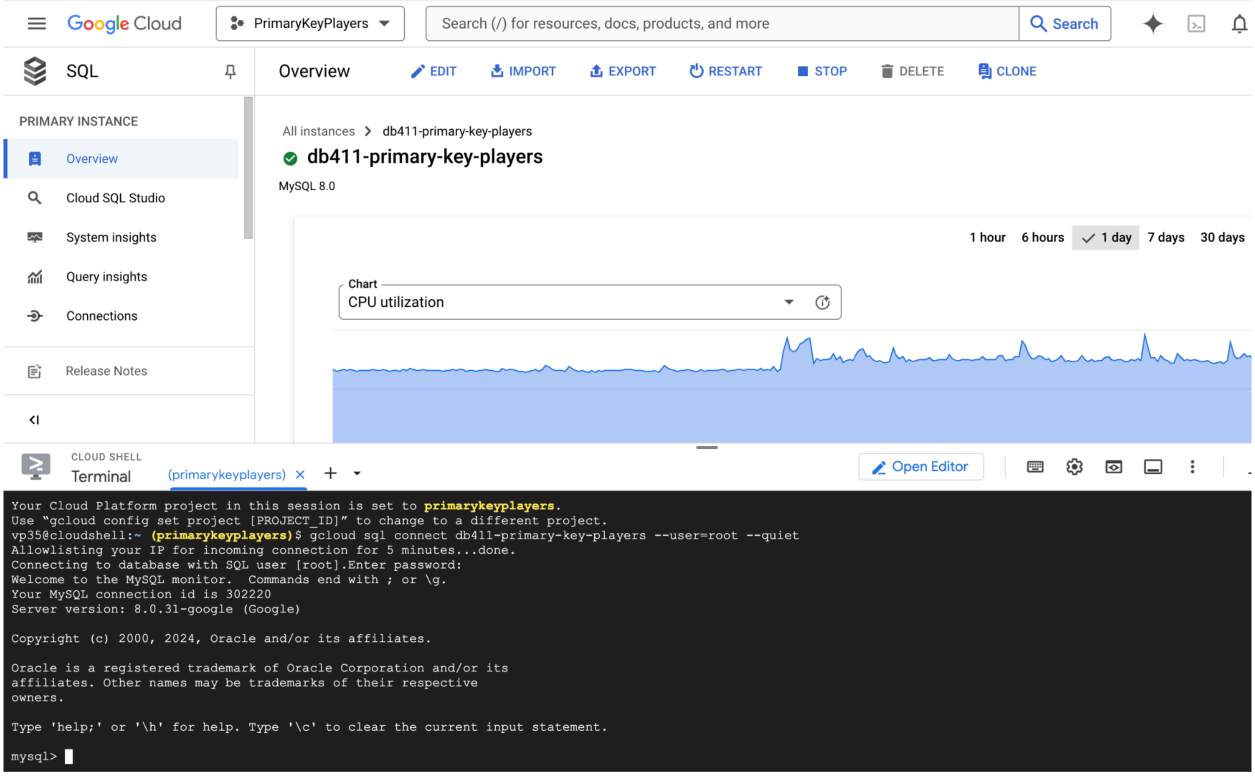
GCP



```
Your Cloud Platform project in this session is set to primarykeyplayers.
Use “gcloud config set project [PROJECT_ID]” to change to a different project.
vp35@cloudshell:~ (primarykeyplayers)$ gcloud sql connect db411-primary-key-players --user=root --quiet
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 302220
Server version: 8.0.31-google (Google)

Copyright (c) 2000, 2024, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

PART 1

Data Definition Language

1. Creating a Hospital table first, as other tables are dependent on them

```
CREATE TABLE Hospital (
    HospitalId INT PRIMARY KEY,
    HospitalName VARCHAR(100),
    Address VARCHAR(255),
    Rating DECIMAL(3, 2) CHECK (Rating BETWEEN 0 AND 5),
    Phone VARCHAR(15) UNIQUE,
    Email VARCHAR(100) UNIQUE
);
```

2. Creating Doctor table

```
CREATE TABLE Doctor (
    DoctorId INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Gender CHAR(1) CHECK (Gender IN ('M', 'F', 'O')),
    Rating DECIMAL(3, 2) CHECK (Rating BETWEEN 0 AND 5),
    Price DECIMAL(10, 2) CHECK (Price >= 0),
    Phone VARCHAR(15) UNIQUE,
    Email VARCHAR(100) UNIQUE,
    HospitalId INT,
    Password CHAR(32),
    FOREIGN KEY (HospitalId) REFERENCES Hospital(HospitalId)
);
```

3. Creating a patient table. We are not able to check if DateOfBirth is in the future. So, we are planning to add a trigger for that.

```
CREATE TABLE Patient (
    PatientId INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    DateOfBirth DATE CHECK (DateOfBirth >= '1900-01-01'),
    Gender CHAR(1) CHECK (Gender IN ('M', 'F', 'O')),
    Address VARCHAR(255),
    Phone VARCHAR(15) UNIQUE,
    Email VARCHAR(100) UNIQUE,
    Password CHAR(32)
);
```

4. Creating LabTest

```
CREATE TABLE LabTest (
    LabTestId INT PRIMARY KEY,
    LabTestName VARCHAR(100),
    Description VARCHAR(255),
    OptimalRange VARCHAR(50),
    Price DECIMAL(10, 2) CHECK (Price >= 0)
);
```

5. Creating Medicine

```
CREATE TABLE Medicine (
    MedicineId INT PRIMARY KEY,
    MedicineName VARCHAR(100),
    Dosage VARCHAR(50),
    Manufacturer VARCHAR(100),
    Price DECIMAL(10, 2) CHECK (Price >= 0)
);
```

6. Creating AppointmentDetails

```
CREATE TABLE AppointmentDetails (
    AppointmentId INT PRIMARY KEY,
    PatientId INT,
    DoctorId INT,
    FOREIGN KEY (PatientId) REFERENCES Patient(PatientId),
    FOREIGN KEY (DoctorId) REFERENCES Doctor(DoctorId)
);
```

7. Creating PrescriptionDetails

```
CREATE TABLE PrescriptionDetails (
    PrescriptionId INT PRIMARY KEY,
    AppointmentId INT,
    DoctorId INT,
    MedicineId INT,
    FOREIGN KEY (AppointmentId) REFERENCES AppointmentDetails(AppointmentId),
    FOREIGN KEY (DoctorId) REFERENCES Doctor(DoctorId),
    FOREIGN KEY (MedicineId) REFERENCES Medicine(MedicineId)
);
```

8. Creating OrderDetails

```
CREATE TABLE OrderDetails (
    OrderId INT PRIMARY KEY,
    DoctorId INT,
    LabTestId INT,
    FOREIGN KEY (DoctorId) REFERENCES Doctor(DoctorId),
    FOREIGN KEY (LabTestId) REFERENCES LabTest(LabTestId)
);
```

9. Creating Transaction

```
CREATE TABLE Transaction (
    TransactionId INT PRIMARY KEY,
    Timestamp TIMESTAMP,
    Amount DECIMAL(10, 2) CHECK (Amount >= 0),
    Mode VARCHAR(50) CHECK (Mode IN ('Cash', 'Credit Card', 'Insurance')),
    HospitalId INT,
    PatientId INT,
    FOREIGN KEY (HospitalId) REFERENCES Hospital(HospitalId),
    FOREIGN KEY (PatientId) REFERENCES Patient(PatientId)
);
```

10. Creating PatientAllergy

```
CREATE TABLE PatientAllergy (
    id INT PRIMARY KEY,
    PatientId INT,
    Allergy VARCHAR(100),
    FOREIGN KEY (PatientId) REFERENCES Patient(PatientId)
);
```

11. Creating MedicineSideEffect

```
CREATE TABLE MedicineSideEffect (
    id INT PRIMARY KEY,
    MedicineId INT,
    SideEffect VARCHAR(255),
    FOREIGN KEY (MedicineId) REFERENCES Medicine(MedicineId)
);
```

12. Creating PrescriptionTimestamp

```
CREATE TABLE PrescriptionTimestamp (
    PrescriptionId INT,
    Timestamp TIMESTAMP,
    FOREIGN KEY (PrescriptionId) REFERENCES PrescriptionDetails(PrescriptionId)
);
```

13. Creating AppointmentStatus

