Big Data Analytics Project 2

Oboni Anower

13923163

July 2024

Task 1 – Entity Relationship Model

1.1. Entities

There are five identified entities in the given use case of MiniNet. Such entities are Users, Subscriptions, Movies, FavoriteList and Actors.

Subscriptions: A logical entity that represents service plans available for users in the MiniNet system.

Users: A physical entity which represents real people to use MiniNet platform. This table stores personal records of each user.

FavoriteList: A logical entity that represents the lists that correspond to the favourite movies of the users. This is a reference table for FavoriteMovies which will be discussed in section 1.2 (e).

Movies: A physical entity that signifies the films and tv shows available on MiniNet.

Actors: A physical entity that has information about different actors who acted in the movies.

1.2. Relationships and Cardinalities

The relationships and cardinalities (TutorialsTeacher, n.d.) between entities are as follows:

a) An HD or UHD subscription is *subscribed by* a user.
`Subscriptions` table has a Primary Key `sub_id` which is a foreign key in `Users` table. This makes a **one-to-one** relationship between `Subscriptions` and `Users` tables.

b) Each user *has* only one favourite list.
`FavoriteList` is a reference table which contains foreign key `User_id` from `Users` table. They have **one-to-one** relationship between them.

c) Users *reviews* movies.
`Users` have **many-to-many** relationships with `Movies` using foreign keys `User_id` and `Movie_id` in `Reviews` junction table. This is because every user can review one or more

movies. Individually, `Users` and `Movies` have **one-to many** relationships with `Reviews`, because a movie can be reviewed only once by a single user.

d) Users have *watch history* for movies.
This is a **many-to-many** relationship using foreign keys `User_id` and `Movie_id` in `WatchHistory` junction table. It is possible because every user can have multiple watched movies. Alternatively, their individual relationship to the junction table is **one-to-many**.

e) Users have *favourite movies*.
Similar to the previous relations, `Users` have **many-to-many** relationships with `Movies` using the `User_id` and `Movie_id` foreign keys in `FavouriteMovies` junction table. The users can have many favourite movies. Since one user can have many favourite movies, and one movie can be liked by many users, the individual relationship with `FavouriteMovies` is **one-to-many**.

This junction table also has reference to `FavoriteList` using foreign key `favlist_id`, with a **one-to-many** relationship. This is due to the fact that each favourite list of a user can have many favourite movies.

f) Movies have *movie actors*.
In this case, `Movies` have **many-to-many** relationships with `Actors` using the junction table `MovieActors`. `Movie_id` and `Actor_id` are the foreign keys that help establish this relationship, suggesting that the movies can be acted by many actors. Their individual relationships show that an actor can star in many movies, whereas a movie can have many actors, making **one-to-many** relationships with `MovieActors` table.

Figure 1 shows the Entity relations with addition to their cardinal relationships. Here, the grey rectangles demonstrate the entities, and the black diamonds represent the relations between them. The cardinalities have been stated for each relationship.

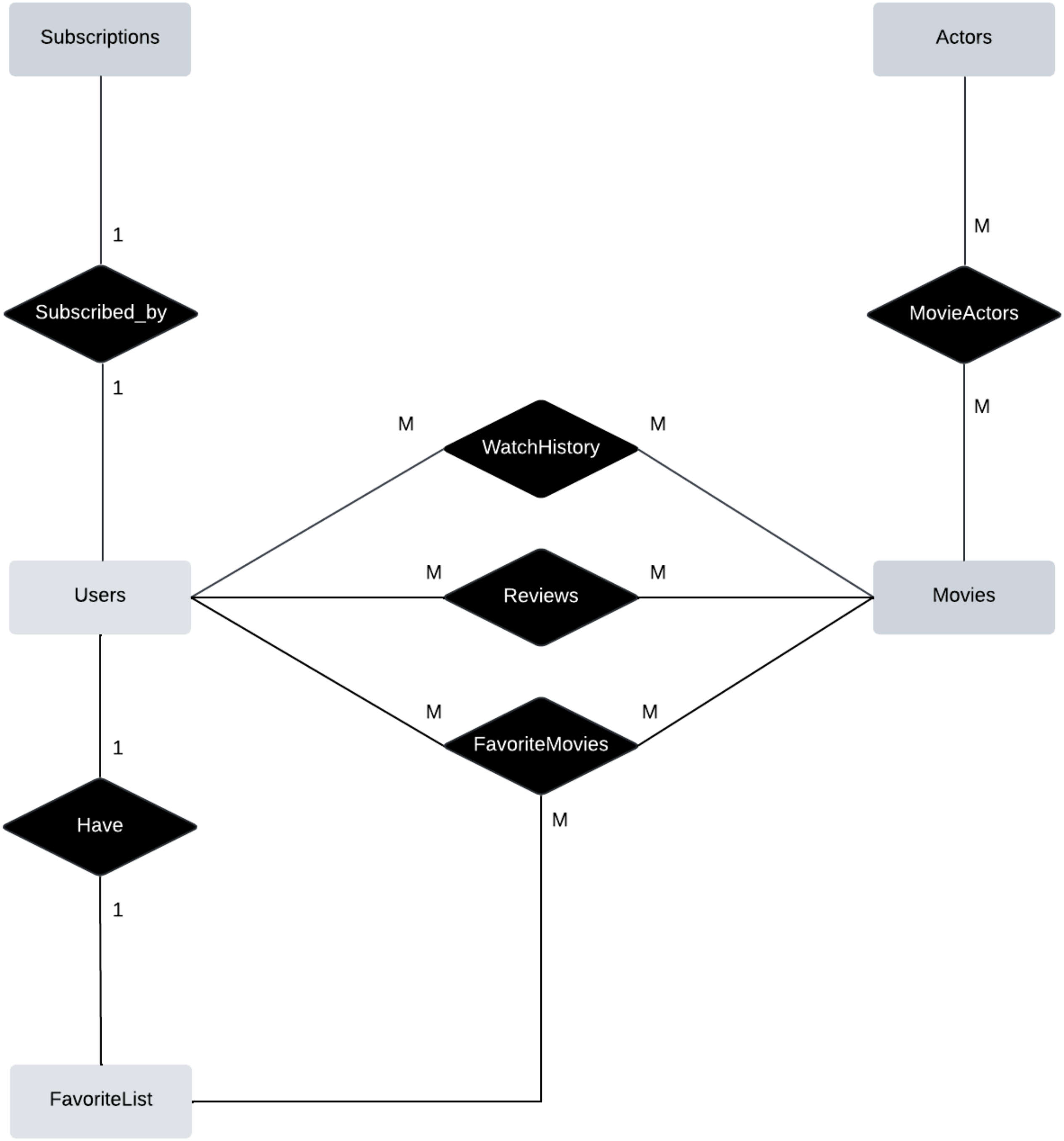
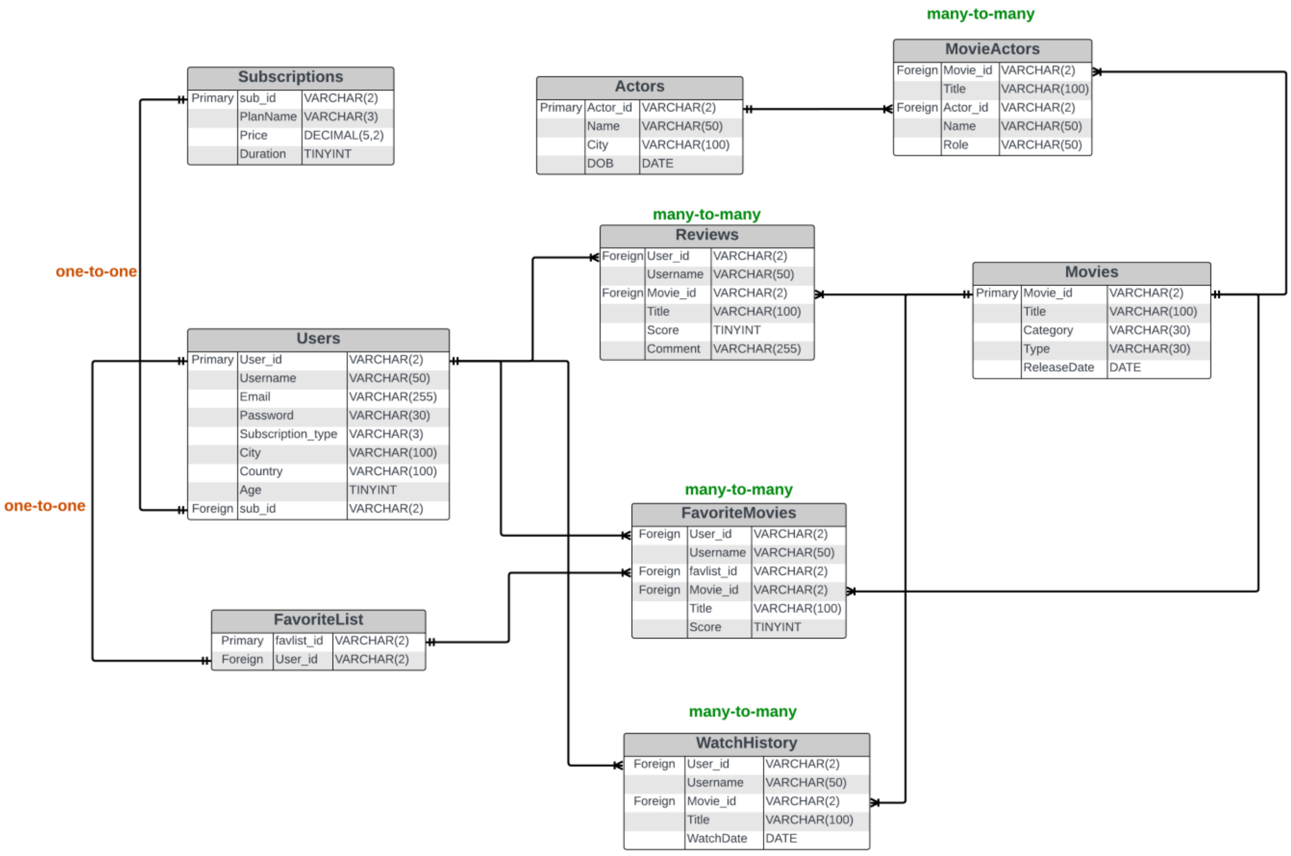


Figure 1: Entity relations and cardinality

1.3. ER Diagram



Subscriptions

Primary	sub_id	VARCHAR(2)
	PlanName	VARCHAR(3)
	Price	DECIMAL(5,2)
	Duration	TINYINT

Actors

Primary	Actor_id	VARCHAR(2)
	Name	VARCHAR(50)
	City	VARCHAR(100)
	DOB	DATE

MovieActors

Foreign	Movie_id	VARCHAR(2)
	Title	VARCHAR(100)
Foreign	Actor_id	VARCHAR(2)
	Name	VARCHAR(50)
	Role	VARCHAR(50)

Reviews

Foreign	User_id	VARCHAR(2)
	Username	VARCHAR(50)
Foreign	Movie_id	VARCHAR(2)
	Title	VARCHAR(100)
	Score	TINYINT
	Comment	VARCHAR(255)

Movies

Primary	Movie_id	VARCHAR(2)
	Title	VARCHAR(100)
	Category	VARCHAR(30)
	Type	VARCHAR(30)
	ReleaseDate	DATE

Users

Primary	User_id	VARCHAR(2)
	Username	VARCHAR(50)
	Email	VARCHAR(255)
	Password	VARCHAR(30)
	Subscription_type	VARCHAR(3)
	City	VARCHAR(100)
	Country	VARCHAR(100)
	Age	TINYINT
Foreign	sub_id	VARCHAR(2)

FavoriteMovies

Foreign	User_id	VARCHAR(2)
	Username	VARCHAR(50)
Foreign	favlist_id	VARCHAR(2)
Foreign	Movie_id	VARCHAR(2)
	Title	VARCHAR(100)
	Score	TINYINT

FavoriteList

Primary	favlist_id	VARCHAR(2)
Foreign	User_id	VARCHAR(2)

WatchHistory

Foreign	User_id	VARCHAR(2)
	Username	VARCHAR(50)
Foreign	Movie_id	VARCHAR(2)
	Title	VARCHAR(100)
	WatchDate	DATE

Figure 2: ER Diagram

Figure 2 shows the Entity Relationship Diagram with the data type of each attribute in entity, reference, and junction tables. The data sizes (Rainardi, 2015) are explained below:

a) All the table ids are stored as variable length string of size 2. The ids will be entered manually which will be no longer than 2 letters throughout task 2-4.

b) Personal names such as Username and Actor names and Roles are stored as variable length string of size 50. Email addresses and passwords are both the same data type with 255 and 30 respectively. Ages are stored as TINYINT (W3Schools, n.d.), meaning signed integer between -128 to 127.

c) City and Countries are stored as variable length string of 100.

d) Movie titles are stored as variable length 100, considering movies with long names. Categories and types are kept of size 30.

e) All sort of dates is set to DATE (W3Schools, n.d.), which gives YYYY-MM-DD format.

f) Rating scores are set to TINYINT (W3Schools, n.d.), which is restricted from 0 to 5 (in task 2). Comments are given variable length string of 255.

g) Subscription types are size 3 because there are only two types, HD and UHD, with the longest entry being 3 characters.

