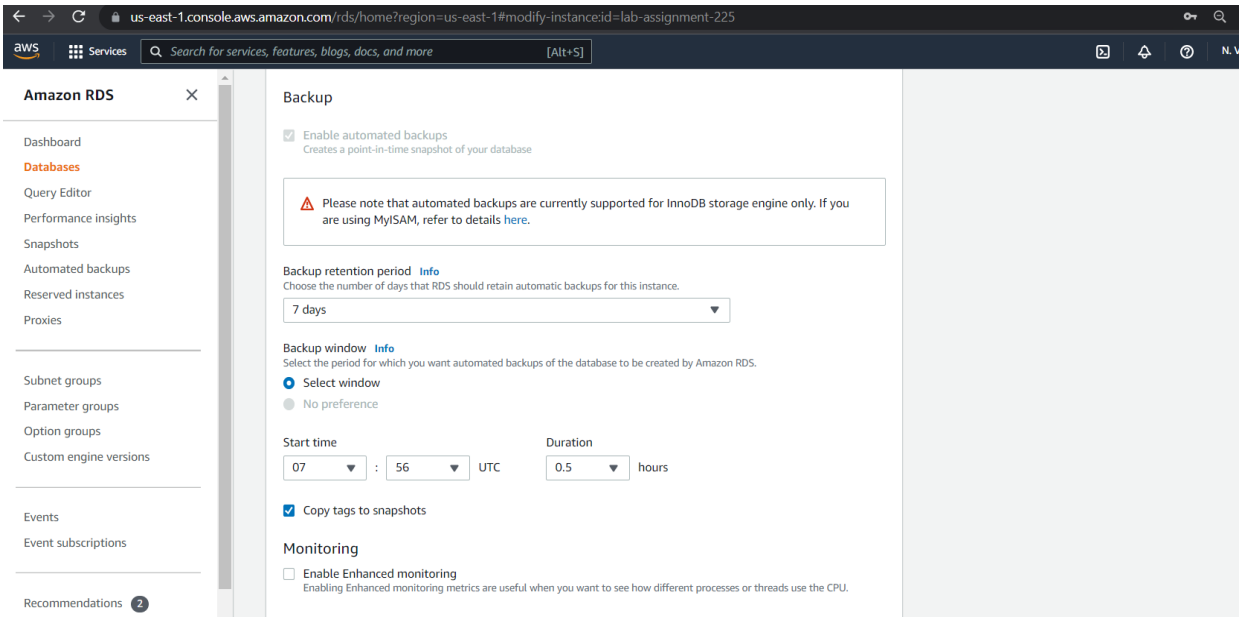


Figure.63 AWS Backup Settings

Figure.63 depicts the AWS Backup job settings related to the RDS where the connection to database tables is made.

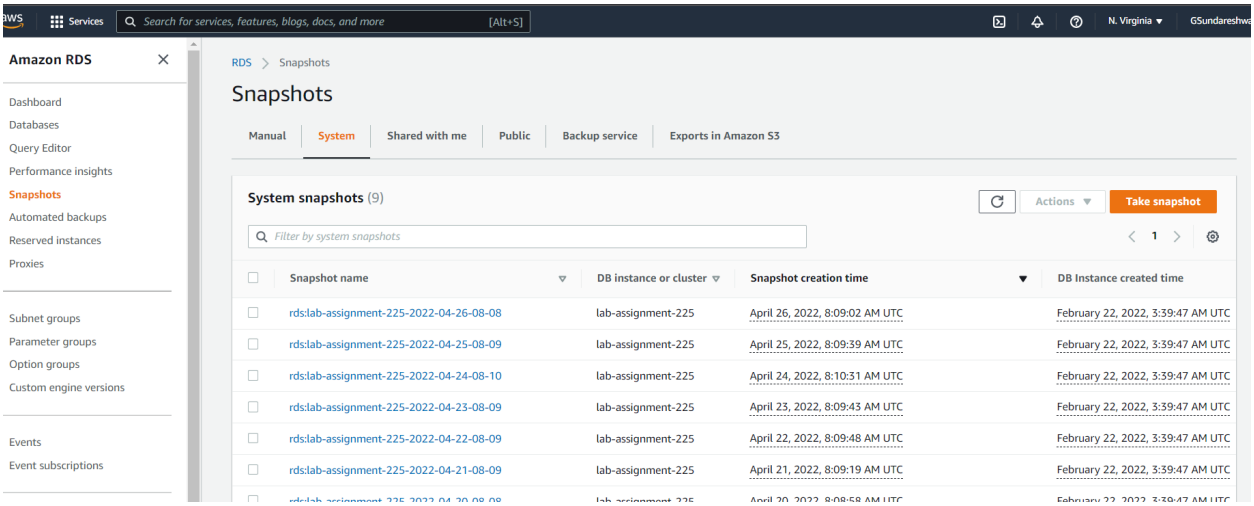


Figure.64 Backup created on regular basis

Figure.64 depicts the backups that were created periodically and retained in snapshots. AWS Backup works efficiently , without much human interference. But the con outweighs the pros.