```
CREATE TABLE AppointmentStatus (
    AppointmentId INT,
    Timestamp TIMESTAMP,
    Status VARCHAR(50) CHECK (Status IN ('Scheduled', 'Completed', 'Cancelled')),
    FOREIGN KEY (AppointmentId) REFERENCES AppointmentDetails(AppointmentId)
);
```

14. Creating OrderTimestamp

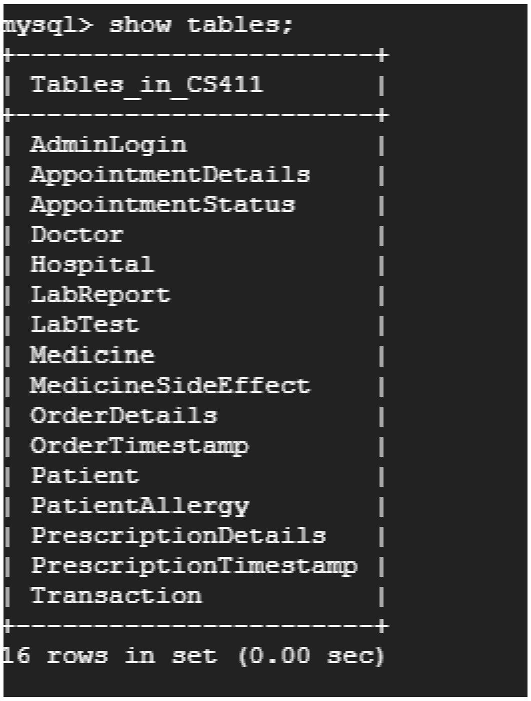
```
CREATE TABLE OrderTimestamp (
    OrderId INT,
    AppointmentId INT,
    Timestamp TIMESTAMP,
    FOREIGN KEY (OrderId) REFERENCES OrderDetails(OrderId),
    FOREIGN KEY (AppointmentId) REFERENCES AppointmentDetails(AppointmentId)
);
```

15. Creating LabReport

```
CREATE TABLE LabReport (
    LabReportId INT PRIMARY KEY,
    Timestamp TIMESTAMP,
    Result VARCHAR(255),
    LabTestId INT,
    PatientId INT,
    FOREIGN KEY (LabTestId) REFERENCES LabTest(LabTestId),
    FOREIGN KEY (PatientId) REFERENCES Patient(PatientId)
);
```

16. Creating AdminLogin

```
CREATE TABLE AdminLogin (
    Email VARCHAR(100) PRIMARY KEY,
    Password CHAR(32)
);
```



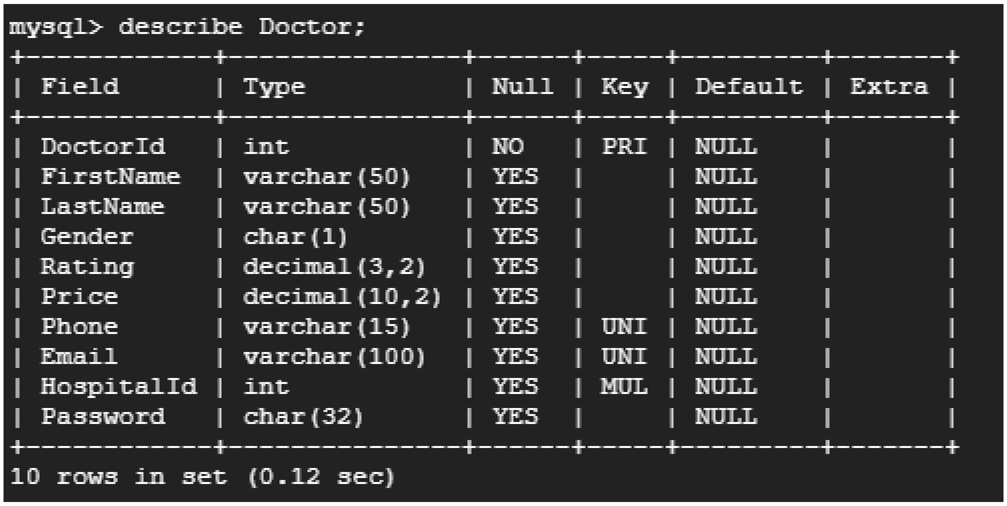
```
mysql> show tables;
+-----------------------+
| Tables_in_CS411       |
+-----------------------+
| AdminLogin            |
| AppointmentDetails    |
| AppointmentStatus     |
| Doctor                |
| Hospital              |
| LabReport             |
| LabTest               |
| Medicine              |
| MedicineSideEffect    |
| OrderDetails          |
| OrderTimestamp        |
| Patient               |
| PatientAllergy        |
| PrescriptionDetails   |
| PrescriptionTimestamp |
| Transaction           |
+-----------------------+
16 rows in set (0.00 sec)
```

As an initial step, we set up all our table structures in our GCP SQL server. So, these are the tables that were mentioned in stage 2.

Insert data into tables

At first, we used the Healthcare dataset to populate our Doctor, Patient, and AppointmentDetails.

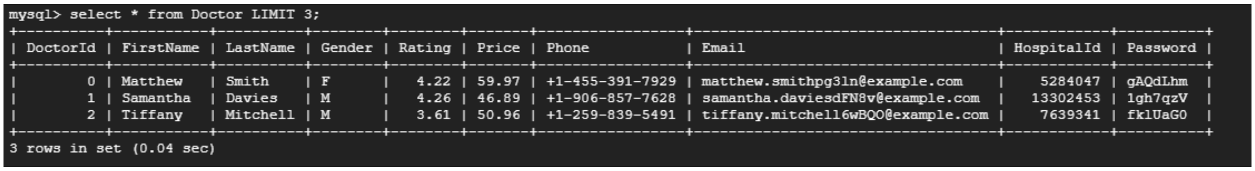
Our Doctor's table looks like this,



```
mysql> describe Doctor;
+------------+---------------+------+-----+---------+-------+
| Field      | Type          | Null | Key | Default | Extra |
+------------+---------------+------+-----+---------+-------+
| DoctorId   | int           | NO   | PRI | NULL    |       |
| FirstName  | varchar(50)   | YES  |     | NULL    |       |
| LastName   | varchar(50)   | YES  |     | NULL    |       |
| Gender     | char(1)       | YES  |     | NULL    |       |
| Rating     | decimal(3,2)  | YES  |     | NULL    |       |
| Price      | decimal(10,2) | YES  |     | NULL    |       |
| Phone      | varchar(15)   | YES  | UNI | NULL    |       |
| Email      | varchar(100)  | YES  | UNI | NULL    |       |
| HospitalId | int           | YES  | MUL | NULL    |       |
| Password   | char(32)      | YES  |     | NULL    |       |
+------------+---------------+------+-----+---------+-------+
10 rows in set (0.12 sec)
```

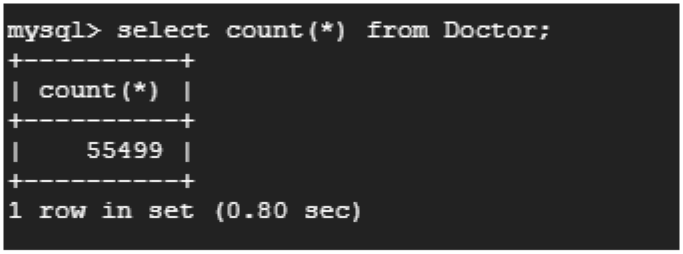
We are generating the unique DoctorId. We are also generating the Rating, Price, Phone, Email, and password for each doctor. Rest is transformed from the healthcare dataset and inserted into our Doctor table.

Now, our doctor's table looks like this.



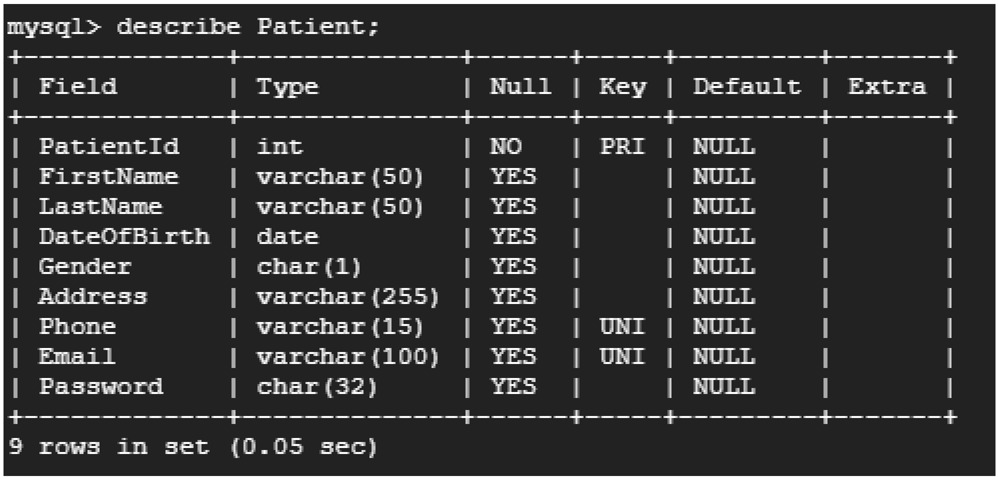
```
mysql> select * from Doctor LIMIT 3;
+----------+-----------+-----------+--------+--------+-------+-----------------+-----------------------------------+------------+----------+
| DoctorId | FirstName | LastName  | Gender | Rating | Price | Phone           | Email                             | HospitalId | Password |
+----------+-----------+-----------+--------+--------+-------+-----------------+-----------------------------------+------------+----------+
|        0 | Matthew   | Smith     | F      |   4.22 | 59.97 | +1-455-391-7929 | matthew.smithpg3ln@example.com    |    5284047 | gAQdLhm  |
|        1 | Samantha  | Davies    | M      |   4.26 | 46.89 | +1-906-857-7628 | samantha.daviesdFN8v@example.com  |   13302453 | 1gh7qzV  |
|        2 | Tiffany   | Mitchell  | M      |   3.61 | 50.96 | +1-259-839-5491 | tiffany.mitchell6wBQO@example.com |    7639341 | fklUaG0  |
+----------+-----------+-----------+--------+--------+-------+-----------------+-----------------------------------+------------+----------+
3 rows in set (0.04 sec)
```

We have nearly 55,500 records for doctors.