Task 2 Create Database Model in SQL

The scripts below show the SQL commands to create the entity and junction tables. The primary and composite keys are highlighted in blue, and the foreign keys are highlighted in purple.

Subscription:

```
CREATE TABLE Subscriptions (
sub_id VARCHAR(2),
PlanName VARCHAR(3) NOT NULL,
Price DECIMAL(5, 2),
Duration TINYINT,
PRIMARY KEY (sub_id)
);
```

Users:

```
CREATE TABLE Users (
User_id VARCHAR(2),
Username VARCHAR(50),
Email VARCHAR(255) UNIQUE NOT NULL,
Password VARCHAR(30),
Subscription_type VARCHAR(3),
City VARCHAR(100),
Country VARCHAR(100),
Age TINYINT,
sub_id VARCHAR(2),
PRIMARY KEY (User_id),
FOREIGN KEY (sub_id) REFERENCES Subscriptions(sub_id)
);
```

Movies:

```
CREATE TABLE Movies (
Movie_id VARCHAR(2),
Title VARCHAR(100),
Category VARCHAR(30),
Type VARCHAR(20),
ReleaseDate DATE,
PRIMARY KEY (Movie_id)
);
```

Actors:

```
CREATE TABLE Actors (
Actor_id VARCHAR(2),
Name VARCHAR(50),
City VARCHAR(100),
DOB DATE,
PRIMARY KEY (Actor_id)
);
```

FavoriteList:

```
CREATE TABLE FavoriteMovies (
User_id VARCHAR(2),
Username VARCHAR(50),
favlist_id VARCHAR(2),
Movie_id VARCHAR(2),
Title VARCHAR(100),
Score Score TINYINT CHECK (Score >= 0 AND Score <= 5),
PRIMARY KEY(User_id, favlist_id, Movie_id),
CONSTRAINT FK5 FOREIGN KEY (User_id) REFERENCES Users(User_id),
CONSTRAINT FK6 FOREIGN KEY (favlist_id) REFERENCES FavoriteList(favlist_id),
CONSTRAINT FK7 FOREIGN KEY (Movie_id) REFERENCES Movies(Movie_id)
);
```

Reviews:

```
CREATE TABLE Reviews (
User_id VARCHAR(2),
Username VARCHAR(50),
Movie_id VARCHAR(2),
Title VARCHAR(100),
Score TINYINT CHECK (Score >= 0 AND Score <= 5),
Comment VARCHAR(255),
PRIMARY KEY (User_id, Movie_id),
CONSTRAINT FK3 FOREIGN KEY (User_id) REFERENCES Users(User_id),
CONSTRAINT FK4 FOREIGN KEY (Movie_id) REFERENCES Movies(Movie_id)
);
```

MovieActors:

```
CREATE TABLE MovieActors (
Movie_id VARCHAR(2),
Title VARCHAR(100),
Actor_id VARCHAR(2),
Name  VARCHAR(50),
Role VARCHAR(50),
PRIMARY KEY (Movie_id, Actor_id),
CONSTRAINT FK1 FOREIGN KEY (Movie_id) REFERENCES Movies(Movie_id),
CONSTRAINT FK2 FOREIGN KEY (Actor_id) REFERENCES Actors(Actor_id)
);
```

FavoriteMovies:

```
CREATE TABLE FavoriteMovies (
User_id VARCHAR(2),
Username VARCHAR(50),
favlist_id VARCHAR(2),
Movie_id VARCHAR(2),
Title VARCHAR(100),
Score Score TINYINT CHECK (Score >= 0 AND Score <= 5),
PRIMARY KEY(User_id, favlist_id, Movie_id),
CONSTRAINT FK5 FOREIGN KEY (User_id) REFERENCES Users(User_id),
CONSTRAINT FK6 FOREIGN KEY (favlist_id) REFERENCES FavoriteList(favlist_id),
CONSTRAINT FK7 FOREIGN KEY (Movie_id) REFERENCES Movies(Movie_id)
);
```

WatchHistory:

```
CREATE TABLE WatchHistory (
User_id VARCHAR(2),
Username VARCHAR(50),
Movie_id VARCHAR(2),
Title VARCHAR(100),
WatchDate DATE,
PRIMARY KEY(User_id, Movie_id),
CONSTRAINT FK8 FOREIGN KEY (User_id) REFERENCES Users(User_id),
CONSTRAINT FK9 FOREIGN KEY (Movie_id) REFERENCES Movies(Movie_id)
);
```

Each entity table has its primary key, while the junction tables contain composite keys that are used as foreign key references. The combinations of the composite primary keys ensure that each record in the junction table is identified uniquely. The foreign keys are necessary to maintain referential integrity between the table relationships by keeping data on both sides intact (Ian, 2016).

For example, in `Reviews` table, composite keys `User_id` and `Movie_id` ensure that a specific user can review a specific movie only and only once. This helps to prevent duplicates. Likewise, the foreign key referencing ensures that any combination of `User_id` and `Movie_id` in `Reviews` table must also be present in `Users` and `Movies` tables. The same referential integrity is followed for all the junction tables with their respective foreign key references.

Task 3 Insert Data to the Database Tables

Once the tables are created, the records are inserted into the tables. For this coursework, mostly the data from supporting materials are used, with slight adjustments to some names and locations. Additionally, primary, and foreign key values are also included manually. It is made sure to include at least five records in each table excluding `Subscription` table, as it has only two types of subscription only.

Subscriptions:

```
INSERT INTO Subscriptions (sub_id, PlanName, Price, Duration) VALUES
('S1', 'HD', 9.99, 1),
('S2', 'UHD', 14.99, 1);
```

Users:

```
INSERT INTO Users (User_id, Username, Email, Password, Subscription_type, City,
Country, Age, sub_id)
VALUES
('U1', 'john_doe', 'john@example.com', 'password1', 'HD', 'New York', 'USA', 28,
'S1'),
('U2', 'alice_smith', 'alice@example.com', 'password2', 'HD', 'London', 'UK', 34,
'S1'),
('U3', 'tanaka_yuji', 'tanaka@example.com', 'password3', 'UHD', 'Osaka', 'Japan',
21, 'S2'),
('U4', 'bob_jones', 'bob@example.com', 'password4', 'UHD', 'Boston', 'USA', 30,
'S2'),
('U5', 'emma_johnson', 'emma@example.com', 'password5', 'HD', 'Sydney',
'Australia', 25, 'S1');
```

Movies:

```
INSERT INTO Movies (Movie_id, Title, Category, Type, ReleaseDate) VALUES
('M1', 'Stranger Things', 'Sci-Fi', 'TV-Show', '2016-07-15'),
('M2', 'Breaking Bad', 'Drama', 'TV-Show', '2008-01-20'),
('M3', 'The Office', 'Comedy','TV-Show', '2005-03-24'),
('M4', 'Parks and Recreation', 'Comedy', 'TV-Show', '2009-04-09'),
('M5', 'The Godfather', 'Crime',  'Film', '1972-03-24');
```

Actors:

```
INSERT INTO Actors (Actor_id, Name, City, DOB) VALUES
('A1', 'Millie Bobby Brown', 'Los Angeles', '2004-02-19'),
('A2', 'Bryan Cranston', 'Hollywood', '1956-03-07'),
('A3', 'Winona Ryder', 'New York', '1971-10-29'),
('A4', 'Aaron Paul', 'Boise', '1979-08-27'),
('A5', 'David Harbour', 'Los Angeles', '1975-04-10');
```

FavoriteList:

```
INSERT INTO FavoriteList (favlist_id, user_id) VALUES
('f1', 'U1'),
('f2', 'U2'),
('f3', 'U3'),
('f4', 'U4'),
('f5', 'U5');
```

Reviews:

```
INSERT INTO Reviews (User_id, Username, Movie_id, Title, Score, Comment) VALUES
('U1', 'john_doe', 'M1','Stranger Things', 5, 'Amazing show!'),
('U2','alice_smith', 'M2','Breaking Bad', 3, 'Good show'),
('U3','tanaka_yuji', 'M3','The Office', 4, 'Funny and smart'),
('U4','bob_jones', 'M4', 'Parks and Recreation', 2, 'Not my taste'),
('U5', 'emma_johnson', 'M5', 'The Godfather', 5, 'A classic!'),
('U1', 'john_doe', 'M5', 'The Godfather', 2, 'Not my taste.');
```

MovieActors:

```
INSERT INTO MovieActors(Movie_id, Title, Actor_id, Name, Role) VALUES
('M1','Stranger Things', 'A1','Millie Bobby Brown', 'Eleven'),
('M2','Breaking Bad', 'A2','Bryan Cranston', 'Walter White'),
('M1','Stranger Things', 'A3', 'Winona Ryder', 'Joyce Byers'),
('M2','Breaking Bad', 'A4', 'Aaron Paul', 'Jesse Pinkman'),
('M1','Stranger Things', 'A5', 'David Harbour', 'Jim Hopper');
```

FavoriteMovies:

```
INSERT INTO FavoriteMovies (User_id, Username, favlist_id,  Movie_id, Title, Score)
VALUES
('U1', 'john_doe', 'f1', 'M3', 'The Office', 5),
('U1', 'john_doe', 'f1', 'M4', 'Parks and Recreation', 4),
('U2', 'alice_smith', 'f2', 'M5', 'The Godfather', 3),
('U3', 'tanaka_yuji', 'f3', 'M1', 'Stranger Things', 5),
('U4', 'bob_jones', 'f4', 'M2', 'Breaking Bad', 4);
```

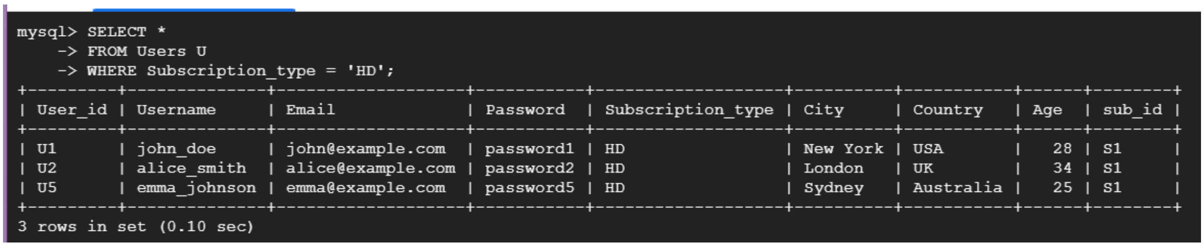
WatchHistory:

```
INSERT INTO WatchHistory (User_id, Username, Movie_id, Title, WatchDate) VALUES
('U1', 'john_doe', 'M1','Stranger Things', '2023-06-10'),
('U2','alice_smith', 'M2','Breaking Bad', '2023-06-11'),
('U3','tanaka_yuji', 'M3','The Office', '2023-06-12'),
('U4','bob_jones', 'M4', 'Parks and Recreation', '2023-06-13'),
('U5', 'emma_johnson', 'M5', 'The Godfather', '2023-06-14'),
('U1', 'john_doe', 'M5', 'The Godfather', '2023-06-16'),
('U3', 'tanaka_yuji', 'M5', 'Breaking Bad', '2023-06-18');
```

Task 4 SQL Queries for Extracting Data

4.1 Exports all data about all users in the HD subscriptions

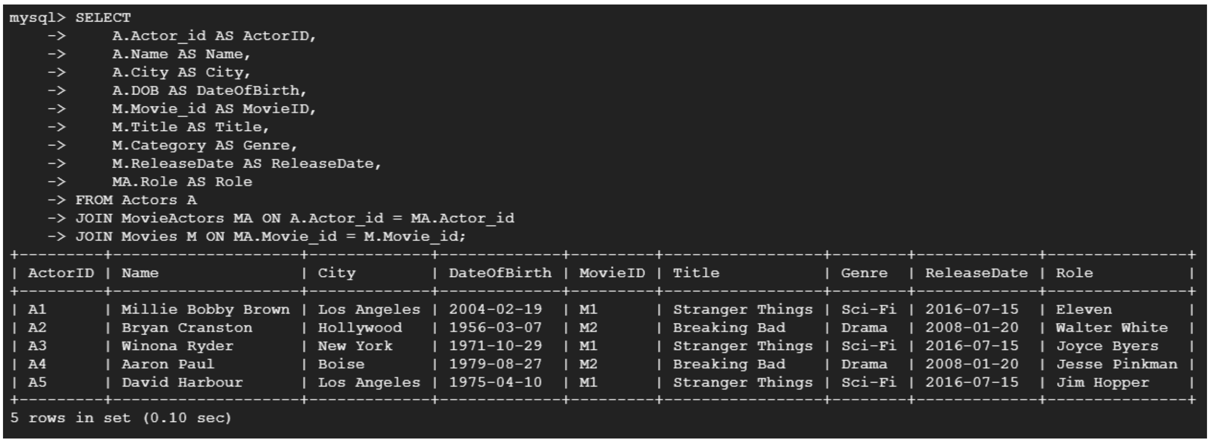
```
SELECT *
FROM Users U
WHERE Subscription_type = 'HD';
```



```
mysql> SELECT *
    -> FROM Users U
    -> WHERE Subscription_type = 'HD';
+---------+--------------+-------------------+-----------+-------------------+----------+-----------+------+--------+
| User_id | Username     | Email             | Password  | Subscription_type | City     | Country   | Age  | sub_id |
+---------+--------------+-------------------+-----------+-------------------+----------+-----------+------+--------+
| U1      | john_doe     | john@example.com  | password1 | HD                | New York | USA       |   28 | S1     |
| U2      | alice_smith  | alice@example.com | password2 | HD                | London   | UK        |   34 | S1     |
| U5      | emma_johnson | emma@example.com  | password5 | HD                | Sydney   | Australia |   25 | S1     |
+---------+--------------+-------------------+-----------+-------------------+----------+-----------+------+--------+
3 rows in set (0.10 sec)
```