14.2.3 Backup Using Windows Task Scheduler

After analyzing the requirements of this project and the factor influencing the backup job such as the locality of data (in this case, the localhost) a new approach for backup has been coined. This approach involves the creation of a windows batch file with the backup commands, which will be run on a scheduled basis.

This batch file will then be linked to a task that is created in the windows task scheduler. The scheduler will be configured in a way, that it runs on specified time whenever the system is up.

Backup created using this task scheduler and windows batch file will be an extract of .sql file containing all the commands that would take to restore the database to the state it was before any crash. This backup is optimal in case if the data is present in the localhost.

Also, unlike AWS backup jobs, it is not limited to just innoDB. Also, this backup job can backup data from SQL, mongo, AWS etc. if configure to the respective type properly in the windows batch file. The destination file name can also be specified in the batch file itself.

The other pro of this process is, it is practically free of cost, the job can be monitored through task scheduler logs .

Using the windows task scheduler, backup option is tested involving both local host and AWS DB . The snapshots below will provide a better understanding of the process.

Through this, it is proved that it is useful with both local host databases and AWS Databases as well.

Figure.65 depicts the widows batch file that was created for AWS and Figure.66 represents the batch file created for the database in localhost

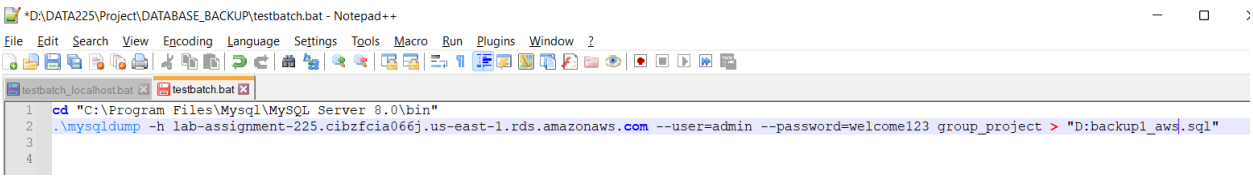


Figure.65 Windows batch file created for AWS DB backup

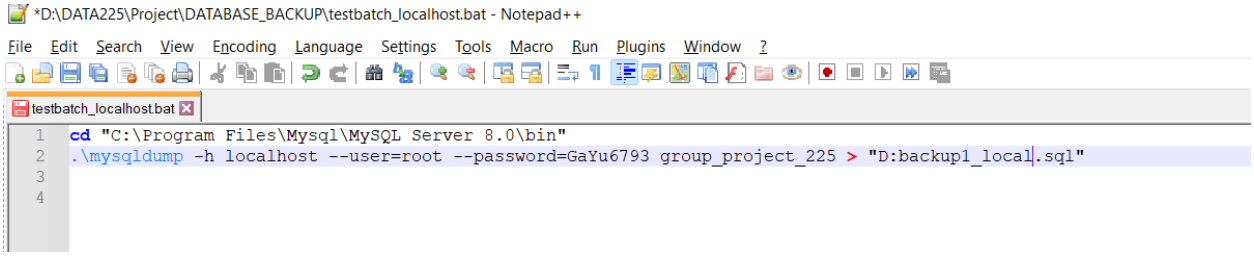


Figure.66 Windows batch file created for localhost database backup

The batch files created runs to generate the backup in .sql format to the destined target location. And the action, trigger setup can be modified to suit the preference and necessity of the project.

Figure.67 depicts the task (the backup job) created pointing to the AWS DB instance and Figure.68 represents the same job but run pointing to the database in local host.