```
mysql> select count(*) from Doctor;
+----------+
| count(*) |
+----------+
|    55499 |
+----------+
1 row in set (0.80 sec)
```

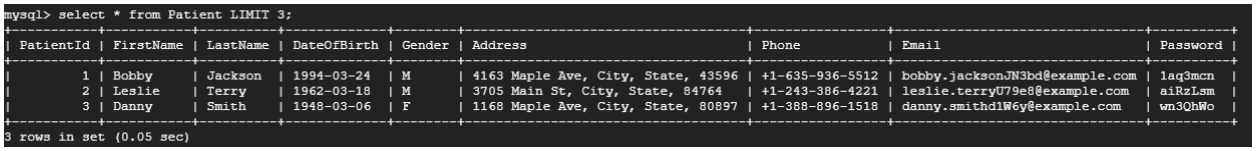
We use the same dataset to populate our Patient details. The structure for the Patient looks like the below,



```
mysql> describe Patient;
+-------------+--------------+------+-----+---------+-------+
| Field       | Type         | Null | Key | Default | Extra |
+-------------+--------------+------+-----+---------+-------+
| PatientId   | int          | NO   | PRI | NULL    |       |
| FirstName   | varchar(50)  | YES  |     | NULL    |       |
| LastName    | varchar(50)  | YES  |     | NULL    |       |
| DateOfBirth | date         | YES  |     | NULL    |       |
| Gender      | char(1)      | YES  |     | NULL    |       |
| Address     | varchar(255) | YES  |     | NULL    |       |
| Phone       | varchar(15)  | YES  | UNI | NULL    |       |
| Email       | varchar(100) | YES  | UNI | NULL    |       |
| Password    | char(32)     | YES  |     | NULL    |       |
+-------------+--------------+------+-----+---------+-------+
9 rows in set (0.05 sec)
```

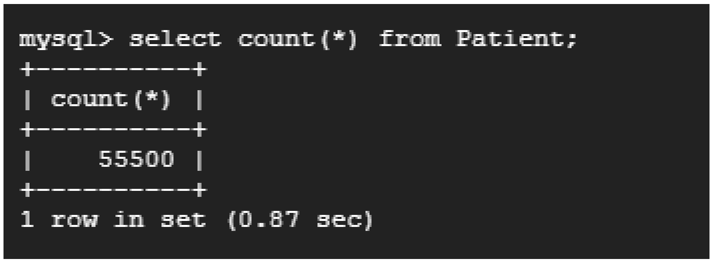
In this structure, we generate unique PatientId, Phone, Email and address. All the other records are transformed from the dataset.

Our patient data looks like this.



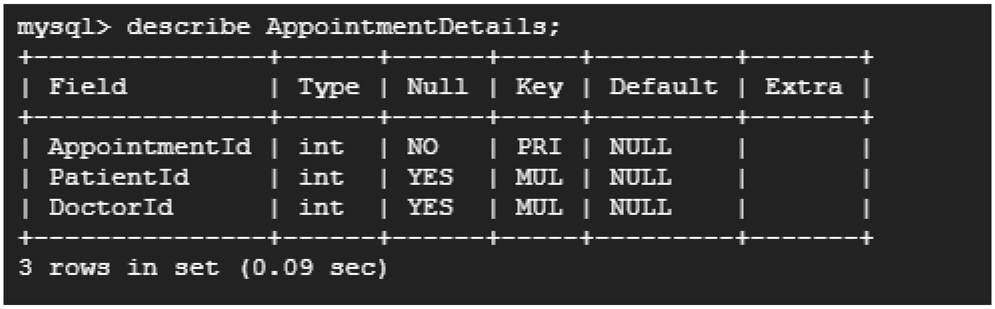
```
mysql> select * from Patient LIMIT 3;
+-----------+-----------+----------+-------------+--------+------------------------------------+-----------------+--------------------------------+----------+
| PatientId | FirstName | LastName | DateOfBirth | Gender | Address                            | Phone           | Email                          | Password |
+-----------+-----------+----------+-------------+--------+------------------------------------+-----------------+--------------------------------+----------+
|         1 | Bobby     | Jackson  | 1994-03-24  | M      | 4163 Maple Ave, City, State, 43596 | +1-635-936-5512 | bobby.jacksonJN3bd@example.com | 1aq3mcn  |
|         2 | Leslie    | Terry    | 1962-03-18  | M      | 3705 Main St, City, State, 84764   | +1-243-386-4221 | leslie.terryU79e8@example.com  | aiRzLsm  |
|         3 | Danny     | Smith    | 1948-03-06  | F      | 1168 Maple Ave, City, State, 80897 | +1-388-896-1518 | danny.smithd1W6y@example.com   | wn3QhWo  |
+-----------+-----------+----------+-------------+--------+------------------------------------+-----------------+--------------------------------+----------+
3 rows in set (0.05 sec)
```

We have 55,500 records in our Patient table.



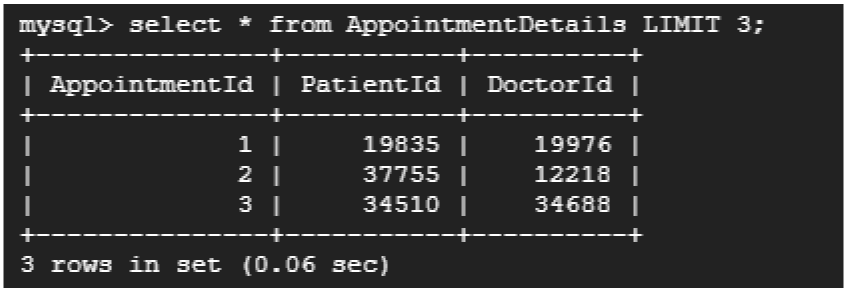
```
mysql> select count(*) from Patient;
+----------+
| count(*) |
+----------+
|    55500 |
+----------+
1 row in set (0.87 sec)
```

With the same dataset's help, we will populate our AppointmentDetails table. Its structure is given below,



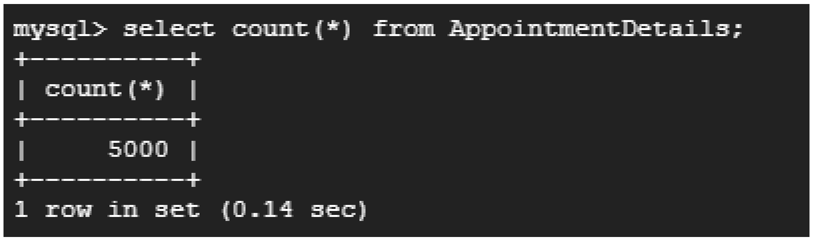
```
mysql> describe AppointmentDetails;
+---------------+------+------+-----+---------+-------+
| Field         | Type | Null | Key | Default | Extra |
+---------------+------+------+-----+---------+-------+
| AppointmentId | int  | NO   | PRI | NULL    |       |
| PatientId     | int  | YES  | MUL | NULL    |       |
| DoctorId      | int  | YES  | MUL | NULL    |       |
+---------------+------+------+-----+---------+-------+
3 rows in set (0.09 sec)
```

It is a table to hold the relationship between the Patient and Doctor. We pick the unique id from the Patient and Doctor and use it here to represent the many-to-many relationship. We generate the AppointmentId. The sample data looks like,



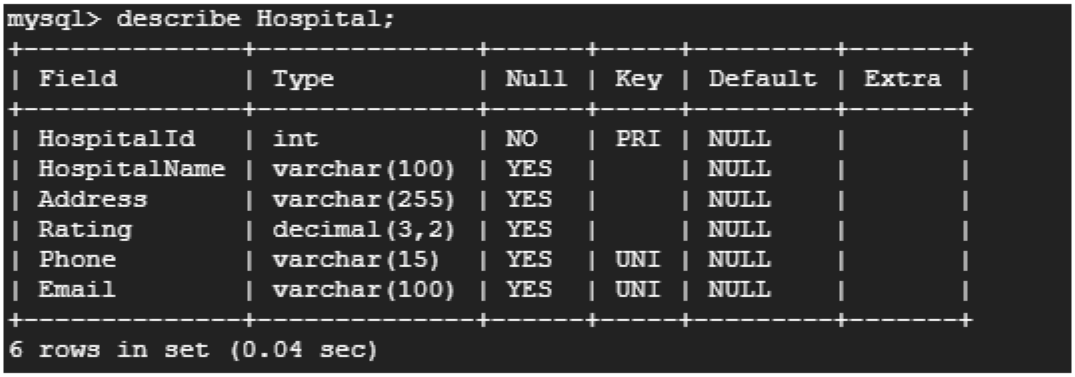
```
mysql> select * from AppointmentDetails LIMIT 3;
+---------------+-----------+----------+
| AppointmentId | PatientId | DoctorId |
+---------------+-----------+----------+
|             1 |     19835 |    19976 |
|             2 |     37755 |    12218 |
|             3 |     34510 |    34688 |
+---------------+-----------+----------+
3 rows in set (0.06 sec)
```

The table has around 5000 records,