4.2 Exports all data about all actors and their associated movies

```
SELECT
    A.Actor_id AS ActorID,
    A.Name AS Name,
    A.City AS City,
    A.DOB AS DateOfBirth,
    M.Movie_id AS MovieID,
    M.Title AS Title,
    M.Category AS Genre,
    M.ReleaseDate AS ReleaseDate,
    MA.Role AS Role
FROM Actors A
JOIN MovieActors MA ON A.Actor_id = MA.Actor_id
JOIN Movies M ON MA.Movie_id = M.Movie_id;
```



```
mysql> SELECT
    ->     A.Actor_id AS ActorID,
    ->     A.Name AS Name,
    ->     A.City AS City,
    ->     A.DOB AS DateOfBirth,
    ->     M.Movie_id AS MovieID,
    ->     M.Title AS Title,
    ->     M.Category AS Genre,
    ->     M.ReleaseDate AS ReleaseDate,
    ->     MA.Role AS Role
    -> FROM Actors A
    -> JOIN MovieActors MA ON A.Actor_id = MA.Actor_id
    -> JOIN Movies M ON MA.Movie_id = M.Movie_id;
+---------+--------------------+-------------+-------------+---------+-----------------+--------+-------------+---------------+
| ActorID | Name               | City        | DateOfBirth | MovieID | Title           | Genre  | ReleaseDate | Role          |
+---------+--------------------+-------------+-------------+---------+-----------------+--------+-------------+---------------+
| A1      | Millie Bobby Brown | Los Angeles | 2004-02-19  | M1      | Stranger Things | Sci-Fi | 2016-07-15  | Eleven        |
| A2      | Bryan Cranston     | Hollywood   | 1956-03-07  | M2      | Breaking Bad    | Drama  | 2008-01-20  | Walter White  |
| A3      | Winona Ryder       | New York    | 1971-10-29  | M1      | Stranger Things | Sci-Fi | 2016-07-15  | Joyce Byers   |
| A4      | Aaron Paul         | Boise       | 1979-08-27  | M2      | Breaking Bad    | Drama  | 2008-01-20  | Jesse Pinkman |
| A5      | David Harbour      | Los Angeles | 1975-04-10  | M1      | Stranger Things | Sci-Fi | 2016-07-15  | Jim Hopper    |
+---------+--------------------+-------------+-------------+---------+-----------------+--------+-------------+---------------+
5 rows in set (0.10 sec)
```

4.3 Exports all data to group actors from a specific city, with average age (per city)

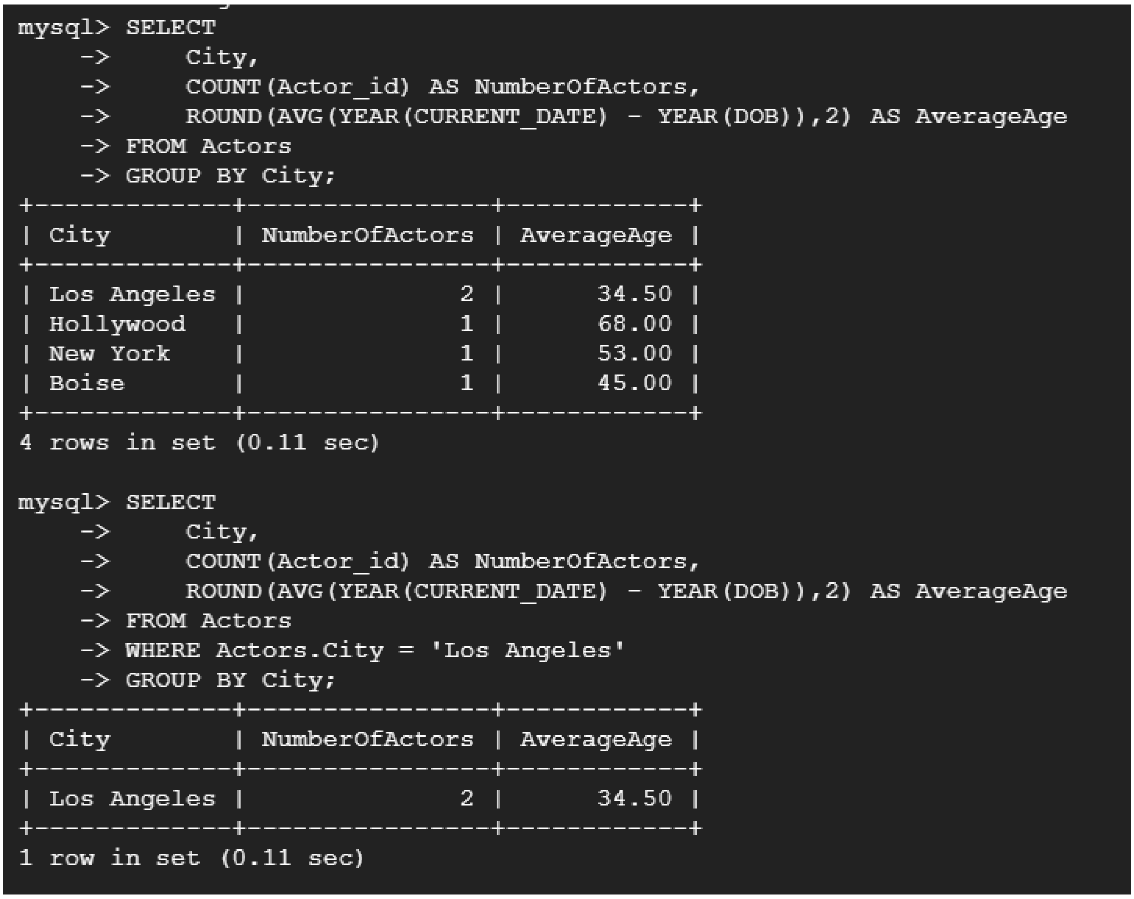
For all cities:

```
SELECT
    City,
    COUNT(Actor_id) AS NumberOfActors,
    ROUND(AVG(YEAR(CURRENT_DATE) - YEAR(DOB)),2) AS AverageAge
FROM Actors
GROUP BY City;
```

For a specific city, in this example 'Los Angeles':

```
SELECT
    City,
    COUNT(Actor_id) AS NumberOfActors,
    ROUND(AVG(YEAR(CURRENT_DATE) - YEAR(DOB)),2) AS AverageAge
FROM Actors
WHERE Actors.City = 'Los Angeles'
GROUP BY City;
```

An Age function (admin, 2024) is created to calculate the average date from the date of birth(`DOB`) attribute in `Actors` table. The ages are rounded to 2 decimal places.



```
mysql> SELECT
    ->     City,
    ->     COUNT(Actor_id) AS NumberOfActors,
    ->     ROUND(AVG(YEAR(CURRENT_DATE) - YEAR(DOB)),2) AS AverageAge
    -> FROM Actors
    -> GROUP BY City;
+-------------+----------------+------------+
| City        | NumberOfActors | AverageAge |
+-------------+----------------+------------+
| Los Angeles |              2 |      34.50 |
| Hollywood   |              1 |      68.00 |
| New York    |              1 |      53.00 |
| Boise       |              1 |      45.00 |
+-------------+----------------+------------+
4 rows in set (0.11 sec)

mysql> SELECT
    ->     City,
    ->     COUNT(Actor_id) AS NumberOfActors,
    ->     ROUND(AVG(YEAR(CURRENT_DATE) - YEAR(DOB)),2) AS AverageAge
    -> FROM Actors
    -> WHERE Actors.City = 'Los Angeles'
    -> GROUP BY City;
+-------------+----------------+------------+
| City        | NumberOfActors | AverageAge |
+-------------+----------------+------------+
| Los Angeles |              2 |      34.50 |
+-------------+----------------+------------+
1 row in set (0.11 sec)
```

4.4 Exports all data to show the favourite comedy movies for a specific user

Queries for all users with favourite comedy movie:

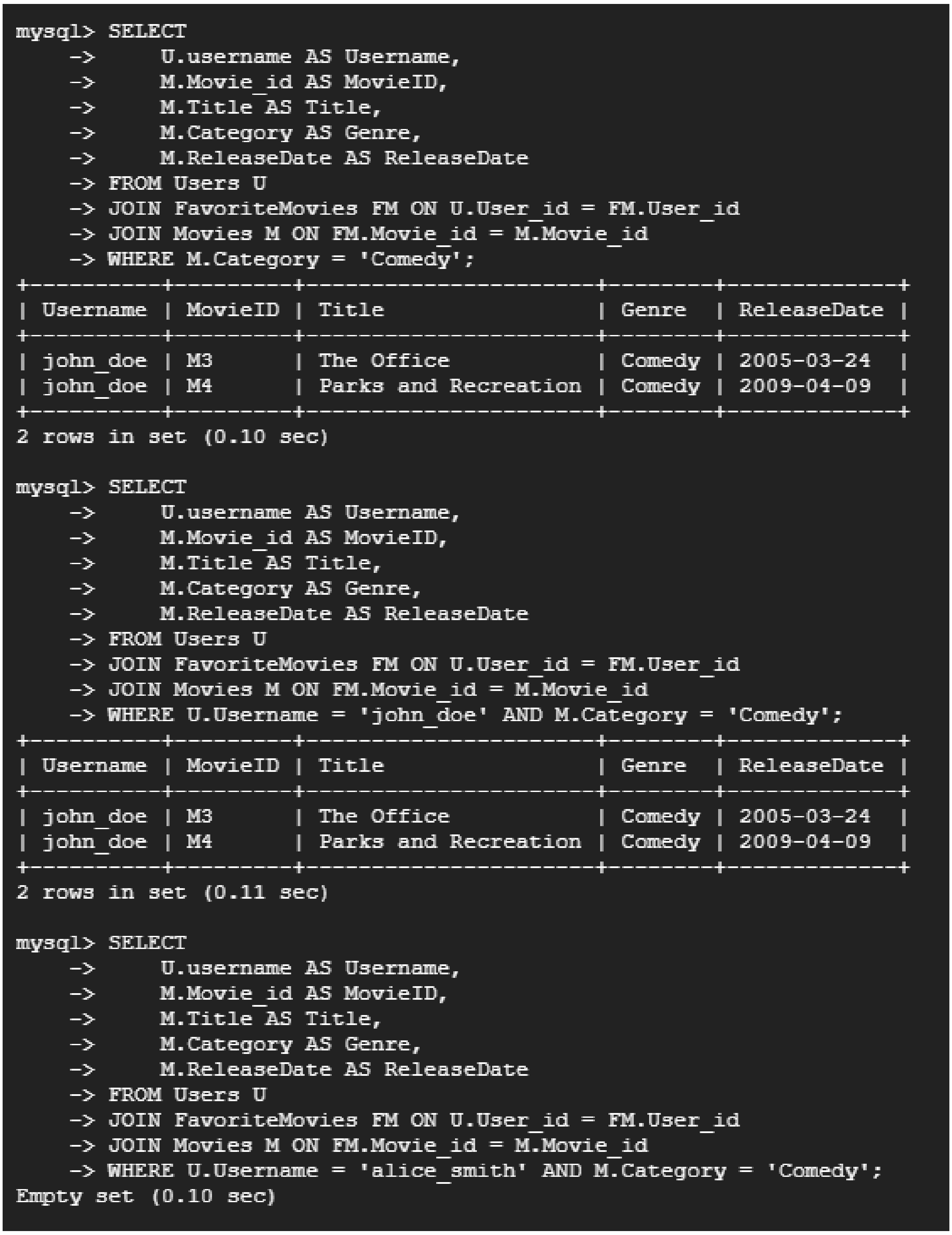
```
SELECT
    U.username AS Username,
    M.Movie_id AS MovieID,
    M.Title AS Title,
    M.Category AS Genre,
    M.ReleaseDate AS ReleaseDate
FROM Users U
JOIN FavoriteMovies FM ON U.User_id = FM.User_id
JOIN Movies M ON FM.Movie_id = M.Movie_id
WHERE M.Category = 'Comedy';
```