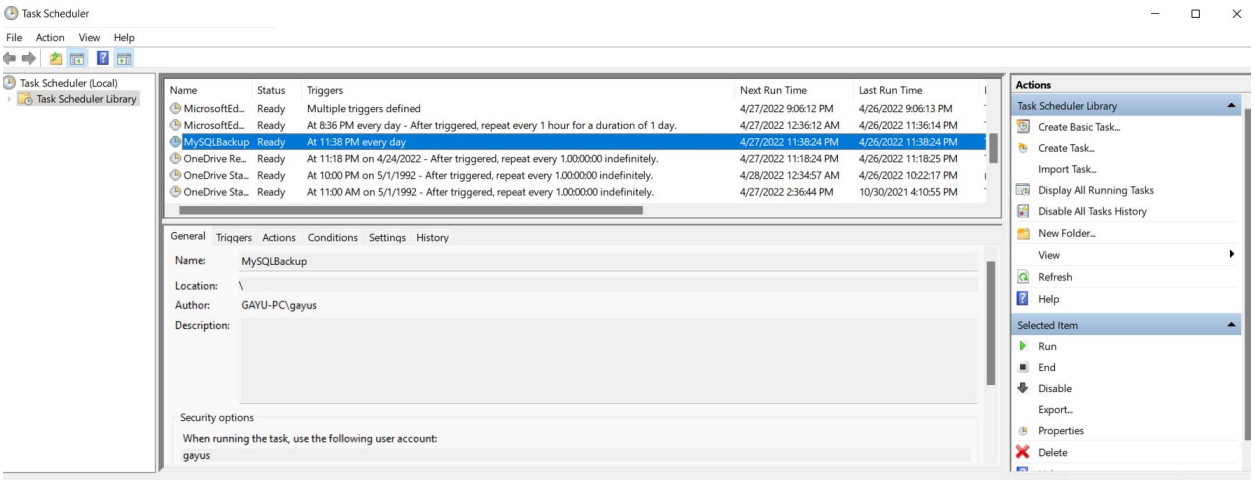


Figure.67 Task created for AWS DB Backup

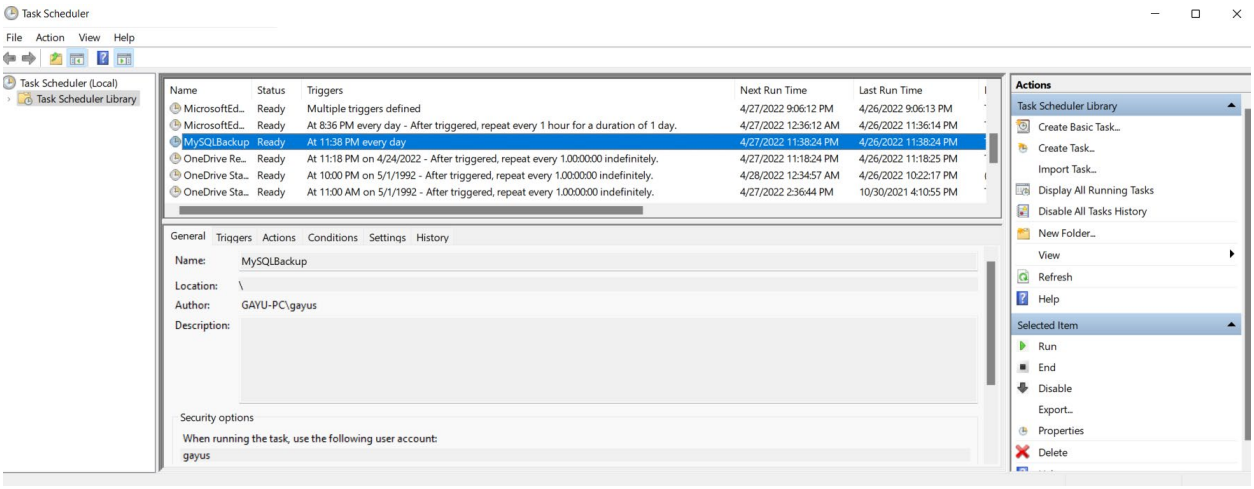


Figure.68 Task created for localhost Backup

Figure.69 represents the scheduled task for AWS running on scheduled time and Figure.70 represents the scheduled task for local host running as scheduled.