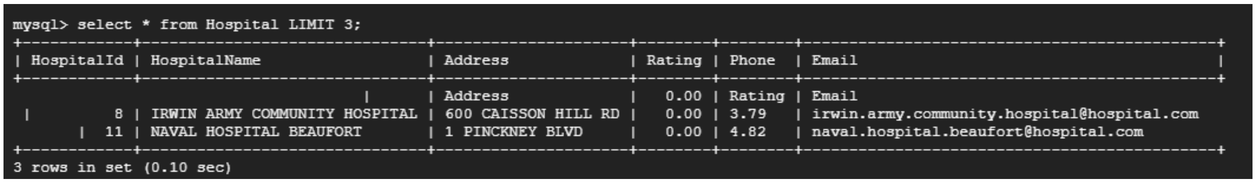
```
mysql> select count(*) from AppointmentDetails;
+----------+
| count(*) |
+----------+
|     5000 |
+----------+
1 row in set (0.14 sec)
```

Now, we use the USA Hospitals dataset to populate our Hospital table. Our hospital table looks like



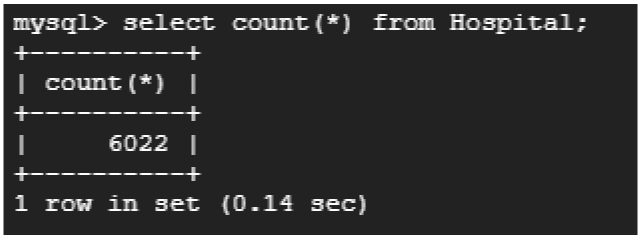
```
mysql> describe Hospital;
+--------------+--------------+------+-----+---------+-------+
| Field        | Type         | Null | Key | Default | Extra |
+--------------+--------------+------+-----+---------+-------+
| HospitalId   | int          | NO   | PRI | NULL    |       |
| HospitalName | varchar(100) | YES  |     | NULL    |       |
| Address      | varchar(255) | YES  |     | NULL    |       |
| Rating       | decimal(3,2) | YES  |     | NULL    |       |
| Phone        | varchar(15)  | YES  | UNI | NULL    |       |
| Email        | varchar(100) | YES  | UNI | NULL    |       |
+--------------+--------------+------+-----+---------+-------+
6 rows in set (0.04 sec)
```

We generated the Rating and Email from our end. All the rest of the data are modified from our dataset to suit our needs. Our Hospital table looks like



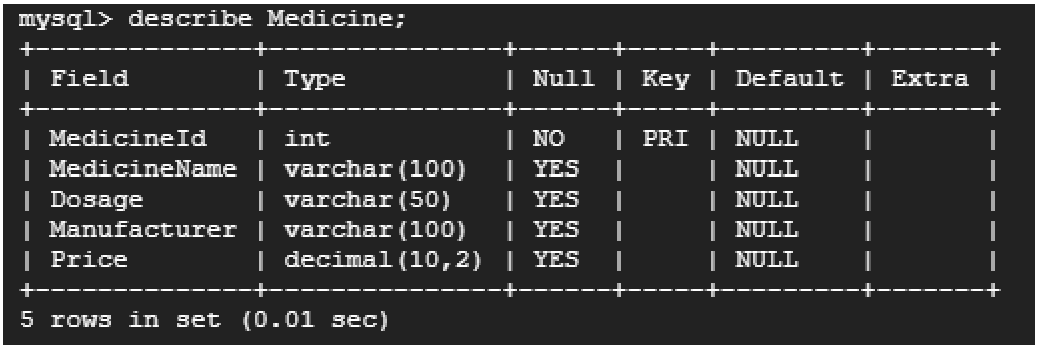
```
mysql> select * from Hospital LIMIT 3;
+------------+------------------------------+--------------------+--------+--------+------------------------------------------+
| HospitalId | HospitalName                 | Address            | Rating | Phone  | Email                                    |
+------------+------------------------------+--------------------+--------+--------+------------------------------------------+
                                     |      | Address            |   0.00 | Rating | Email
 |         8 | IRWIN ARMY COMMUNITY HOSPITAL | 600 CAISSON HILL RD |   0.00 | 3.79   | irwin.army.community.hospital@hospital.com
       |  11 | NAVAL HOSPITAL BEAUFORT      | 1 PINCKNEY BLVD    |   0.00 | 4.82   | naval.hospital.beaufort@hospital.com
+------------+------------------------------+--------------------+--------+--------+------------------------------------------+
3 rows in set (0.10 sec)
```

We have a total of 6022 records in our Hospital table.



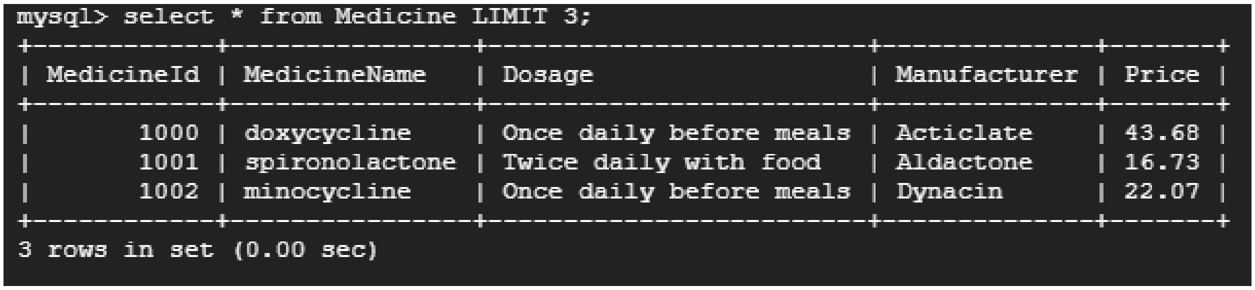
```
mysql> select count(*) from Hospital;
+----------+
| count(*) |
+----------+
|     6022 |
+----------+
1 row in set (0.14 sec)
```

Now, we use the Drugs, Side Effects, and Medical Condition dataset to populate our Medicine and MedicineSideEffect Tables. Our Medicine table holds the set of all medicines, and it structure is like below.



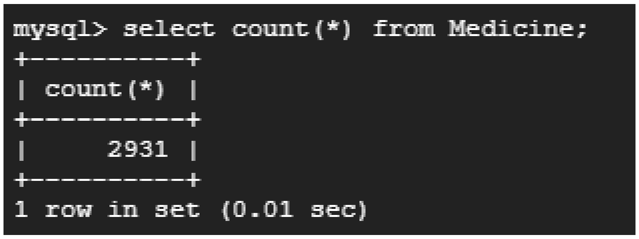
```
mysql> describe Medicine;
+--------------+---------------+------+-----+---------+-------+
| Field        | Type          | Null | Key | Default | Extra |
+--------------+---------------+------+-----+---------+-------+
| MedicineId   | int           | NO   | PRI | NULL    |       |
| MedicineName | varchar(100)  | YES  |     | NULL    |       |
| Dosage       | varchar(50)   | YES  |     | NULL    |       |
| Manufacturer | varchar(100)  | YES  |     | NULL    |       |
| Price        | decimal(10,2) | YES  |     | NULL    |       |
+--------------+---------------+------+-----+---------+-------+
5 rows in set (0.01 sec)
```

We modified the above dataset to give MedicneName and Manufacturer. The sample data looks like



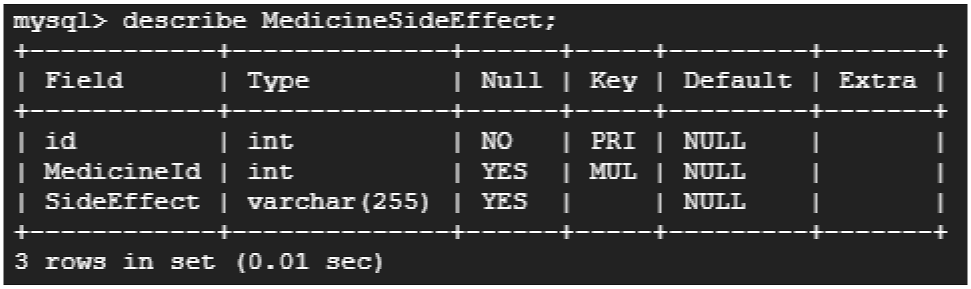
```
mysql> select * from Medicine LIMIT 3;
+------------+----------------+-------------------------+--------------+-------+
| MedicineId | MedicineName   | Dosage                  | Manufacturer | Price |
+------------+----------------+-------------------------+--------------+-------+
|       1000 | doxycycline    | Once daily before meals | Acticlate    | 43.68 |
|       1001 | spironolactone | Twice daily with food   | Aldactone    | 16.73 |
|       1002 | minocycline    | Once daily before meals | Dynacin      | 22.07 |
+------------+----------------+-------------------------+--------------+-------+
3 rows in set (0.00 sec)
```

Our database holds a total of 2931 medicine and their details.