Queries for a specific user with favourite comedy movie:

```
SELECT
    U.username AS Username,
    M.Movie_id AS MovieID,
    M.Title AS Title,
    M.Category AS Genre,
    M.ReleaseDate AS ReleaseDate
FROM Users U
JOIN FavoriteMovies FM ON U.User_id = FM.User_id
JOIN Movies M ON FM.Movie_id = M.Movie_id
WHERE U.Username = 'john_doe' AND M.Category = 'Comedy';
```

Queries for a specific user with favourite comedy movie. This result will return empty because the specific user does not have a favourite comedy movie. :

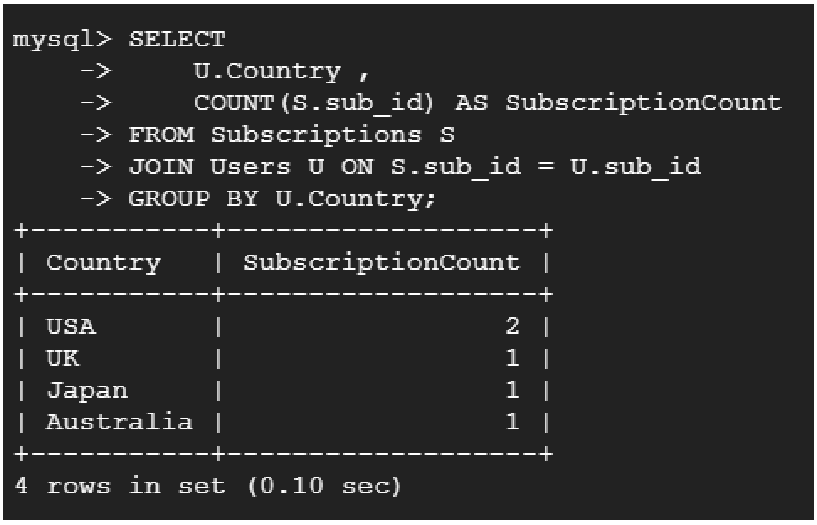
```
SELECT
    U.username AS Username,
    M.Movie_id AS MovieID,
    M.Title AS Title,
    M.Category AS Genre,
    M.ReleaseDate AS ReleaseDate
FROM Users U
JOIN FavoriteMovies FM ON U.User_id = FM.User_id
JOIN Movies M ON FM.Movie_id = M.Movie_id
WHERE U.Username = 'alice_smith' AND M.Category = 'Comedy';
```



```
mysql> SELECT
    ->     U.username AS Username,
    ->     M.Movie_id AS MovieID,
    ->     M.Title AS Title,
    ->     M.Category AS Genre,
    ->     M.ReleaseDate AS ReleaseDate
    -> FROM Users U
    -> JOIN FavoriteMovies FM ON U.User_id = FM.User_id
    -> JOIN Movies M ON FM.Movie_id = M.Movie_id
    -> WHERE M.Category = 'Comedy';
+----------+---------+----------------------+--------+-------------+
| Username | MovieID | Title                | Genre  | ReleaseDate |
+----------+---------+----------------------+--------+-------------+
| john_doe | M3      | The Office           | Comedy | 2005-03-24  |
| john_doe | M4      | Parks and Recreation | Comedy | 2009-04-09  |
+----------+---------+----------------------+--------+-------------+
2 rows in set (0.10 sec)

mysql> SELECT
    ->     U.username AS Username,
    ->     M.Movie_id AS MovieID,
    ->     M.Title AS Title,
    ->     M.Category AS Genre,
    ->     M.ReleaseDate AS ReleaseDate
    -> FROM Users U
    -> JOIN FavoriteMovies FM ON U.User_id = FM.User_id
    -> JOIN Movies M ON FM.Movie_id = M.Movie_id
    -> WHERE U.Username = 'john_doe' AND M.Category = 'Comedy';
+----------+---------+----------------------+--------+-------------+
| Username | MovieID | Title                | Genre  | ReleaseDate |
+----------+---------+----------------------+--------+-------------+
| john_doe | M3      | The Office           | Comedy | 2005-03-24  |
| john_doe | M4      | Parks and Recreation | Comedy | 2009-04-09  |
+----------+---------+----------------------+--------+-------------+
2 rows in set (0.11 sec)

mysql> SELECT
    ->     U.username AS Username,
    ->     M.Movie_id AS MovieID,
    ->     M.Title AS Title,
    ->     M.Category AS Genre,
    ->     M.ReleaseDate AS ReleaseDate
    -> FROM Users U
    -> JOIN FavoriteMovies FM ON U.User_id = FM.User_id
    -> JOIN Movies M ON FM.Movie_id = M.Movie_id
    -> WHERE U.Username = 'alice_smith' AND M.Category = 'Comedy';
Empty set (0.10 sec)
```

4.5 Exports all data to count how many subscriptions are in the database per country

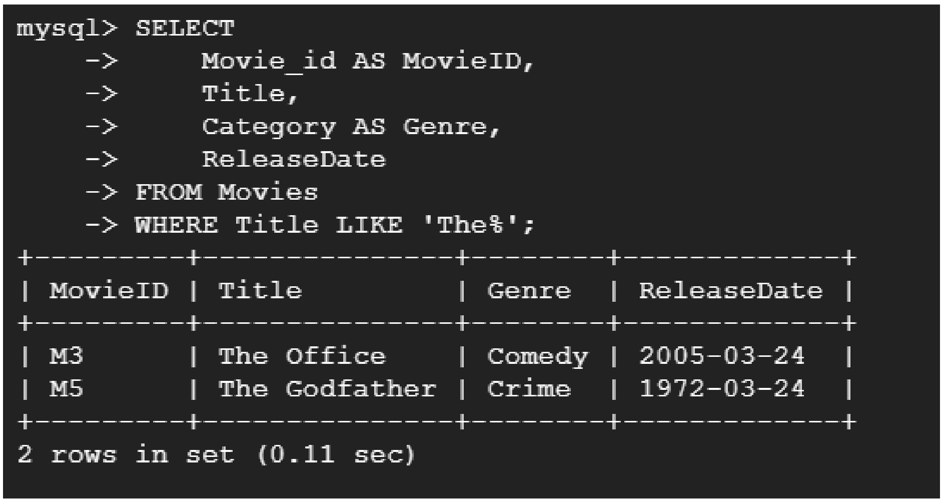
```
SELECT
    U.Country ,
    COUNT(S.sub_id) AS SubscriptionCount
FROM Subscriptions S
JOIN Users U ON S.sub_id = U.sub_id
GROUP BY U.Country;
```



```
mysql> SELECT
    ->     U.Country ,
    ->     COUNT(S.sub_id) AS SubscriptionCount
    -> FROM Subscriptions S
    -> JOIN Users U ON S.sub_id = U.sub_id
    -> GROUP BY U.Country;
+-----------+-------------------+
| Country   | SubscriptionCount |
+-----------+-------------------+
| USA       |                 2 |
| UK        |                 1 |
| Japan     |                 1 |
| Australia |                 1 |
+-----------+-------------------+
4 rows in set (0.10 sec)
```

4.6 Exports all data to find the movies that start with the keyword The

```
SELECT
    Movie_id AS MovieID,
    Title,
    Category AS Genre,
    ReleaseDate
FROM Movies
WHERE Title LIKE 'The%';
```

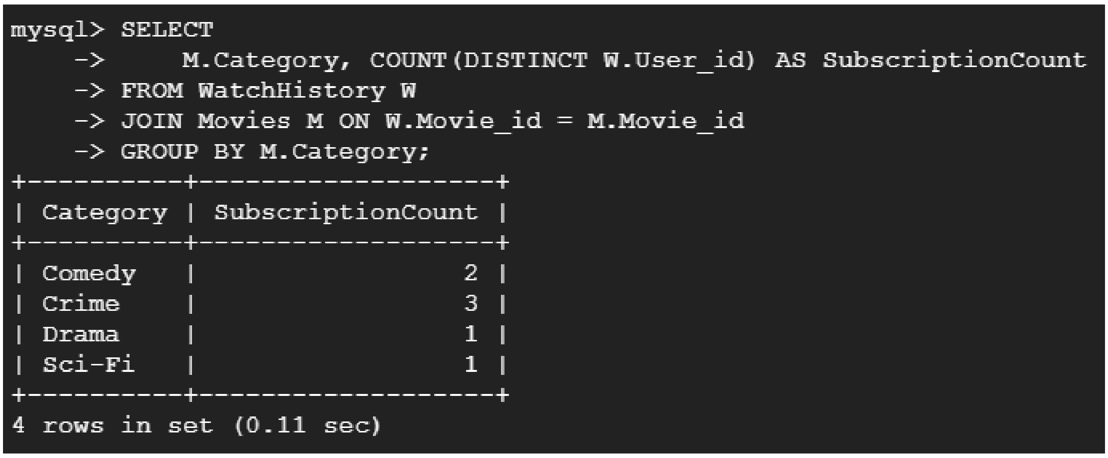


```
mysql> SELECT
    ->     Movie_id AS MovieID,
    ->     Title,
    ->     Category AS Genre,
    ->     ReleaseDate
    -> FROM Movies
    -> WHERE Title LIKE 'The%';
+---------+---------------+--------+-------------+
| MovieID | Title         | Genre  | ReleaseDate |
+---------+---------------+--------+-------------+
| M3      | The Office    | Comedy | 2005-03-24  |
| M5      | The Godfather | Crime  | 1972-03-24  |
+---------+---------------+--------+-------------+
2 rows in set (0.11 sec)
```

4.7 Exports data to find the number of subscriptions per movie category

Shows the number of subscribed users who have watched movies from each category:

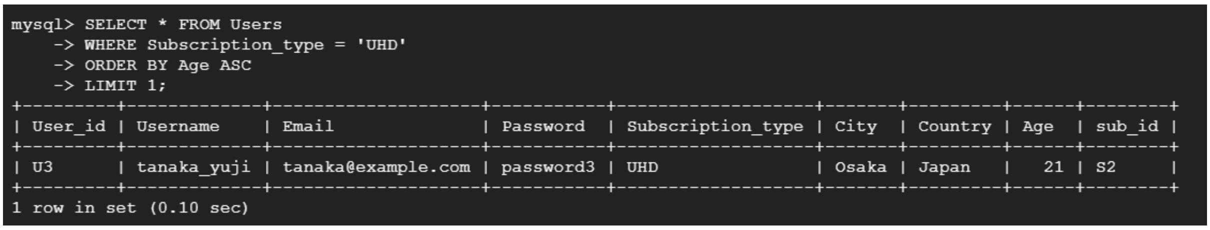
```
SELECT
    M.Category, COUNT(DISTINCT W.User_id) AS SubscriptionCount
FROM WatchHistory W
JOIN Movies M ON W.Movie_id = M.Movie_id
GROUP BY M.Category;
```



```
mysql> SELECT
    ->     M.Category, COUNT(DISTINCT W.User_id) AS SubscriptionCount
    -> FROM WatchHistory W
    -> JOIN Movies M ON W.Movie_id = M.Movie_id
    -> GROUP BY M.Category;
+----------+-------------------+
| Category | SubscriptionCount |
+----------+-------------------+
| Comedy   |                 2 |
| Crime    |                 3 |
| Drama    |                 1 |
| Sci-Fi   |                 1 |
+----------+-------------------+
4 rows in set (0.11 sec)
```

4.8 Exports data to find the username and the city of the youngest customer in the UHD subscription category

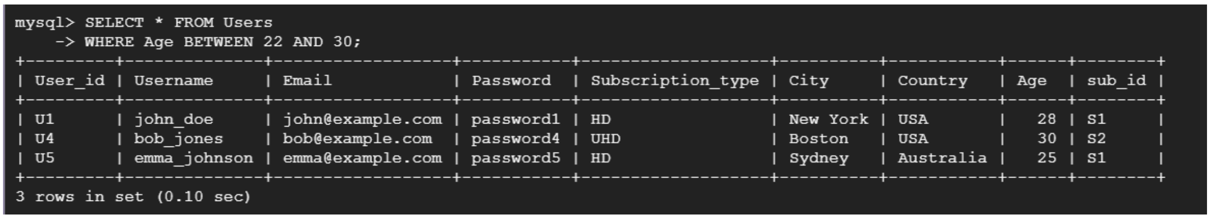
```
SELECT * FROM Users
WHERE Subscription_type = 'UHD'
ORDER BY Age ASC
LIMIT 1;
```



```
mysql> SELECT * FROM Users
    -> WHERE Subscription_type = 'UHD'
    -> ORDER BY Age ASC
    -> LIMIT 1;
+---------+-------------+--------------------+-----------+-------------------+-------+---------+------+--------+
| User_id | Username    | Email              | Password  | Subscription_type | City  | Country | Age  | sub_id |
+---------+-------------+--------------------+-----------+-------------------+-------+---------+------+--------+
| U3      | tanaka_yuji | tanaka@example.com | password3 | UHD               | Osaka | Japan   |   21 | S2     |
+---------+-------------+--------------------+-----------+-------------------+-------+---------+------+--------+
1 row in set (0.10 sec)
```