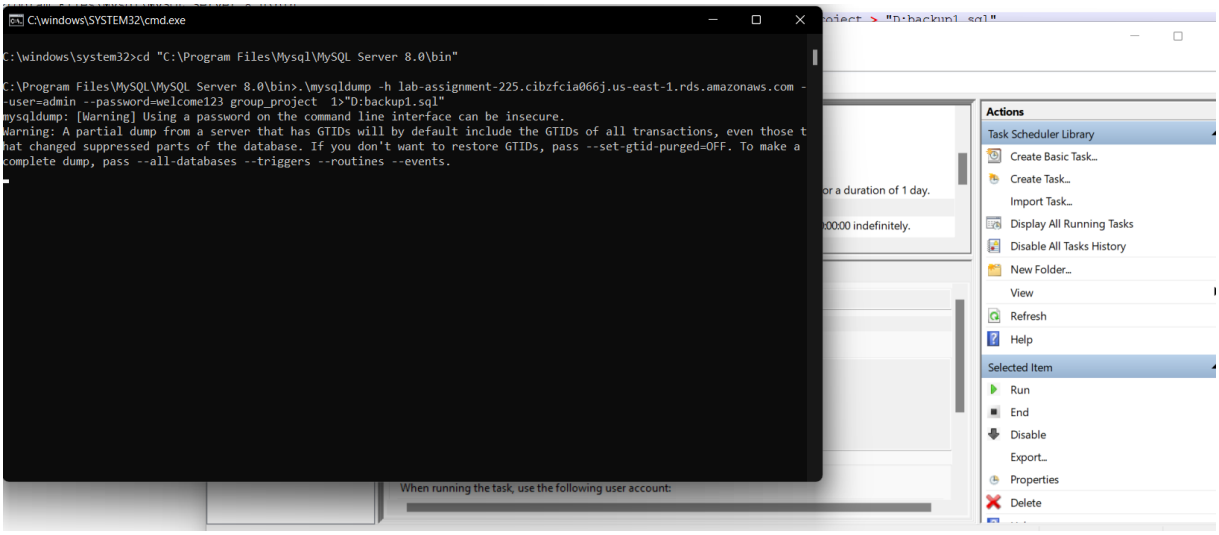


Figure.69 Backup task of AWS DB running as scheduled

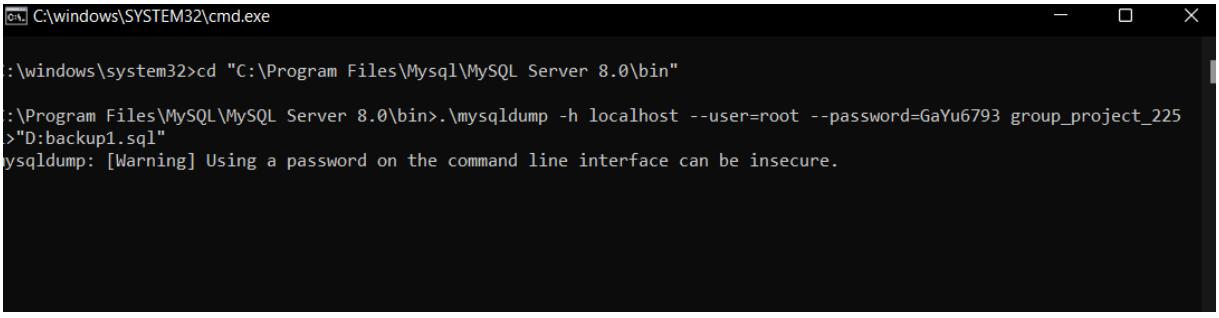


Figure.70 Backup task of localhost running as scheduled

Figure.70 shows the completion of backup job and figure 71 backup file generated by both the AWS and Localhost backup jobs.

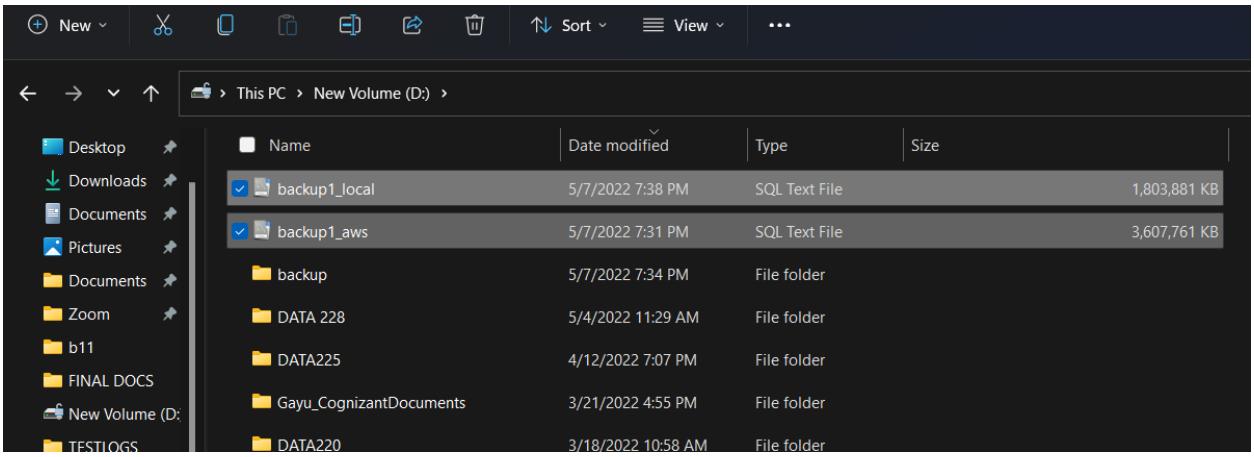


Figure.71 SQL Files created after backup job completion

14.3 Recovery

In the above sections, different backup approaches were explored. A backup is always only useful when it is restored. Suppose if any crash or corruption happens to the database itself or the data, it is mandatory to restore the database with the most recent backup so that there is a minimal data loss, if fortunate it is none. It also ensures that, the data integrity. For this reason, it is always necessary to restore the backup as soon as the crash recovery is done. Restoration of backed up .sql file or s3 instance is a separate process and this will be discussed in this section.

Recovery of the backup created through each of those above listed approach is discussed below.

14.3.1 SQLBackupAndFTP Tool – Recovery of the Backup

This tool generates a SQL file on successful backup operation and sends a confirmation mail. The backup can be restored by opening the .SQL file in MySQL workbench and executing the script. The only con is, in case of larger data size, the longer it takes to restore.

Figure.72 depicts the successful completion of backup job and mail received after it. Figure.73 represents the snapshot of a portion of .SQL file opened using MySQL . Once this .SQL file is run, the database can be restored to most recent backed up state.