```
mysql> select count(*) from Medicine;
+----------+
| count(*) |
+----------+
|     2931 |
+----------+
1 row in set (0.01 sec)
```

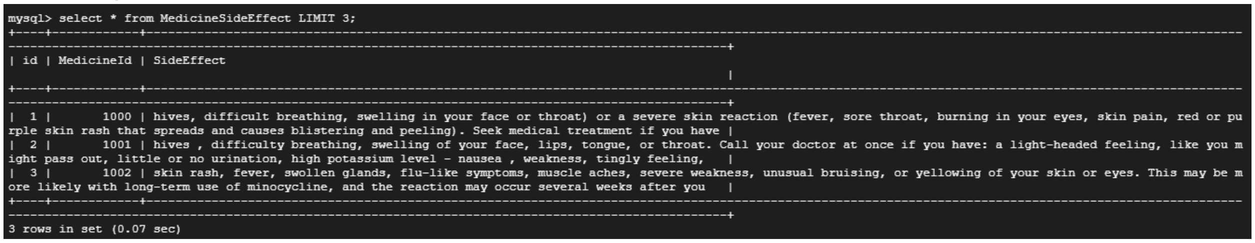
We use the same dataset to populate our MedicineSideEffect; our structure looks like the one below.



```
mysql> describe MedicineSideEffect;
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| id         | int          | NO   | PRI | NULL    |       |
| MedicineId | int          | YES  | MUL | NULL    |       |
| SideEffect | varchar(255) | YES  |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
3 rows in set (0.01 sec)
```

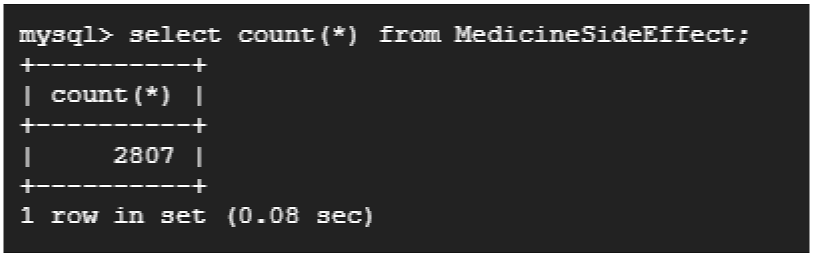
We are referencing the MedicineId from the Medicine table, and we extract SideEffects from the dataset.

The sample data looks like,



```
mysql> select * from MedicineSideEffect LIMIT 3;
+----+------------+---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------+
| id | MedicineId | SideEffect
                                                                                                         |
+----+------------+---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------+
|  1 |       1000 | hives, difficult breathing, swelling in your face or throat) or a severe skin reaction (fever, sore throat, burning in your eyes, skin pain, red or pu
rple skin rash that spreads and causes blistering and peeling). Seek medical treatment if you have |
|  2 |       1001 | hives , difficulty breathing, swelling of your face, lips, tongue, or throat. Call your doctor at once if you have: a light-headed feeling, like you m
ight pass out, little or no urination, high potassium level - nausea , weakness, tingly feeling,   |
|  3 |       1002 | skin rash, fever, swollen glands, flu-like symptoms, muscle aches, severe weakness, unusual bruising, or yellowing of your skin or eyes. This may be m
ore likely with long-term use of minocycline, and the reaction may occur several weeks after you   |
+----+------------+---------------------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------+
3 rows in set (0.07 sec)
```

We have a total of 2807 records.



```
mysql> select count(*) from MedicineSideEffect;
+----------+
| count(*) |
+----------+
|     2807 |
+----------+
1 row in set (0.08 sec)
```

Thus, we have populated the tables below

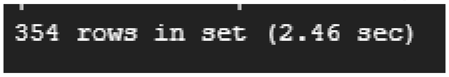
1. Doctor
2. Hospital
3. Patient
4. Medicine
5. AppointmentDetails
6. MedicineSideEffect

Advanced Queries:

1. The first advanced query will be from the admin side, where the admin can check the patients' appointments with top-qualified doctors with ratings >= 4.9. The query is

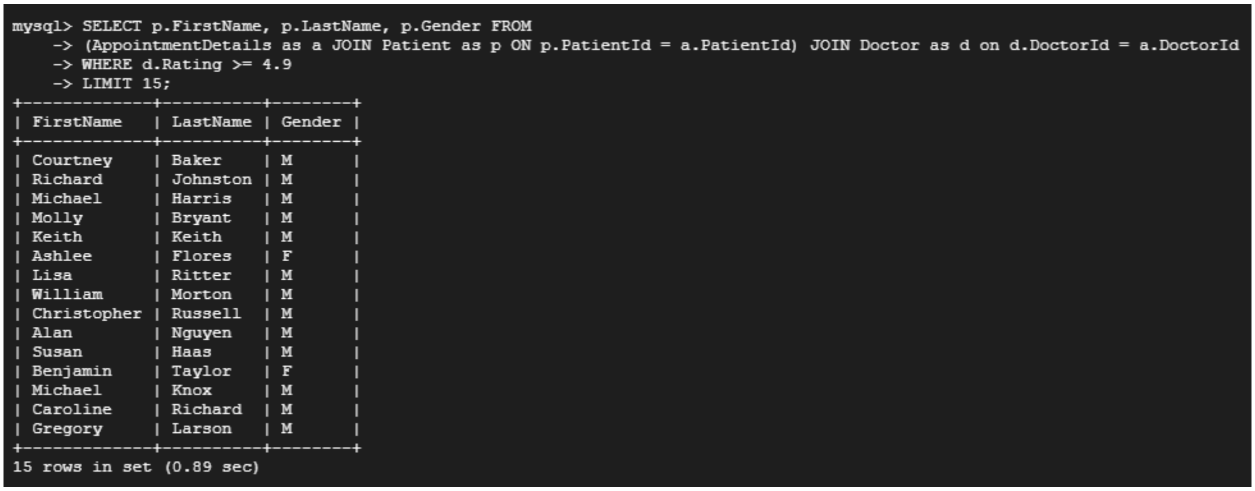
```
SELECT p.FirstName, p.LastName, p.Gender FROM
(AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d
on d.DoctorId = a.DoctorId
WHERE d.Rating >= 4.9;
```

We found 354 records for this query.



```
354 rows in set (2.46 sec)
```

For display purposes, we will limit it to the top 15 results.



```
mysql> SELECT p.FirstName, p.LastName, p.Gender FROM
    -> (AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d on d.DoctorId = a.DoctorId
    -> WHERE d.Rating >= 4.9
    -> LIMIT 15;
+-------------+----------+--------+
| FirstName   | LastName | Gender |
+-------------+----------+--------+
| Courtney    | Baker    | M      |
| Richard     | Johnston | M      |
| Michael     | Harris   | M      |
| Molly       | Bryant   | M      |
| Keith       | Keith    | M      |
| Ashlee      | Flores   | F      |
| Lisa        | Ritter   | M      |
| William     | Morton   | M      |
| Christopher | Russell  | M      |
| Alan        | Nguyen   | M      |
| Susan       | Haas     | M      |
| Benjamin    | Taylor   | F      |
| Michael     | Knox     | M      |
| Caroline    | Richard  | M      |
| Gregory     | Larson   | M      |
+-------------+----------+--------+
15 rows in set (0.89 sec)
```

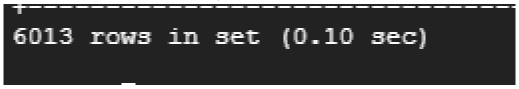
2. The second query is also on the admin side, where we would like to view the number of doctors associated with a particular hospital. We would like to view it hospital-wise in ascending order.

The query for the above problem is,