4.9 Exports data to find users between 22 - 30 years old inclusive

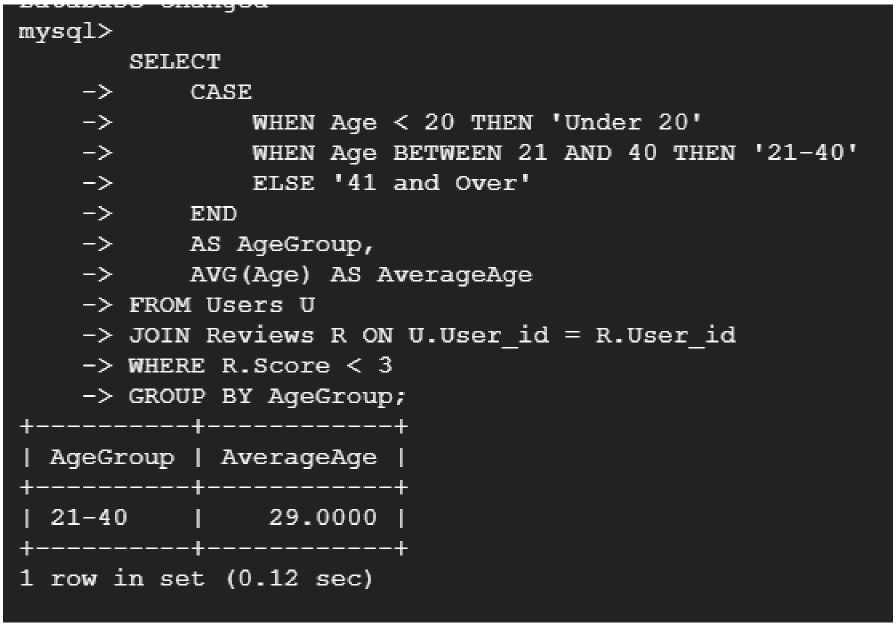
```
SELECT * FROM Users
WHERE Age BETWEEN 22 AND 30;
```



```
mysql> SELECT * FROM Users
    -> WHERE Age BETWEEN 22 AND 30;
+---------+--------------+-------------------+-----------+-------------------+----------+-----------+------+--------+
| User_id | Username     | Email             | Password  | Subscription_type | City     | Country   | Age  | sub_id |
+---------+--------------+-------------------+-----------+-------------------+----------+-----------+------+--------+
| U1      | john_doe     | john@example.com  | password1 | HD                | New York | USA       |   28 | S1     |
| U4      | bob_jones    | bob@example.com   | password4 | UHD               | Boston   | USA       |   30 | S2     |
| U5      | emma_johnson | emma@example.com  | password5 | HD                | Sydney   | Australia |   25 | S1     |
+---------+--------------+-------------------+-----------+-------------------+----------+-----------+------+--------+
3 rows in set (0.10 sec)
```

4.10 Exports data to find the average age of users with low score reviews (less than 3)

```
SELECT
    CASE
        WHEN Age < 20 THEN 'Under 20'
        WHEN Age BETWEEN 21 AND 40 THEN '21-40'
        ELSE '41 and Over'
    END
    AS AgeGroup,
    AVG(age) AS AverageAge
FROM Users U
JOIN Reviews R ON U.user_id = R.user_id
WHERE R.score < 3
GROUP BY AgeGroup;
```



```
mysql>
       SELECT
    ->     CASE
    ->         WHEN Age < 20 THEN 'Under 20'
    ->         WHEN Age BETWEEN 21 AND 40 THEN '21-40'
    ->         ELSE '41 and Over'
    ->     END
    ->     AS AgeGroup,
    ->     AVG(Age) AS AverageAge
    -> FROM Users U
    -> JOIN Reviews R ON U.User_id = R.User_id
    -> WHERE R.Score < 3
    -> GROUP BY AgeGroup;
+----------+------------+
| AgeGroup | AverageAge |
+----------+------------+
| 21-40    |    29.0000 |
+----------+------------+
1 row in set (0.12 sec)
```

This query only returns 1 record, because all the users in `Users` table are between 21 and 40. There is no users below 21 or above 40.

Task 5 Python scripts

The python file `mininet.py` is included in the zip folder as the report.

Imports

```
import mysql.connector
from mysql.connector import Error
from prettytable import PrettyTable
```

Connection to Database mininet_db

```
def create_connection(host_name, database_name, user_name, user_password):
    connection = None
    try:
        connection = mysql.connector.connect(
            host=host_name,
            database=database_name,
            user=user_name,
            passwd=user_password
        )
        print("MySQL Database connection successful")
    except Error as err:
        print(f"Error: '{err}'")

    return connection
```

Query execution

```
def create_table(results, cursor):
    """will return the executed queries in tabular form"""
    table = PrettyTable()
    table.field_names = [column[0] for column in cursor.description]  # fethces column names from column description (mySQL, n.d.)
    for row in results:
        table.add_row(row)  # Adds each row of the query results to the table
    return table

def execute_query(connection, query, params=None):
    cursor = connection.cursor()
    try:
        cursor.execute(query, params)
        results = cursor.fetchall()
        table = create_table(results, cursor)  # Create the table
        print(table)
        print("Query successful")
        cursor.close()
    except Error as err:
        print(f"Error: '{err}'")
```

Query 1. Exports all data about users in the HD or UHD subscriptions

This query allows to choose to export user data for either HD or UHD subscribers. ‘%s’ in Subscription_type = %s acts as a placeholder that allows to insert variables into the query. This input placeholder is also used in query 3 and 4.

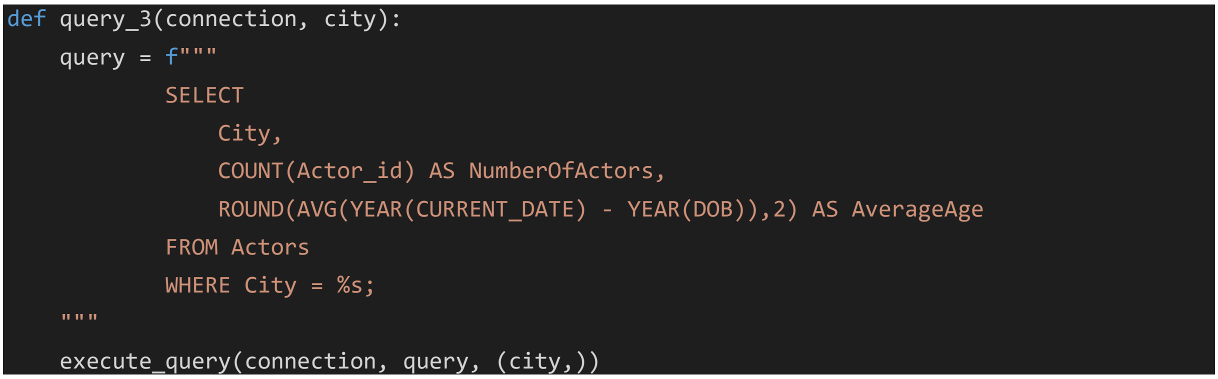
```
def query_1(connection, sub_type):
    query = f"""
            SELECT *
            FROM Users U
            WHERE Subscription_type = %s;
    """
    execute_query(connection, query, (sub_type,))
```

Query 2. Exports all data about actors and their associated movies

```
def query_2(connection):
    query = f"""
            SELECT
                A.Actor_id AS ActorID,
                A.Name AS Name,
                A.City AS City,
                A.DOB AS DateOfBirth,
                M.Movie_id AS MovieID,
                M.Title AS Title,
                M.Category AS Genre,
                M.ReleaseDate AS ReleaseDate,
                MA.Role AS Role
            FROM Actors A
            JOIN MovieActors MA ON A.Actor_id = MA.Actor_id
            JOIN Movies M ON MA.Movie_id = M.Movie_id;
    """
    execute_query(connection, query)
```

Query 3 Exports data to group actors from specific city, with average age

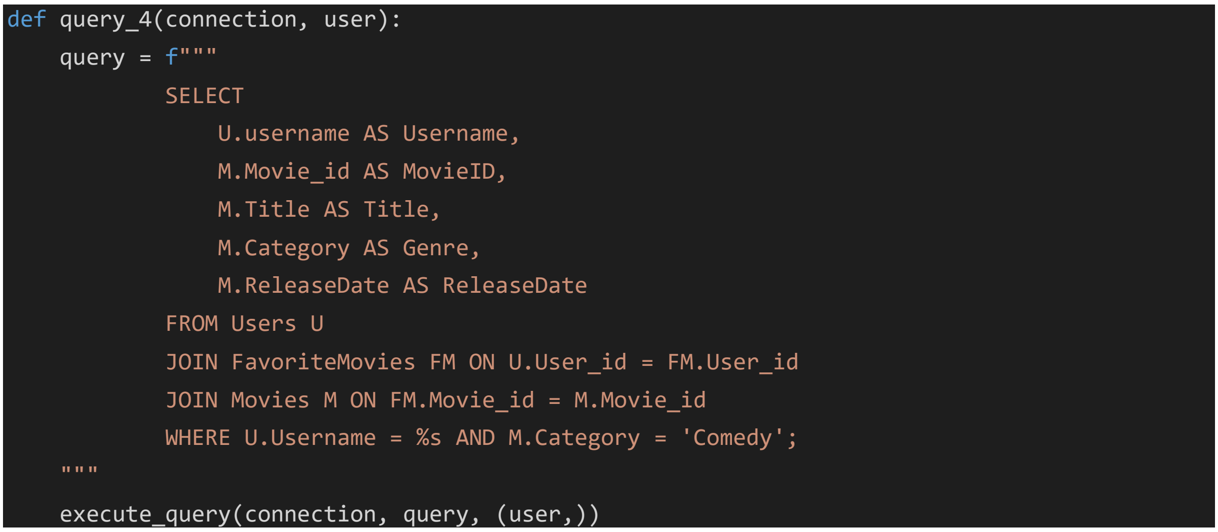
This query allows to insert a desired value of City. Based on this value, the query will export the group of actors belonging to that city. .



```
def query_3(connection, city):
    query = f"""
            SELECT
                City,
                COUNT(Actor_id) AS NumberOfActors,
                ROUND(AVG(YEAR(CURRENT_DATE) - YEAR(DOB)),2) AS AverageAge
            FROM Actors
            WHERE City = %s;
    """
    execute_query(connection, query, (city,))
```

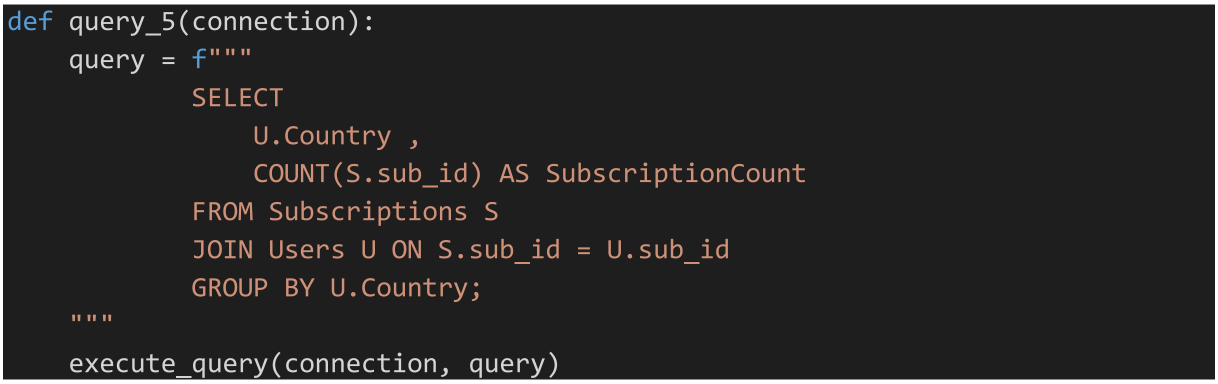
Query 4 Exports all data to show favourite comedy movies for specific user

Here, a username can be specified to export all the favourite comedy movies of that user.



```
def query_4(connection, user):
    query = f"""
            SELECT
                U.username AS Username,
                M.Movie_id AS MovieID,
                M.Title AS Title,
                M.Category AS Genre,
                M.ReleaseDate AS ReleaseDate
            FROM Users U
            JOIN FavoriteMovies FM ON U.User_id = FM.User_id
            JOIN Movies M ON FM.Movie_id = M.Movie_id
            WHERE U.Username = %s AND M.Category = 'Comedy';
    """
    execute_query(connection, query, (user,))
```

Query 5 Exports all data to count how many subscriptions are in the database per country



```
def query_5(connection):
    query = f"""
            SELECT
                U.Country ,
                COUNT(S.sub_id) AS SubscriptionCount
            FROM Subscriptions S
            JOIN Users U ON S.sub_id = U.sub_id
            GROUP BY U.Country;
    """
    execute_query(connection, query)
```