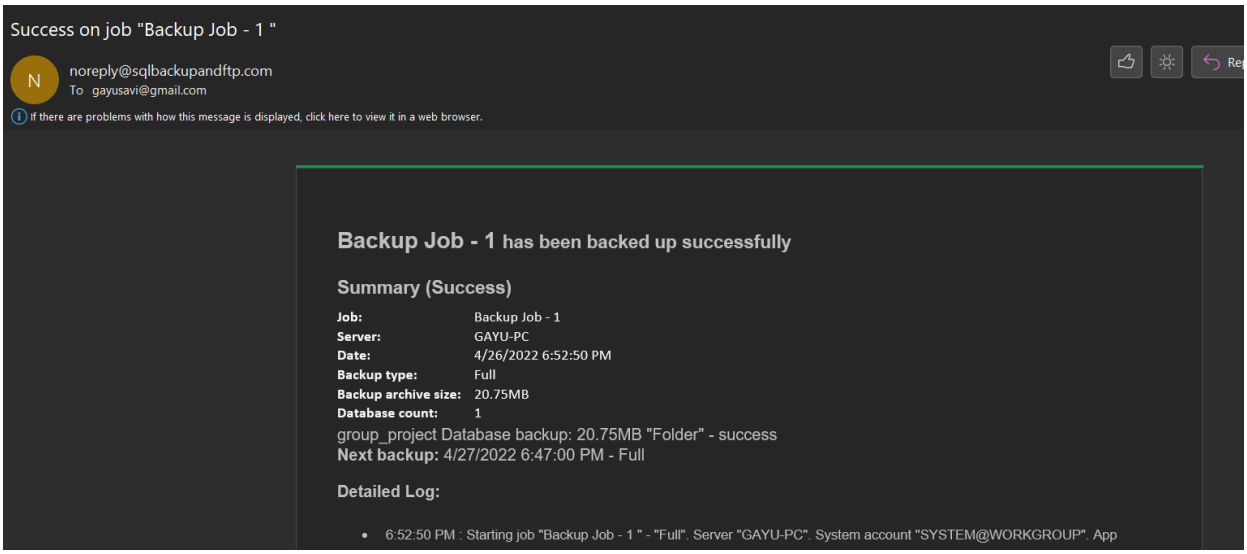


Figure.72 Confirmation mail after backup job completion

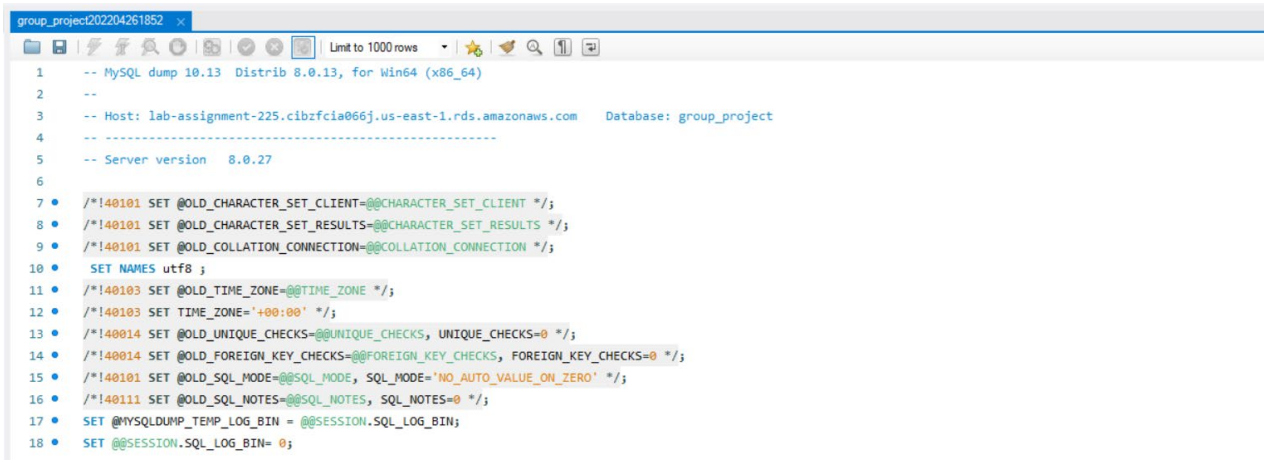


Figure.73 Backed up .SQL file which can be executed to restore the database

14.3.2 AWS RDS – Recovery of the BackUp

AWS Backs up the data regularly as a snapshot of type General Purpose SSD. The restoration of this backup is easy since all it takes is a few clicks. As said before, AWS DB instances are costly since its billed monthly. If the dataset is needed to be accessed from multiple

systems simultaneously , it is always recommended to choose a cloud approach, which is using software like AWS instead of having the data in localhost.

Fisure.74 depicts the recovery of data backed up periodically. The lab-assignment-225 is the original instance and ‘mydbinstance’ is the backup which is getting recovered.