```
SELECT h.HospitalName, COUNT(*), AVG(h.Rating)
FROM Hospital as h JOIN Doctor as d ON h.HospitalId = d.HospitalId
```

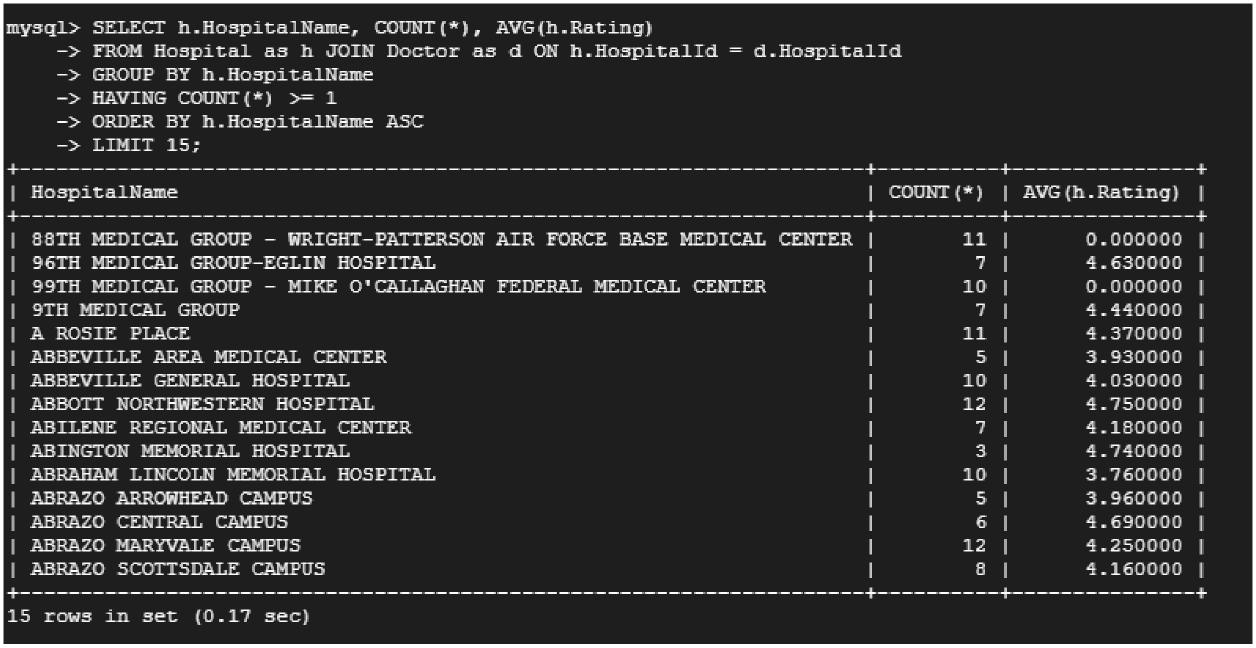
```
GROUP BY h.HospitalName
HAVING COUNT(*) >= 1
ORDER BY h.HospitalName ASC;
```

On execution, it gives



```
+-----------------------------------
6013 rows in set (0.10 sec)
```

For visual purposes, we are limiting to the first 15 records.

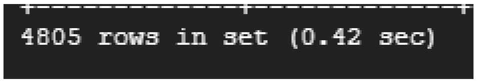


```
mysql> SELECT h.HospitalName, COUNT(*), AVG(h.Rating)
    -> FROM Hospital as h JOIN Doctor as d ON h.HospitalId = d.HospitalId
    -> GROUP BY h.HospitalName
    -> HAVING COUNT(*) >= 1
    -> ORDER BY h.HospitalName ASC
    -> LIMIT 15;
+--------------------------------------------------------------------+----------+---------------+
| HospitalName                                                       | COUNT(*) | AVG(h.Rating) |
+--------------------------------------------------------------------+----------+---------------+
| 88TH MEDICAL GROUP - WRIGHT-PATTERSON AIR FORCE BASE MEDICAL CENTER |       11 |      0.000000 |
| 96TH MEDICAL GROUP-EGLIN HOSPITAL                                  |        7 |      4.630000 |
| 99TH MEDICAL GROUP - MIKE O'CALLAGHAN FEDERAL MEDICAL CENTER       |       10 |      0.000000 |
| 9TH MEDICAL GROUP                                                  |        7 |      4.440000 |
| A ROSIE PLACE                                                      |       11 |      4.370000 |
| ABBEVILLE AREA MEDICAL CENTER                                      |        5 |      3.930000 |
| ABBEVILLE GENERAL HOSPITAL                                         |       10 |      4.030000 |
| ABBOTT NORTHWESTERN HOSPITAL                                       |       12 |      4.750000 |
| ABILENE REGIONAL MEDICAL CENTER                                    |        7 |      4.180000 |
| ABINGTON MEMORIAL HOSPITAL                                         |        3 |      4.740000 |
| ABRAHAM LINCOLN MEMORIAL HOSPITAL                                  |       10 |      3.760000 |
| ABRAZO ARROWHEAD CAMPUS                                            |        5 |      3.960000 |
| ABRAZO CENTRAL CAMPUS                                              |        6 |      4.690000 |
| ABRAZO MARYVALE CAMPUS                                             |       12 |      4.250000 |
| ABRAZO SCOTTSDALE CAMPUS                                           |        8 |      4.160000 |
+--------------------------------------------------------------------+----------+---------------+
15 rows in set (0.17 sec)
```

As an admin, it is important to know the frequent customers who use our apps to book an appointment. A frequent customer must have at least one booking. We are looking for customer-wise bookings for this query.

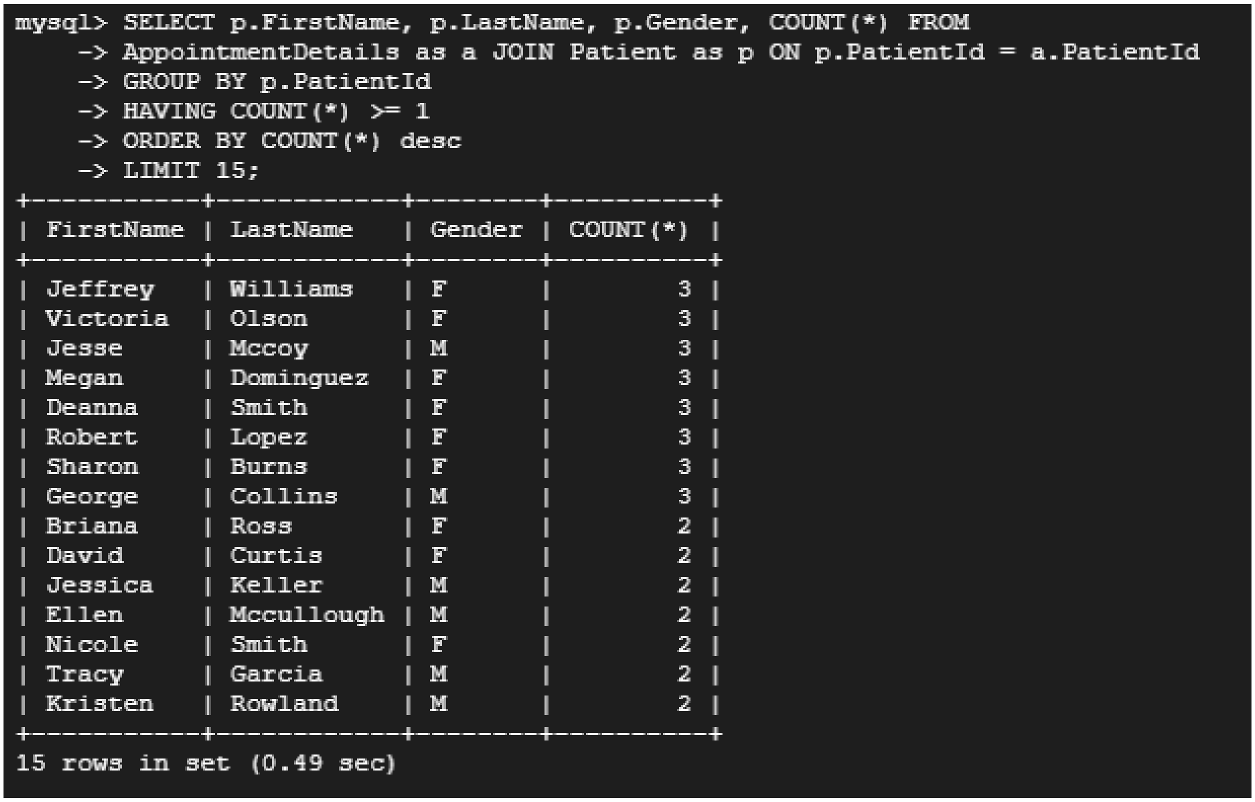
```
SELECT p.FirstName, p.LastName, p.Gender, COUNT(*) FROM
AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
GROUP BY p.PatientId
HAVING COUNT(*) >= 1
ORDER BY COUNT(*) desc;
```

This query gives us results with 4805 rows.



```
+--------------+--------------+
4805 rows in set (0.42 sec)
```

We are limiting it to the top 15 for visibility purposes

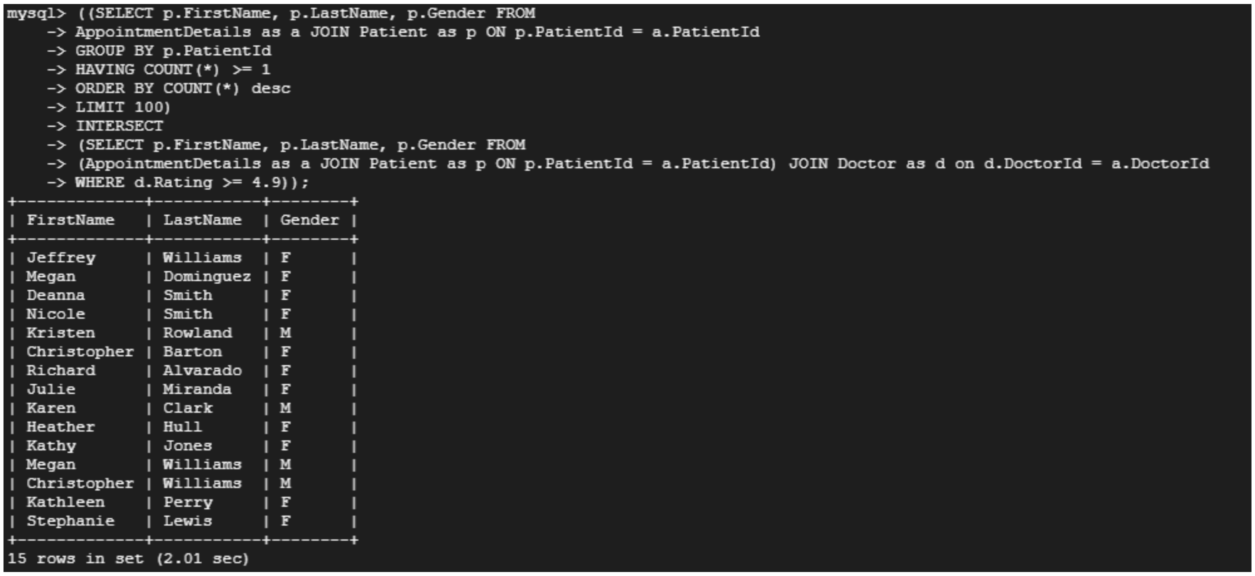