Main function

```
def main():
    host = "34.171.50.39" # host IP address for pwd-mininet-sql from GCP
    database = "mininet_db" # database name
    user = "root" # default user
    password = "123456789" # user password
    connection = create_connection(host, database, user, password) # establishes connection

    if connection:
        try:
            while True:
                print("\nSelect a query to run:")
                print("1. Export all data about users in the HD or UHD subscriptions.")
                print("2. Export all data about actors and their associated movies.")
                print("3. Export all data to group actors from a specific city, showing also the average age (per city).")
                print("4. Export all data to show the favourite comedy movies for a specific user.")
                print("5. Export all data to count how many subscriptions are in the database per country.")
                print("e. Exit")
                choice = input("Enter your choice: ") # user input to choose query (1-5)

                if choice == '1':
                    sub_type = input("Enter the subscription (HD/UHD) of your choice: ")
                    query_1(connection, sub_type)
                elif choice == '2':
                    query_2(connection)
                elif choice == '3':
                    city = input("Enter the specific city of your choice: ")
                    query_3(connection, city)
                elif choice == '4':
                    user = input("Enter the specific user of your choice: ")
                    query_4(connection, user)
                elif choice == '5':
                    query_5(connection)
                elif choice == 'e':
                    break
                else:
                    print("Invalid choice!")

        except Error as e:
            print("Connection Not Available.")

if __name__ == "__main__":
    main()
```

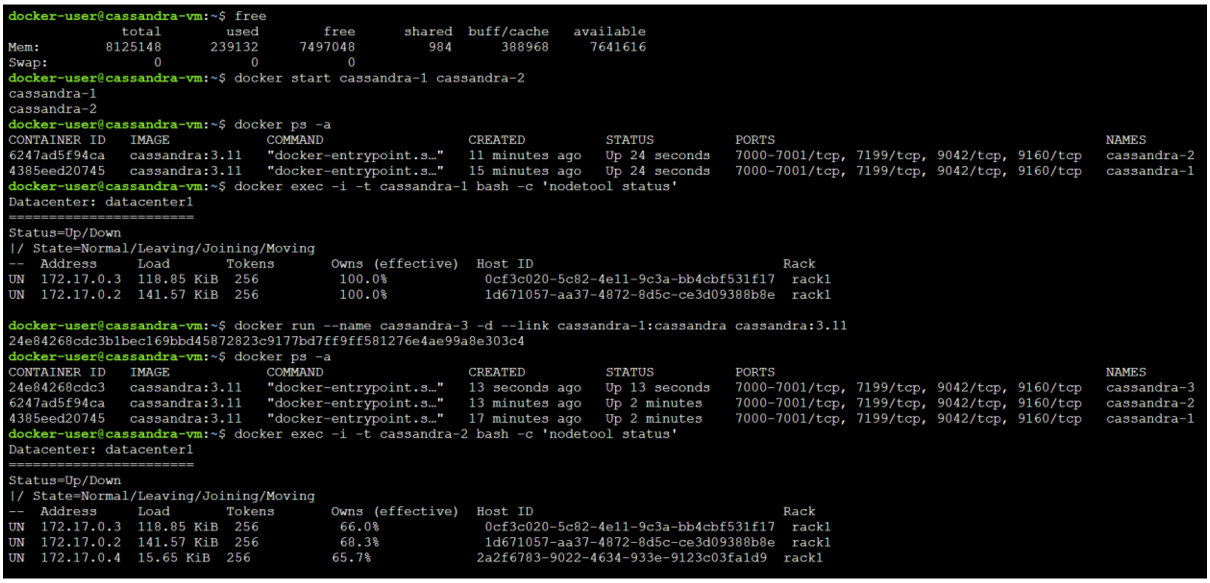
In the main function, all the queries are executed using if-else statements. When the python file is run, it shows if the connection is successful or not. Only when it successfully connects to `mininet_db` database, it goes into a while loop, that allows users (python application user, not a user from Users table!) to request a query. In some cases, it allows further user requests for input values to match the query. As seen earlier, it enables users to request with an input value for a specified key in the WHERE clause of the query to filter the output.

Pip Installs

```
pip install mysql-connector-python

pip install prettytable
```

Task 6 CQL and Python in Apache Cassandra

Cassandra containers



```
docker-user@cassandra-vm:~$ free
              total        used        free      shared  buff/cache   available
Mem:        8125148      239132     7497048         984      388968     7641616
Swap:             0           0           0
docker-user@cassandra-vm:~$ docker start cassandra-1 cassandra-2
cassandra-1
cassandra-2
docker-user@cassandra-vm:~$ docker ps -a
CONTAINER ID   IMAGE            COMMAND                  CREATED          STATUS          PORTS                                         NAMES
6247ad5f94ca   cassandra:3.11   "docker-entrypoint.s…"   11 minutes ago   Up 24 seconds   7000-7001/tcp, 7199/tcp, 9042/tcp, 9160/tcp   cassandra-2
4385eed20745   cassandra:3.11   "docker-entrypoint.s…"   15 minutes ago   Up 24 seconds   7000-7001/tcp, 7199/tcp, 9042/tcp, 9160/tcp   cassandra-1
docker-user@cassandra-vm:~$ docker exec -i -t cassandra-1 bash -c 'nodetool status'
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.17.0.3  118.85 KiB  256          100.0%            0cf3c020-5c82-4e11-9c3a-bb4cbf531f17  rack1
UN  172.17.0.2  141.57 KiB  256          100.0%            1d671057-aa37-4872-8d5c-ce3d09388b8e  rack1

docker-user@cassandra-vm:~$ docker run --name cassandra-3 -d --link cassandra-1:cassandra cassandra:3.11
24e84268cdc3b1bec169bbd45872823c9177bd7ff9ff581276e4ae99a8e303c4
docker-user@cassandra-vm:~$ docker ps -a
CONTAINER ID   IMAGE            COMMAND                  CREATED          STATUS          PORTS                                         NAMES
24e84268cdc3   cassandra:3.11   "docker-entrypoint.s…"   13 seconds ago   Up 13 seconds   7000-7001/tcp, 7199/tcp, 9042/tcp, 9160/tcp   cassandra-3
6247ad5f94ca   cassandra:3.11   "docker-entrypoint.s…"   13 minutes ago   Up 2 minutes    7000-7001/tcp, 7199/tcp, 9042/tcp, 9160/tcp   cassandra-2
4385eed20745   cassandra:3.11   "docker-entrypoint.s…"   17 minutes ago   Up 2 minutes    7000-7001/tcp, 7199/tcp, 9042/tcp, 9160/tcp   cassandra-1
docker-user@cassandra-vm:~$ docker exec -i -t cassandra-2 bash -c 'nodetool status'
Datacenter: datacenter1
=======================
Status=Up/Down
|/ State=Normal/Leaving/Joining/Moving
--  Address     Load       Tokens       Owns (effective)  Host ID                               Rack
UN  172.17.0.3  118.85 KiB  256          66.0%             0cf3c020-5c82-4e11-9c3a-bb4cbf531f17  rack1
UN  172.17.0.2  141.57 KiB  256          68.3%             1d671057-aa37-4872-8d5c-ce3d09388b8e  rack1
UN  172.17.0.4  15.65 KiB  256          65.7%             2a2f6783-9022-4634-933e-9123c03fa1d9  rack1
```

Image 1: Running containers in the cluster

Image 1 shows the activation three nodes in the cluster. Although the nodes are in three different servers, they are all residing in the same rack1. Among the nodes, CQL queries are executed in Cassandra-1, and Python application is connected to the entire cluster (steliosot, 2024).

Creates mininet keyspace in cql

```
CREATE KEYSPACE mininet
  WITH REPLICATION = {
   'class' : 'SimpleStrategy',
   'replication_factor' : 3
  };

USE mininet;
```

6.1. Create Users table

```
CREATE TABLE mininet.Users (
   User_id UUID,
   Username varchar,
   Email varchar,
   Password varchar,
   Subscription_type varchar,
   City varchar,
   Country varchar,
   Age int,
   PRIMARY KEY (User_id)
);
```

In this case, `User_id` acts as both primary key and partition key. This makes `User_id` unique across the entire `Users` table and will be distributed across all three nodes within the cluster. These user ids are generated automatically using Universally Unique Identifiers (UUID), instead of manually inserting like in task 2.

Unlike SQL, in CQL, the datatype for age is more general datatype int (DataStax3, 2024).

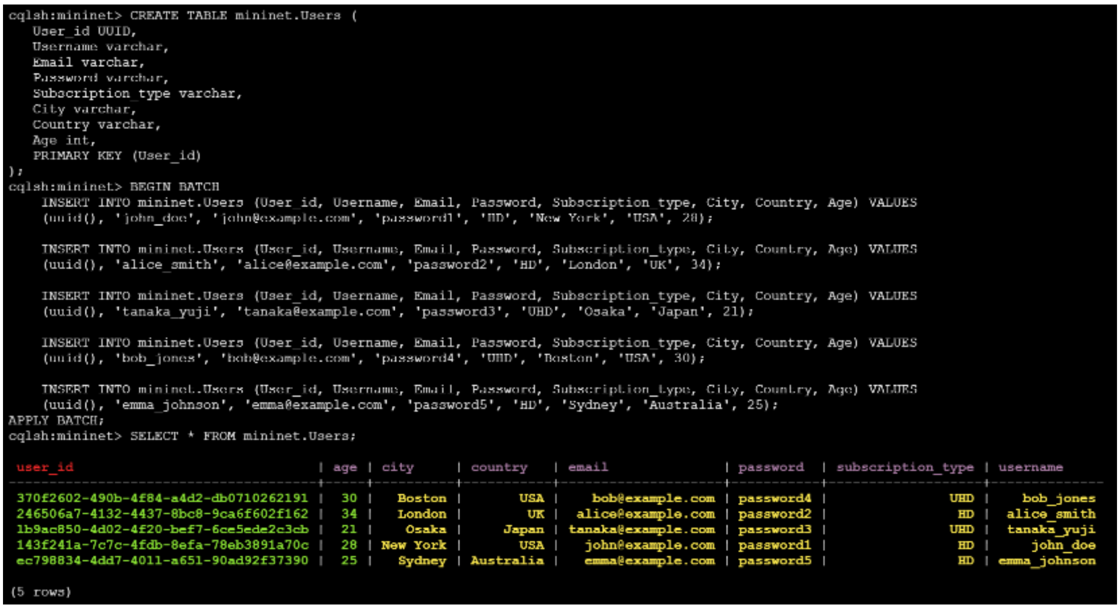
Insert records to User Table

```
BEGIN BATCH
    INSERT INTO mininet.Users (User_id, Username, Email, Password,
Subscription_type, City, Country, Age) VALUES
    (uuid(), 'john_doe', 'john@example.com', 'password1', 'HD', 'New York', 'USA',
28);

    INSERT INTO mininet.Users (User_id, Username, Email, Password,
Subscription_type, City, Country, Age) VALUES
    (uuid(), 'alice_smith', 'alice@example.com', 'password2', 'HD', 'London', 'UK',
34);

    INSERT INTO mininet.Users (User_id, Username, Email, Password,
Subscription_type, City, Country, Age) VALUES
    (uuid(), 'tanaka_yuji', 'tanaka@example.com', 'password3', 'UHD', 'Osaka',
'Japan', 21);

    INSERT INTO mininet.Users (User_id, Username, Email, Password,
Subscription_type, City, Country, Age) VALUES
    (uuid(), 'bob_jones', 'bob@example.com', 'password4', 'UHD', 'Boston', 'USA',
30);

    INSERT INTO mininet.Users (User_id, Username, Email, Password,
Subscription_type, City, Country, Age) VALUES
    (uuid(), 'emma_johnson', 'emma@example.com', 'password5', 'HD', 'Sydney',
'Australia', 25);
APPLY BATCH;
```

A BATCH is used to ensure atomicity while writing the records into the `Users` table. `Users` is single partition because all its rows are related with single partition key User_id. In single partition scenarios, atomicity ensures that either all the records are inserted to the table, or no data will be inserted (datastax, n.d.).