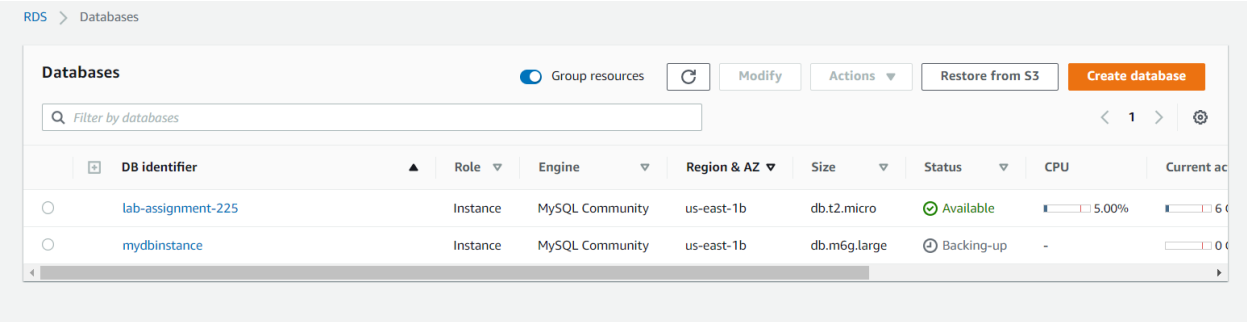


Figure.74 Recovery of the Backup instance

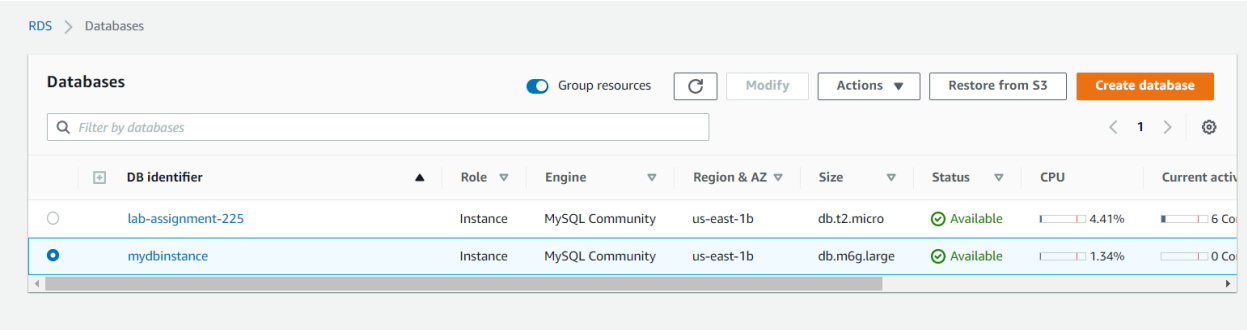


Figure.75 Successful restoration of the backup instance

Figure.75 represents the successful completion of the restoration work. The new instance created after recovery will have the connection details to the MySQL as present in the original instance.

14.3.3 Windows Task Scheduler – Recovery of the Backup

This task generates a .SQL file on successful backup operation. Similar to the first approach, the backup can be restored by opening the .SQL file in MySQL workbench and executing the script. The only con is, in case of larger data size, the longer it takes to restore.

Figure.76 depicts the successful completion of backup job of both AWS and localhost databases. Figure.77 represents the snapshot of a portion of .SQL file opened using MySQL . Once this .SQL file is run, the database can be restored to most recent backed up state.