```
mysql> SELECT p.FirstName, p.LastName, p.Gender, COUNT(*) FROM
    -> AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
    -> GROUP BY p.PatientId
    -> HAVING COUNT(*) >= 1
    -> ORDER BY COUNT(*) desc
    -> LIMIT 15;
+-----------+------------+--------+----------+
| FirstName | LastName   | Gender | COUNT(*) |
+-----------+------------+--------+----------+
| Jeffrey   | Williams   | F      |        3 |
| Victoria  | Olson      | F      |        3 |
| Jesse     | Mccoy      | M      |        3 |
| Megan     | Dominguez  | F      |        3 |
| Deanna    | Smith      | F      |        3 |
| Robert    | Lopez      | F      |        3 |
| Sharon    | Burns      | F      |        3 |
| George    | Collins    | M      |        3 |
| Briana    | Ross       | F      |        2 |
| David     | Curtis     | F      |        2 |
| Jessica   | Keller     | M      |        2 |
| Ellen     | Mccullough | M      |        2 |
| Nicole    | Smith      | F      |        2 |
| Tracy     | Garcia     | M      |        2 |
| Kristen   | Rowland    | M      |        2 |
+-----------+------------+--------+----------+
15 rows in set (0.49 sec)
```

Recently, in our app, we rolled out an offer in which the top 100 frequent customers can get one free appointment with top-qualified doctors free of cost. As an admin, I would like to view how many members out of 100 had utilized that offer, so I run a query like

```
((SELECT p.FirstName, p.LastName, p.Gender FROM
AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
GROUP BY p.PatientId
HAVING COUNT(*) >= 1
ORDER BY COUNT(*) desc
LIMIT 100)
INTERSECT
(SELECT p.FirstName, p.LastName, p.Gender FROM
(AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d
on d.DoctorId = a.DoctorId
WHERE d.Rating >= 4.9));
```

Actually, I found only 15 members in the top 100 who used this offer.



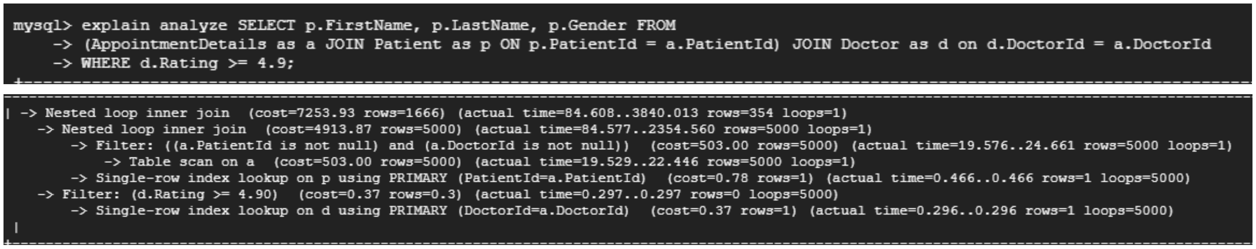
```
mysql> ((SELECT p.FirstName, p.LastName, p.Gender FROM
    -> AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
    -> GROUP BY p.PatientId
    -> HAVING COUNT(*) >= 1
    -> ORDER BY COUNT(*) desc
    -> LIMIT 100)
    -> INTERSECT
    -> (SELECT p.FirstName, p.LastName, p.Gender FROM
    -> (AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d on d.DoctorId = a.DoctorId
    -> WHERE d.Rating >= 4.9));
+-------------+-----------+--------+
| FirstName   | LastName  | Gender |
+-------------+-----------+--------+
| Jeffrey     | Williams  | F      |
| Megan       | Dominguez | F      |
| Deanna      | Smith     | F      |
| Nicole      | Smith     | F      |
| Kristen     | Rowland   | M      |
| Christopher | Barton    | F      |
| Richard     | Alvarado  | F      |
| Julie       | Miranda   | F      |
| Karen       | Clark     | M      |
| Heather     | Hull      | F      |
| Kathy       | Jones     | F      |
| Megan       | Williams  | M      |
| Christopher | Williams  | M      |
| Kathleen    | Perry     | F      |
| Stephanie   | Lewis     | F      |
+-------------+-----------+--------+
15 rows in set (2.01 sec)
```

Creating Index

First query

```
SELECT p.FirstName, p.LastName, p.Gender FROM
(AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d
on d.DoctorId = a.DoctorId
WHERE d.Rating >= 4.9;
```

Initially, we ran the analyze query.



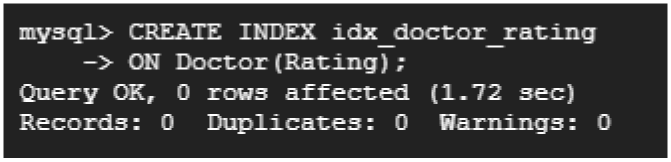
```
mysql> explain analyze SELECT p.FirstName, p.LastName, p.Gender FROM
    -> (AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d on d.DoctorId = a.DoctorId
    -> WHERE d.Rating >= 4.9;
```

```
| -> Nested loop inner join  (cost=7253.93 rows=1666) (actual time=84.608..3840.013 rows=354 loops=1)
    -> Nested loop inner join  (cost=4913.87 rows=5000) (actual time=84.577..2354.560 rows=5000 loops=1)
        -> Filter: ((a.PatientId is not null) and (a.DoctorId is not null))  (cost=503.00 rows=5000) (actual time=19.576..24.661 rows=5000 loops=1)
            -> Table scan on a  (cost=503.00 rows=5000) (actual time=19.529..22.446 rows=5000 loops=1)
        -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.78 rows=1) (actual time=0.466..0.466 rows=1 loops=5000)
    -> Filter: (d.Rating >= 4.90)  (cost=0.37 rows=0.3) (actual time=0.297..0.297 rows=0 loops=5000)
        -> Single-row index lookup on d using PRIMARY (DoctorId=a.DoctorId)  (cost=0.37 rows=1) (actual time=0.296..0.296 rows=1 loops=5000)
 |
```

The query execution shows several costly operations:

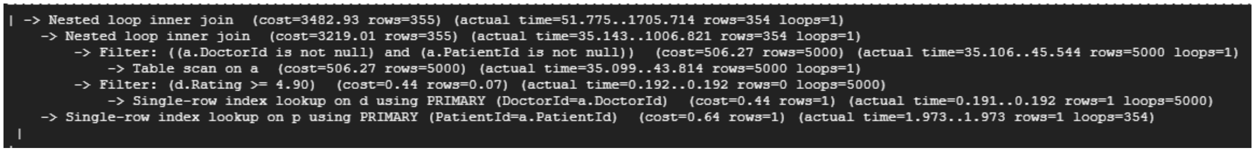
- Total cost: 7253.93 (Nested loop inner join)
- Inner join cost: 4913.87 (Patient and AppointmentDetails)
- Table scan cost: 503.00 (Patient table)
- Filter cost (Rating >= 4.90): 0.37

From this query, we can observe that it was searched for based on a rating of >= 4.9, so let’s try indexing it.



```
mysql> CREATE INDEX idx_doctor_rating
    -> ON Doctor(Rating);
Query OK, 0 rows affected (1.72 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Now, we have created an index in the Rating field named idx_doctor_rating. Let's analyze again.



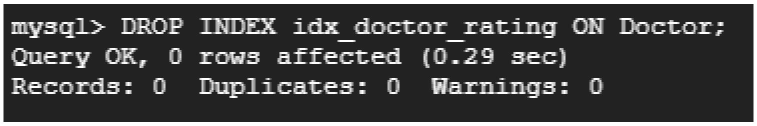
```
| -> Nested loop inner join  (cost=3482.93 rows=355) (actual time=51.775..1705.714 rows=354 loops=1)
    -> Nested loop inner join  (cost=3219.01 rows=355) (actual time=35.143..1006.821 rows=354 loops=1)
        -> Filter: ((a.DoctorId is not null) and (a.PatientId is not null))  (cost=506.27 rows=5000) (actual time=35.106..45.544 rows=5000 loops=1)
            -> Table scan on a  (cost=506.27 rows=5000) (actual time=35.099..43.814 rows=5000 loops=1)
        -> Filter: (d.Rating >= 4.90)  (cost=0.44 rows=0.07) (actual time=0.192..0.192 rows=0 loops=5000)
            -> Single-row index lookup on d using PRIMARY (DoctorId=a.DoctorId)  (cost=0.44 rows=1) (actual time=0.191..0.192 rows=1 loops=5000)
    -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.64 rows=1) (actual time=1.973..1.973 rows=1 loops=354)
 |
```

Current Index Analysis (Rating Index)

The EXPLAIN ANALYZE output shows:

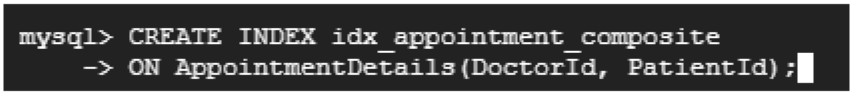
- Total cost reduced from 7253.93 to 3482.93 after adding Rating index1
- The nested loop inner join cost reduced from 4913.87 to 3219.011
- Filter operation on Rating remains relatively constant at cost=0.44

We then drop the rating index.