6.2 Export All Users

```
SELECT * FROM mininet.Users;
```



```
cqlsh:mininet> CREATE TABLE mininet.Users (
   User_id UUID,
   Username varchar,
   Email varchar,
   Password varchar,
   Subscription_type varchar,
   City varchar,
   Country varchar,
   Age int,
   PRIMARY KEY (User_id)
);
cqlsh:mininet> BEGIN BATCH
    INSERT INTO mininet.Users (User_id, Username, Email, Password, Subscription_type, City, Country, Age) VALUES
    (uuid(), 'john_doe', 'john@example.com', 'password1', 'HD', 'New York', 'USA', 28);

    INSERT INTO mininet.Users (User_id, Username, Email, Password, Subscription_type, City, Country, Age) VALUES
    (uuid(), 'alice_smith', 'alice@example.com', 'password2', 'HD', 'London', 'UK', 34);

    INSERT INTO mininet.Users (User_id, Username, Email, Password, Subscription_type, City, Country, Age) VALUES
    (uuid(), 'tanaka_yuji', 'tanaka@example.com', 'password3', 'UHD', 'Osaka', 'Japan', 21);

    INSERT INTO mininet.Users (User_id, Username, Email, Password, Subscription_type, City, Country, Age) VALUES
    (uuid(), 'bob_jones', 'bob@example.com', 'password4', 'UHD', 'Boston', 'USA', 30);

    INSERT INTO mininet.Users (User_id, Username, Email, Password, Subscription_type, City, Country, Age) VALUES
    (uuid(), 'emma_johnson', 'emma@example.com', 'password5', 'HD', 'Sydney', 'Australia', 25);
APPLY BATCH;
cqlsh:mininet> SELECT * FROM mininet.Users;

 user_id                              | age | city     | country   | email              | password  | subscription_type | username
--------------------------------------+-----+----------+-----------+--------------------+-----------+-------------------+--------------
 370f2602-490b-4f84-a4d2-db0710262191 |  30 |   Boston |       USA |    bob@example.com | password4 |               UHD |    bob_jones
 246506a7-4132-4437-8bc8-9ca6f602f162 |  34 |   London |        UK |  alice@example.com | password2 |                HD |  alice_smith
 1b9ac850-4d02-4f20-bef7-6ce5ede2c3cb |  21 |    Osaka |     Japan | tanaka@example.com | password3 |               UHD |  tanaka_yuji
 143f241a-7c7c-4fdb-8efa-78eb3891a70c |  28 | New York |       USA |   john@example.com | password1 |                HD |     john_doe
 ec798834-4dd7-4011-a651-90ad92f37390 |  25 |   Sydney | Australia |   emma@example.com | password5 |                HD | emma_johnson

(5 rows)
```

6.3 Export all users from a specific country

```
CREATE INDEX ON mininet.Users (Country);
```

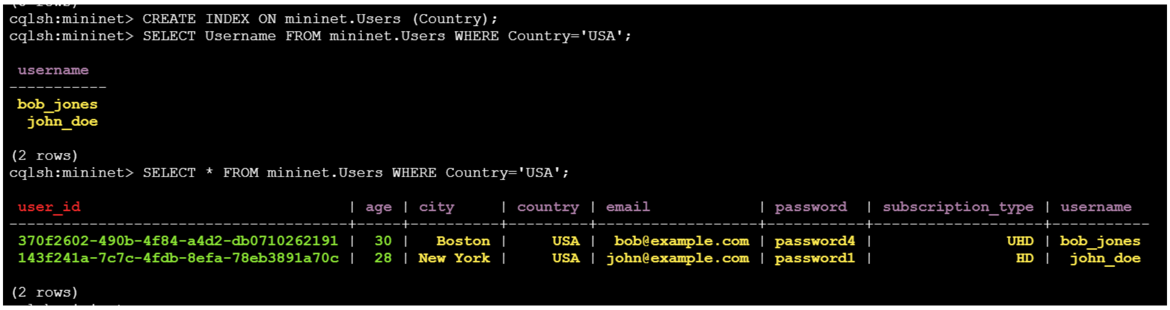
To search from the City column, an index is created which allows to search inside a key for a particular value. In this case, only the users belonging to Country(key)=USA(value) will be exported.

```
SELECT Username FROM mininet.Users WHERE Country='USA';
```

The script above only returns the usernames from USA.

```
SELECT * FROM mininet.Users WHERE Country='USA';
```

Returns the entire data of users from USA.

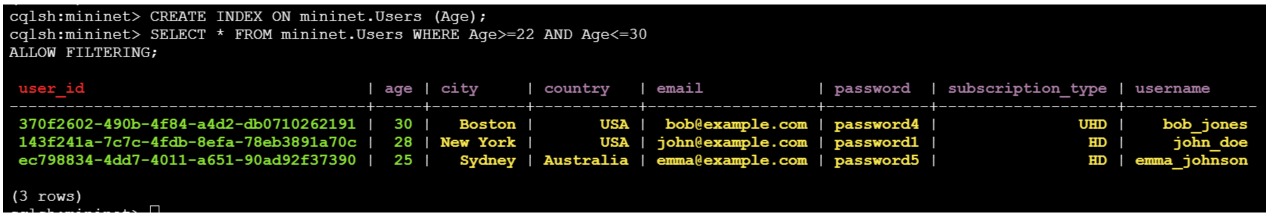


```
cqlsh:mininet> CREATE INDEX ON mininet.Users (Country);
cqlsh:mininet> SELECT Username FROM mininet.Users WHERE Country='USA';

 username
-----------
 bob_jones
  john_doe

(2 rows)
cqlsh:mininet> SELECT * FROM mininet.Users WHERE Country='USA';

 user_id                              | age | city     | country | email            | password  | subscription_type | username
--------------------------------------+-----+----------+---------+------------------+-----------+-------------------+-----------
 370f2602-490b-4f84-a4d2-db0710262191 |  30 |   Boston |     USA |  bob@example.com | password4 |               UHD | bob_jones
 143f241a-7c7c-4fdb-8efa-78eb3891a70c |  28 | New York |     USA | john@example.com | password1 |                HD |  john_doe

(2 rows)
```

6.4 Export data to find users between 22-30 years old (including 22 and 30)

```
CREATE INDEX ON mininet.Users (Age);

SELECT * FROM mininet.Users WHERE Age>=22 AND Age<=30
ALLOW FILTERING;
```

In this case, the index is created on Age to allow filtering using Age key and a specified value, from 22 to 30.



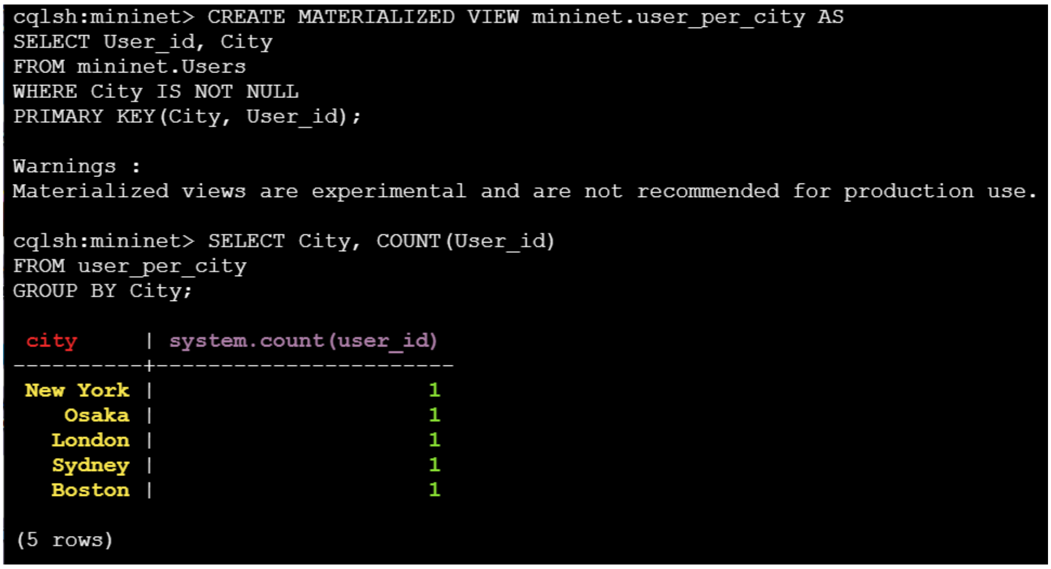
```
cqlsh:mininet> CREATE INDEX ON mininet.Users (Age);
cqlsh:mininet> SELECT * FROM mininet.Users WHERE Age>=22 AND Age<=30
ALLOW FILTERING;

 user_id                              | age | city     | country   | email             | password  | subscription_type | username
--------------------------------------+-----+----------+-----------+-------------------+-----------+-------------------+--------------
 370f2602-490b-4f84-a4d2-db0710262191 |  30 |   Boston |       USA |   bob@example.com | password4 |               UHD |    bob_jones
 143f241a-7c7c-4fdb-8efa-78eb3891a70c |  28 | New York |       USA |  john@example.com | password1 |                HD |     john_doe
 ec798834-4dd7-4011-a651-90ad92f37390 |  25 |   Sydney | Australia |  emma@example.com | password5 |                HD | emma_johnson

(3 rows)
```

6.5 Count how many users exist per specific city

```
CREATE MATERIALIZED VIEW mininet.user_per_city AS
SELECT User_id, City
FROM mininet.Users
WHERE City IS NOT NULL
PRIMARY KEY(City, User_id);
```

A new `user_per_city` table is created from `Users` table, that has additional primary key but same data as the source table. Here, only `City` and `User_id` properties are used, with City being the new addition to primary key. In materialized view's primary key, rows with null values are to be excluded (dataStax2, n.d.). This materialized table is then used to export the query to group the number of users per city.

```
SELECT City, COUNT(User_id)
FROM user_per_city
GROUP BY City;
```



```
cqlsh:mininet> CREATE MATERIALIZED VIEW mininet.user_per_city AS
SELECT User_id, City
FROM mininet.Users
WHERE City IS NOT NULL
PRIMARY KEY(City, User_id);

Warnings :
Materialized views are experimental and are not recommended for production use.

cqlsh:mininet> SELECT City, COUNT(User_id)
FROM user_per_city
GROUP BY City;

 city     | system.count(user_id)
----------+-----------------------
 New York |                     1
    Osaka |                     1
   London |                     1
   Sydney |                     1
   Boston |                     1

(5 rows)
```

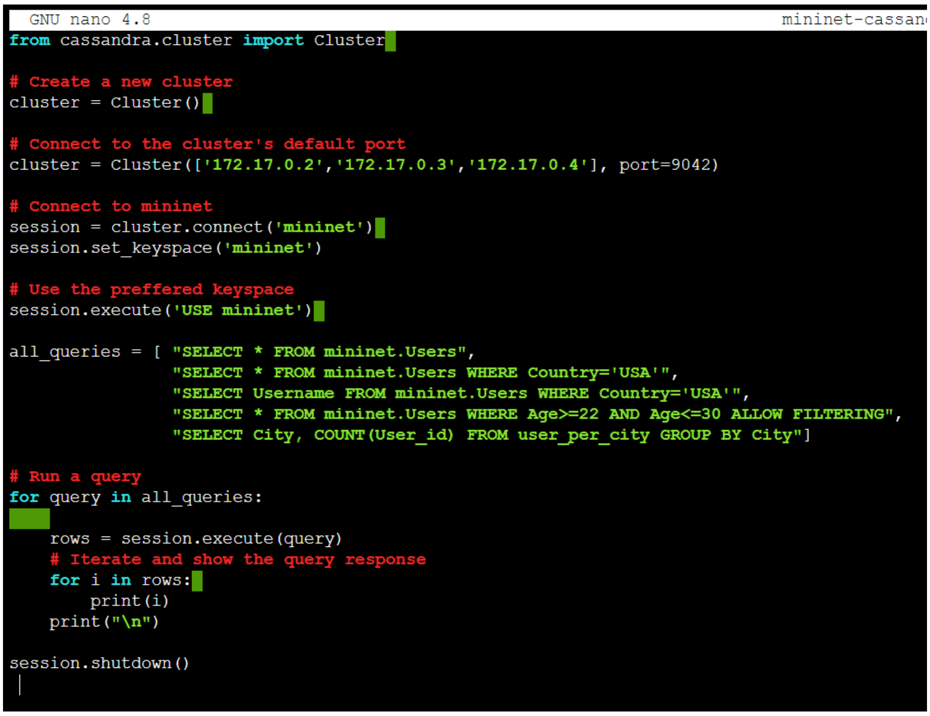
Python Script in Apache Cassandra

```
from cassandra.cluster import Cluster

# Create a new cluster
cluster = Cluster()

# Connect to the cluster's default port
cluster = Cluster(['172.17.0.2','172.17.0.3','172.17.0.4'],
port=9042)

# Connect to mininet
session = cluster.connect('mininet')
session.set_keyspace('mininet')

# Use the preffered keyspace
session.execute('USE mininet')

all_queries = [ "SELECT * FROM mininet.Users",
                "SELECT * FROM mininet.Users WHERE Country='USA'",
                "SELECT Username FROM mininet.Users WHERE
Country='USA'",
                "SELECT * FROM mininet.Users WHERE Age>=22 AND
Age<=30 ALLOW FILTERING",
                "SELECT City, COUNT(User_id) FROM user_per_city
GROUP BY City"
]

# Run a query
for query in all_queries:
    rows = session.execute(query)
    print(f"Query {query}: "
    # Iterate and show the query response
    for i in rows:
        print(i)

    print("\n")
session.shutdown()
```

This python script in `mininet-cassandra.py` (also see Image 2) allows to connect to all three nodes in the cluster using their IP addresses (Image 1) with the help of `cassandra-driver`.