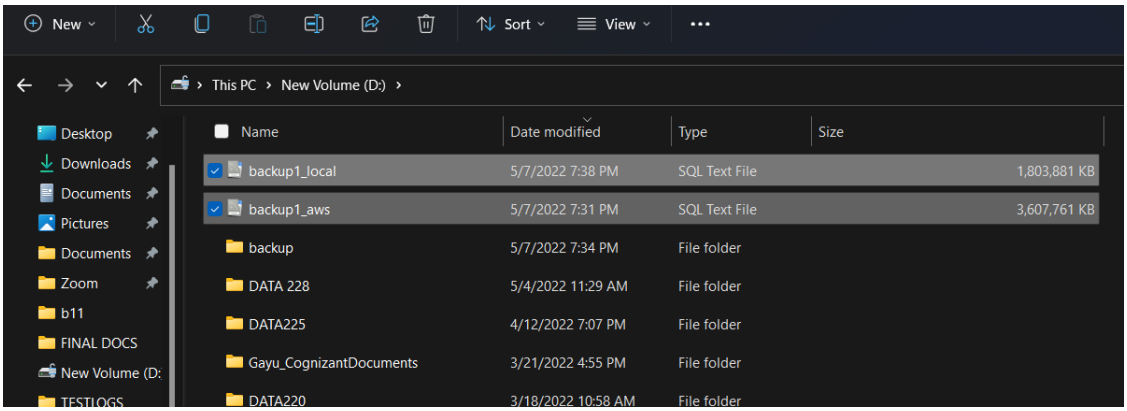


Figure.76 SQL Files created after backup job completion

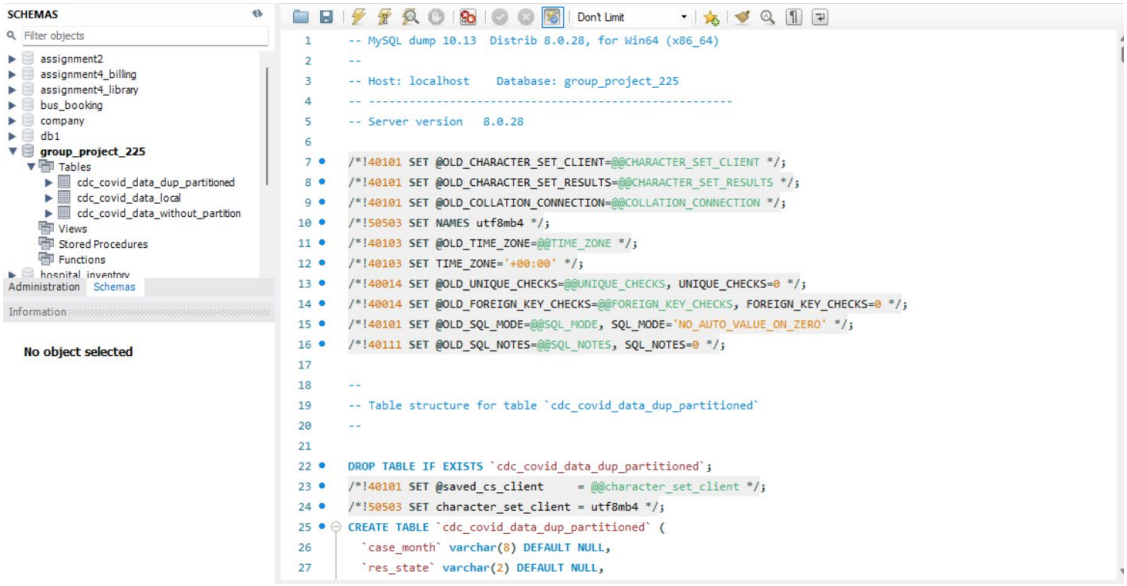


Figure.77 Backed up .SQL file which can be executed to restore the database

14.4 Observations Gathered

Failure backup and recovery may seem exhausting amount of work to be done .There are many techniques available which can minimize the need to recover, but no amount of planning

and regulation can prevent an unexpected failure of a system. The above discussed options are three among many. This can be explored further and the best suited one can be chosen.

The concluded solution for this project is the task scheduler method. This is cost free and it runs for all kinds of databases. It is not limited to the engine type or configuration.

Failure, corruption, and crash can happen anytime. Sometimes due to manual error, sometime due to catastrophic error like natural disasters. A story backup and recovery system in place will ensure peace of mind as well as security to the data. Since most of the time, the data dealt with is sensitive.

15. INSIGHTS ATTAINED

This project and the insights attained from the different loads can be come handy when deciding on which approach to use for what kind of data. Some of the operations performed in this project and the insights attained from each of the steps will be listed down below:

- Having audit logs help in tracking the changes made , along with the information regarding who changed , what kind of change it was and also the timestamp at which the change was made. In case if any misinformation was fed into the system this can be tracked easily with the help of the audit log
- Creating users with different levels of privilege helps in narrowing down on who has access to what. Important operations such as insert, update and delete can be limited to the users who require that kind of privileges . The other lower end users who only need to view the data, can be granted only with select privilege. This helps in increasing the security of the data
- 4 types of operations decided as the cross product of 2 different connections (JDBC and ORM) and 2 different table types(one with and one without partitions) helped in understanding how each of those factors works. Performing concurrent large and small reads on them helped in realizing how efficient the system is when dealing with respective loads. The consistent insight attained from the test using concurrent large and small reads is that JDBC performs faster than ORM when fetching the data. The tables with partitions work well when categorizing the data using partitions are necessary and often used as a part of business requirement. Fetching the data based on partitions fare well with the partitioned tables whereas, in case of operations where it

is necessary to scan the entire data , there doesn't exist much difference between the time taken for execution using partitioned or non-partitioned tables.

- The insights attained from testing the system for the most optimal locality factor can be discussed here. Locality is the process where the computation is moved to the location where the data resides. Two approaches were tested for this, one being "InnoDB Clustering" and the other being "AWS RDS". AWS Comes in handy when dealing with larger datasets and anyone with the connection can access the data. Hence, AWS RDS is decided as the optimal solution among the test approaches
- Different ways to reduce the memory pressure were discussion and the main insight attained is, optimization of query to the maximum possible level so that unnecessary memory usage can be cut down. Most of memory usage issues can be limited when the schema creation and query optimization are done properly
- NDB Cluster approach is implemented to deal with the capacity issues. This way, the capacity of the program can be significantly increased . DigitalOcean can be helpful with creating droplets .My SQL can be installed in each of the droplet created . A master node and cluster management created in DigitalOcean will come handy when dealing with capacity issues.
- It is mandatory for a system to have an efficient failure , backup and recovery system in place. The optimal approach for this project is, since the data resides on local host, a windows batch file is created which will run a backup job regularly on scheduled intervals. This backup will be a .sql file and in case if any crash happens , the system can be restored to the last checkpoint by executing the backed up .sql file

There are multiple ways to approach a solution while fulfilling the business requirement. The most optimal approach decided for one project is not necessarily the same for another requirement. The optimal solution solely depends on the type of data, the requirements of the system as well as the business requirements set by the executives. There is a fine line between balancing the optimal solution to fit all the categorical needs and overfitting it. Hence, all the above mentioned deciding factors must be analyzed and the optimal one fitting all the scenarios must be decided. A few compromises is necessary here and there, but this compromises will help in setting up an efficient system.

16. LIMITATIONS

It is undeniable that, every model is not 100 percent perfect. There will always be some limitations occurring, even if not now , it might occur in the future. The limitations identified while working on the process must be kept track so that in future releases some of those can be addressed if possible.