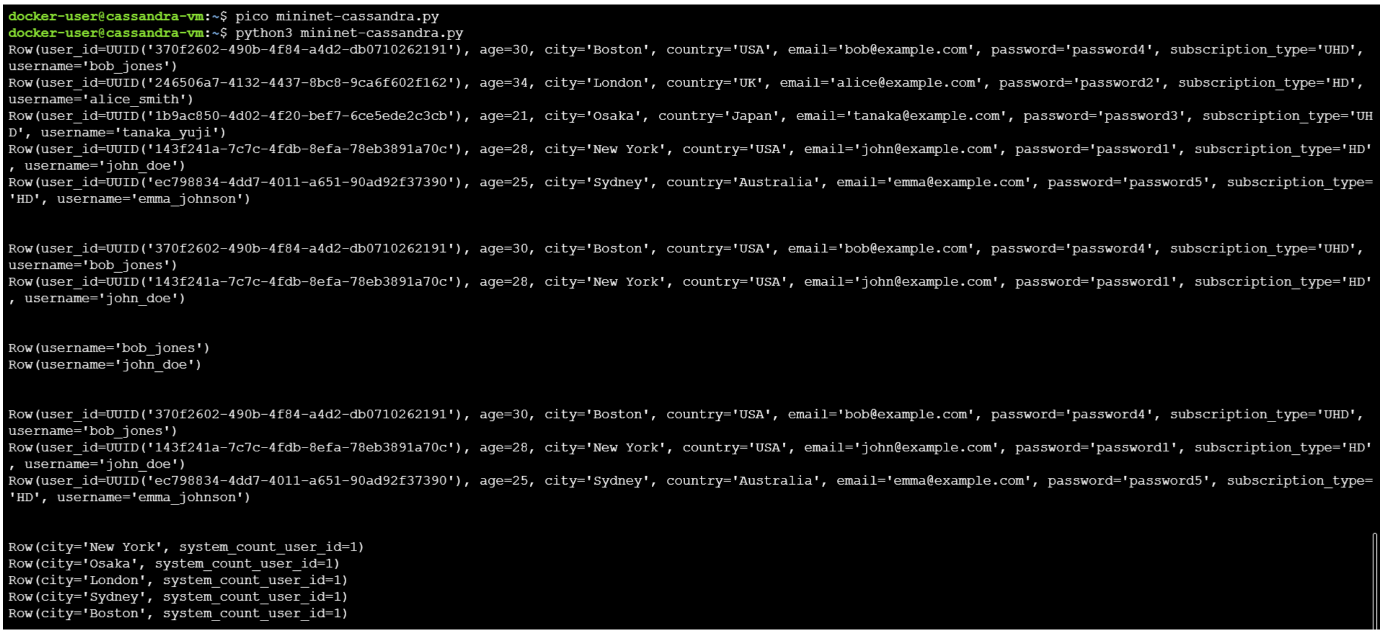
In this file, the queries are stored as rows in a list called `all_queries` . The session iterates through each row and prints the query response, shown in Image 3.

Image 4 shows that even when one of the nodes are down (by terminating Cassandra-1), the python application still runs successfully on another available node from the cluster.



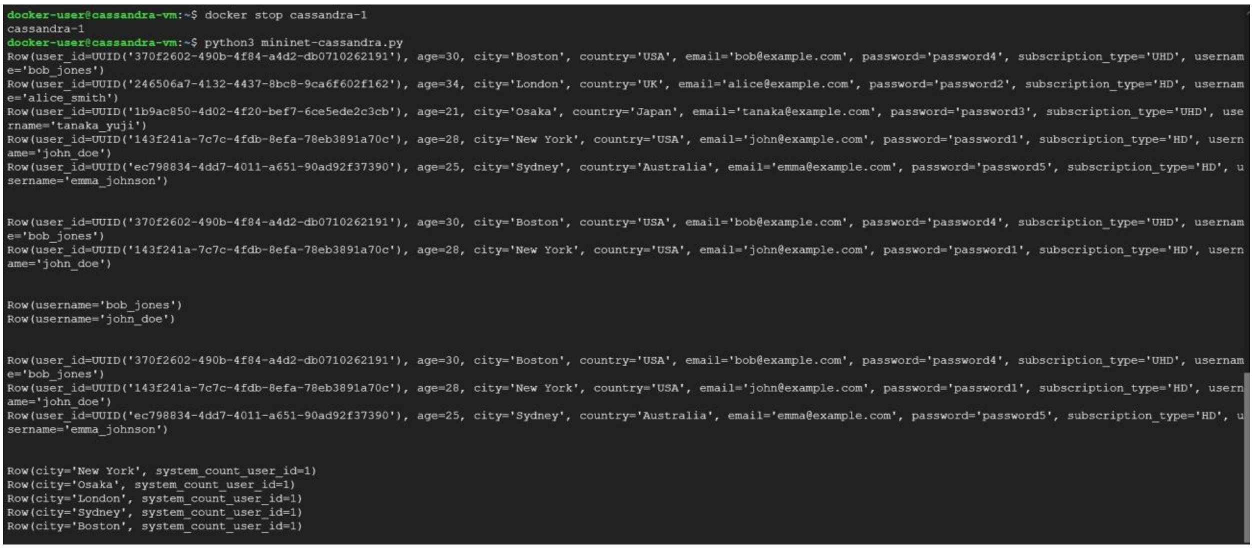
```
  GNU nano 4.8                                                   mininet-cassan
from cassandra.cluster import Cluster

# Create a new cluster
cluster = Cluster()

# Connect to the cluster's default port
cluster = Cluster(['172.17.0.2','172.17.0.3','172.17.0.4'], port=9042)

# Connect to mininet
session = cluster.connect('mininet')
session.set_keyspace('mininet')

# Use the preffered keyspace
session.execute('USE mininet')

all_queries = [ "SELECT * FROM mininet.Users",
                "SELECT * FROM mininet.Users WHERE Country='USA'",
                "SELECT Username FROM mininet.Users WHERE Country='USA'",
                "SELECT * FROM mininet.Users WHERE Age>=22 AND Age<=30 ALLOW FILTERING",
                "SELECT City, COUNT(User_id) FROM user_per_city GROUP BY City"]

# Run a query
for query in all_queries:

    rows = session.execute(query)
    # Iterate and show the query response
    for i in rows:
        print(i)
    print("\n")

session.shutdown()
```

Image 2: python script of mininet-cassandra.py inside pico editor



```
docker-user@cassandra-vm:~$ pico mininet-cassandra.py
docker-user@cassandra-vm:~$ python3 mininet-cassandra.py
Row(user_id=UUID('370f2602-490b-4f84-a4d2-db0710262191'), age=30, city='Boston', country='USA', email='bob@example.com', password='password4', subscription_type='UHD',
username='bob_jones')
Row(user_id=UUID('246506a7-4132-4437-8bc8-9ca6f602f162'), age=34, city='London', country='UK', email='alice@example.com', password='password2', subscription_type='HD',
username='alice_smith')
Row(user_id=UUID('1b9ac850-4d02-4f20-bef7-6ce5ede2c3cb'), age=21, city='Osaka', country='Japan', email='tanaka@example.com', password='password3', subscription_type='UH
D', username='tanaka_yuji')
Row(user_id=UUID('143f241a-7c7c-4fdb-8efa-78eb3891a70c'), age=28, city='New York', country='USA', email='john@example.com', password='password1', subscription_type='HD'
, username='john_doe')
Row(user_id=UUID('ec798834-4dd7-4011-a651-90ad92f37390'), age=25, city='Sydney', country='Australia', email='emma@example.com', password='password5', subscription_type=
'HD', username='emma_johnson')


Row(user_id=UUID('370f2602-490b-4f84-a4d2-db0710262191'), age=30, city='Boston', country='USA', email='bob@example.com', password='password4', subscription_type='UHD',
username='bob_jones')
Row(user_id=UUID('143f241a-7c7c-4fdb-8efa-78eb3891a70c'), age=28, city='New York', country='USA', email='john@example.com', password='password1', subscription_type='HD'
, username='john_doe')


Row(username='bob_jones')
Row(username='john_doe')


Row(user_id=UUID('370f2602-490b-4f84-a4d2-db0710262191'), age=30, city='Boston', country='USA', email='bob@example.com', password='password4', subscription_type='UHD',
username='bob_jones')
Row(user_id=UUID('143f241a-7c7c-4fdb-8efa-78eb3891a70c'), age=28, city='New York', country='USA', email='john@example.com', password='password1', subscription_type='HD'
, username='john_doe')
Row(user_id=UUID('ec798834-4dd7-4011-a651-90ad92f37390'), age=25, city='Sydney', country='Australia', email='emma@example.com', password='password5', subscription_type=
'HD', username='emma_johnson')


Row(city='New York', system_count_user_id=1)
Row(city='Osaka', system_count_user_id=1)
Row(city='London', system_count_user_id=1)
Row(city='Sydney', system_count_user_id=1)
Row(city='Boston', system_count_user_id=1)
```

Image 3: Output after running mininetcasandra.py



```
docker-user@cassandra-vm:~$ docker stop cassandra-1
cassandra-1
docker-user@cassandra-vm:~$ python3 mininet-cassandra.py
Row(user_id=UUID('370f2602-490b-4f84-a4d2-db0710262191'), age=30, city='Boston', country='USA', email='bob@example.com', password='password4', subscription_type='UHD', usernam
e='bob_jones')
Row(user_id=UUID('246506a7-4132-4437-8bc8-9ca6f602f162'), age=34, city='London', country='UK', email='alice@example.com', password='password2', subscription_type='HD', usernam
e='alice_smith')
Row(user_id=UUID('1b9ac850-4d02-4f20-bef7-6ce5ede2c3cb'), age=21, city='Osaka', country='Japan', email='tanaka@example.com', password='password3', subscription_type='UHD', use
rname='tanaka_yuji')
Row(user_id=UUID('143f241a-7c7c-4fdb-8efa-78eb3891a70c'), age=28, city='New York', country='USA', email='john@example.com', password='password1', subscription_type='HD', usern
ame='john_doe')
Row(user_id=UUID('ec798834-4dd7-4011-a651-90ad92f37390'), age=25, city='Sydney', country='Australia', email='emma@example.com', password='password5', subscription_type='HD', u
sername='emma_johnson')


Row(user_id=UUID('370f2602-490b-4f84-a4d2-db0710262191'), age=30, city='Boston', country='USA', email='bob@example.com', password='password4', subscription_type='UHD', usernam
e='bob_jones')
Row(user_id=UUID('143f241a-7c7c-4fdb-8efa-78eb3891a70c'), age=28, city='New York', country='USA', email='john@example.com', password='password1', subscription_type='HD', usern
ame='john_doe')


Row(username='bob_jones')
Row(username='john_doe')


Row(user_id=UUID('370f2602-490b-4f84-a4d2-db0710262191'), age=30, city='Boston', country='USA', email='bob@example.com', password='password4', subscription_type='UHD', usernam
e='bob_jones')
Row(user_id=UUID('143f241a-7c7c-4fdb-8efa-78eb3891a70c'), age=28, city='New York', country='USA', email='john@example.com', password='password1', subscription_type='HD', usern
ame='john_doe')
Row(user_id=UUID('ec798834-4dd7-4011-a651-90ad92f37390'), age=25, city='Sydney', country='Australia', email='emma@example.com', password='password5', subscription_type='HD', u
sername='emma_johnson')


Row(city='New York', system_count_user_id=1)
Row(city='Osaka', system_count_user_id=1)
Row(city='London', system_count_user_id=1)
Row(city='Sydney', system_count_user_id=1)
Row(city='Boston', system_count_user_id=1)
```

Image 4: Python application after stopping 1 node.

Bibliography

admin. (2024). *How to Calculate Age in SQL from Date of Birth*. Retrieved from Sada Tech: https://tech.sadaalomma.com/sql/how-to-calculate-age-in-sql-from-date-of-birth/

datastax. (n.d.). *Batching inserts, updates and deletes*. Retrieved from DataStax Documentation: https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useBatch.html

dataStax2. (n.d.). *Creating a materialized view*. Retrieved from DataStax Documentation: https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useCreateMV.html

DataStax3. (2024). *CQL data types*. Retrieved from DataStax Documentation: https://docs.datastax.com/en/cql-oss/3.1/cql/cql_reference/cql_data_types_c.html

Ian. (2016). *What is Referential Integrity?* Retrieved from Database.Guide: https://database.guide/what-is-referential-integrity/

mySQL. (n.d.). *10.5.15 MySQLCursor.description Property*. Retrieved from MySQL: https://dev.mysql.com/doc/connector-python/en/connector-python-api-mysqlcursor-description.html

Rainardi, V. (2015). *Data Types for Common Columns* . Retrieved from Data Platform and Data Science: https://dwbi1.wordpress.com/2015/09/19/data-types-for-common-columns/

steliosot. (2024). *Lab7: Introduction to Apache Cassandra*. Retrieved from GitHub: https://github.com/warestack/bda/blob/main/session7/README.md

TutorialsTeacher. (n.d.). *Tables Relations in SQL Server: One-to-One, One-to-Many, Many-to-Many*. Retrieved from TutorialsTeacher.com: https://www.tutorialsteacher.com/sqlserver/tables-relations

W3Schools. (n.d.). *SQL Data Types for MySQL, SQL Server, and MS Access*. Retrieved from W3schools: https://www.w3schools.com/sql/sql_datatypes.asp

Other Sources

https://docs.datastax.com/en/cql-oss/3.3/cql/cql_using/useBatchGoodExample.html

https://cassandra.apache.org/doc/stable/cassandra/cql/dml.html