Some of the limitations observed in this project development are listed below:

- Since the data resides in MySQL , all the major limitations of MySQL dealing with larger datasets will be applicable here. But the requirement is to create a system with MySQL, so these limitations of MSQL will persist in the system developed as well.
- AWS RDS is used to test the locality and capacity factor, but since the account used is a normal personal account, the bills generated are quite on the upper side. So, the AWS RDS comes in handy only when the development is done for business, not for personal reasons.
- The dataset from CDC doesn't have a primary column since the patient identity and details are not to be revealed, and the dataset contained only the most generic information. For instance, in the state of Texas, there may be more than one Hispanic female in the age range of 65+ that got affected and all the other factors might also be same for more than one person. Tracking individual records is not possible in this case.
- The system doesn't have a process in place where an email is generated in case of any crucial column is getting updated. Even though an audit log is kept in place, sometimes the changes to this data might impact other applications before the audit log is noticed and the correction is done.
- Encryption mechanism can be kept in place since the data from CDC serves as a source to multiple other systems. A strong encryption mechanism can help prevent the data loss.

- Exploring other paid services were not done since most of the services are not cost effective. Sometimes , a better solution can be arrived from a paid service.
- Due to limited resources, additional exploratory scenarios were not tested
- Visualizations done are mostly 2D. In some cases, 3D visualizations might be more helpful than 2D.
- Geospatial data is not utilized to the full extent.

17. SCOPE

Some of the limitations discussion in the above sections can be fixed in the near future. The most probable ones are listed below

- Encryption can be done if necessary and the business asks for it
- An email verification system can be kept in place. Algorithms like AWS Key management services or RSA , DSA , DH etc can come handy for this process
- 3D Visualizations can be implemented if required
- Stress testing can be done to cover a wide range of scenarios, so that high scalability can be achieved with the most optimal solution
- User permission can be refined more, so that multiple tiers of permission can be created.
- The implementation can be migrated to other database systems be it, SQL or NoSQL as per the requirements
- API Can be developed in case if necessary and requested by the business

18. CONCLUSION

While designing the model, there doesn't exist one single approach since each of the deciding factor plays a major role. One solution might be the most preferred for factor A whereas, it might be the worst performing one for factor B. All these factors combine in generating the optimal model. All the factors must work in perfect, if not related harmony so that the solution doesn't have too many loopholes and drawbacks. MySQL has so many advantages since it is a mature language. This advantage should be utilized to the maximum so that future fixes can be minimized to the maximum possible extent.

19. REFERENCES

1) https://www.cloudways.com/blog/mysql-performance-tuning/
2) https://data.cdc.gov/Case-Surveillance/COVID-19-Case-Surveillance-Public-Use-Data-with-Ge/n8mc-b4w4/data
3) https://data.cdc.gov/Case-Surveillance/COVID-19-Case-Surveillance-Public-Use-Data-with-Ge/n8mc-b4w4
4) https://severalnines.com/blog/handling-large-data-volumes-mysql-and-mariadb
5) https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html
6) https://dev.mysql.com/doc/refman/8.0/en/optimize-table.html#optimize-table-innodb-details
7) https://dev.mysql.com/doc/refman/8.0/en/glossary.html#glos_page
8) https://stackoverflow.com/questions/5541421/mysql-sharding-approaches
9) https://dev.mysql.com/doc/refman/5.7/en/mysql-cluster-installation.html
10) https://dev.mysql.com/doc/mysql-shell/8.0/en/mysql-innodb-cluster.html
11) https://seo-explorer.io/blog/twenty-ways-to-optimize-mysql-for-faster-insert-rate/#innodb_buffer_pool_size
12) https://dev.mysql.com/blog-archive/mysql-innodb-cluster-a-hands-on-tutorial/
13) https://stackoverflow.com/questions/612428/pros-and-cons-of-the-mysql-archive-storage-engine
14) https://dev.mysql.com/doc/refman/8.0/en/memory-use.html
15) Why recovery is needed in DBMS - GeeksforGeeks
16) https://sqlbackupandftp.com/
17) https://www.mysql.com/products/cluster/mcm/

18) https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/mysql-cluster-basics.html

19) what is a digital ocean droplet - Search (bing.com)

20) https://www.starwindsoftware.com/resource-library/data-locality/

21) Optimizing Schema and Data Types - High Performance MySQL, 3rd Edition [Book] (oreilly.com)

22) How to fix MySQL high memory usage (bobcares.com)

23) MySQL :: MySQL 8.0 Reference Manual :: 8.12.3.1 How MySQL Uses Memory

24) Droplets | DigitalOcean's Scalable Virtual Machines

25) Droplet Quickstart :: DigitalOcean Documentation

26) Optimizing MySQL Stored Routines (logicalread.com)

27) Monitoring Stored Procedure Usage | Database Journal

28) https://dev.mysql.com/doc/mysql-cluster-excerpt/8.0/en/mysql-cluster-overview.html

29) https://dev.mysql.com/doc/mysql-router/8.0/en/

30) Use SQLAlchemy ORMs to Access MySQL Data in Python (cdata.com)