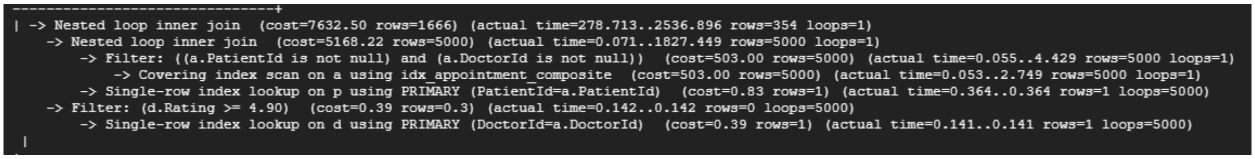
```
mysql> DROP INDEX idx_doctor_rating ON Doctor;
Query OK, 0 rows affected (0.29 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Now, we have created a composite index for DoctorId and PatientId because they can reduce the need for table scans by providing a covering index for the join conditions.



```
mysql> CREATE INDEX idx_appointment_composite
    -> ON AppointmentDetails(DoctorId, PatientId);
```

The result was



```
-----------------------------+
| -> Nested loop inner join  (cost=7632.50 rows=1666) (actual time=278.713..2536.896 rows=354 loops=1)
    -> Nested loop inner join  (cost=5168.22 rows=5000) (actual time=0.071..1827.449 rows=5000 loops=1)
        -> Filter: ((a.PatientId is not null) and (a.DoctorId is not null))  (cost=503.00 rows=5000) (actual time=0.055..4.429 rows=5000 loops=1)
            -> Covering index scan on a using idx_appointment_composite  (cost=503.00 rows=5000) (actual time=0.053..2.749 rows=5000 loops=1)
        -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.83 rows=1) (actual time=0.364..0.364 rows=1 loops=5000)
    -> Filter: (d.Rating >= 4.90)  (cost=0.39 rows=0.3) (actual time=0.142..0.142 rows=0 loops=5000)
        -> Single-row index lookup on d using PRIMARY (DoctorId=a.DoctorId)  (cost=0.39 rows=1) (actual time=0.141..0.141 rows=1 loops=5000)
 |
```

Analysis of Composite Index Performance

Why Composite Index Didn't Help:

1. The cost actually increased from 7253.93 to 7632.501, which is unexpected
2. The number of rows examined in the nested loop increased to 16661, compared to 354 with just the rating index2
3. The covering index scan shows a cost of 503.001, which indicates the optimizer is still performing full scans

Root Causes:

1. The query optimizer might not be effectively using the composite index because:
    - The WHERE clause on Rating is the most selective condition
    - The join order chosen by the optimizer isn't optimal for the composite index structure
2. The composite index (DoctorId, PatientId) might not be in the optimal order for this specific query pattern1.

Better Optimization Strategy

Based on these results, a better approach would be:

1. Keep the Rating index on Doctor table as it showed the best performance (reducing cost to 3482.93)

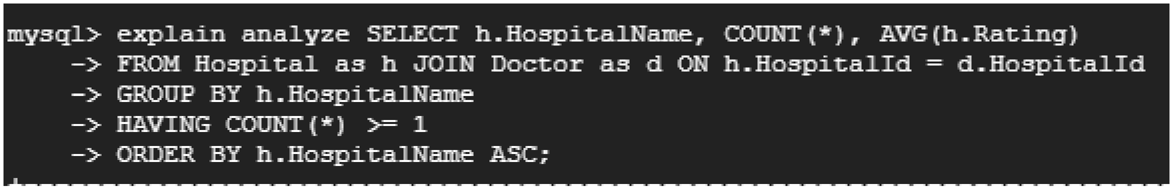
Scenario	**Total Cost**	**Inner Join Cost**	**Rows Examined**
No Index	7253.93	4913.87	5000
Rating Index	3482.93	3219.01	354

Composite Index	7632.50	5168.22	1666

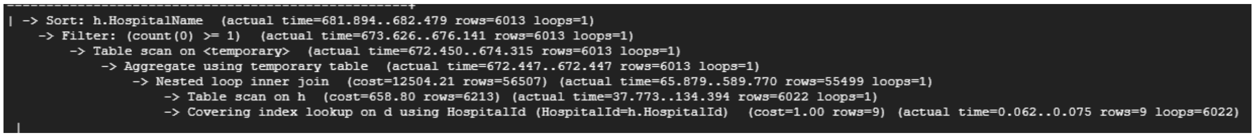
Now, we will move to our second query

```
SELECT h.HospitalName, COUNT(*), AVG(h.Rating)
FROM Hospital as h JOIN Doctor as d ON h.HospitalId = d.HospitalId
GROUP BY h.HospitalName
HAVING COUNT(*) >= 1
ORDER BY h.HospitalName ASC;
```

When I analyze it, I get it as,



```
mysql> explain analyze SELECT h.HospitalName, COUNT(*), AVG(h.Rating)
    -> FROM Hospital as h JOIN Doctor as d ON h.HospitalId = d.HospitalId
    -> GROUP BY h.HospitalName
    -> HAVING COUNT(*) >= 1
    -> ORDER BY h.HospitalName ASC;
+-------------------------------------------------------------------------------
```

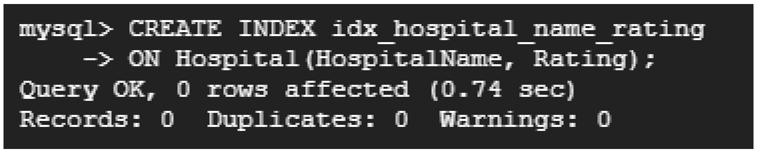


```
-----------------------------------------------------+
| -> Sort: h.HospitalName  (actual time=681.894..682.479 rows=6013 loops=1)
    -> Filter: (count(0) >= 1)  (actual time=673.626..676.141 rows=6013 loops=1)
        -> Table scan on <temporary>  (actual time=672.450..674.315 rows=6013 loops=1)
            -> Aggregate using temporary table  (actual time=672.447..672.447 rows=6013 loops=1)
                -> Nested loop inner join  (cost=12504.21 rows=56507) (actual time=65.879..589.770 rows=55499 loops=1)
                    -> Table scan on h  (cost=658.80 rows=6213) (actual time=37.773..134.394 rows=6022 loops=1)
                    -> Covering index lookup on d using HospitalId (HospitalId=h.HospitalId)  (cost=1.00 rows=9) (actual time=0.062..0.075 rows=9 loops=6022)
 |
```

Query Execution Breakdown:

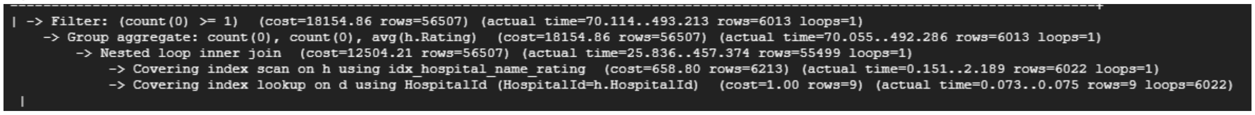
- Total cost: 12504.21
- Table scan on Hospital (h): 658.80 (6213 rows)
- Nested loop inner join: 7632.50 (1666 rows)
- Aggregate using temporary table: 672.447 (6013 rows)
- Sort by HospitalName: 681.894 (6013 rows)

We now create a composite index of HospitalName and rating



```
mysql> CREATE INDEX idx_hospital_name_rating
    -> ON Hospital(HospitalName, Rating);
Query OK, 0 rows affected (0.74 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

I got



```
-------------------------------------------------------------------------------------------------------+
| -> Filter: (count(0) >= 1)  (cost=18154.86 rows=56507) (actual time=70.114..493.213 rows=6013 loops=1)
    -> Group aggregate: count(0), count(0), avg(h.Rating)  (cost=18154.86 rows=56507) (actual time=70.055..492.286 rows=6013 loops=1)
        -> Nested loop inner join  (cost=12504.21 rows=56507) (actual time=25.836..457.374 rows=55499 loops=1)
            -> Covering index scan on h using idx_hospital_name_rating  (cost=658.80 rows=6213) (actual time=0.151..2.189 rows=6022 loops=1)
            -> Covering index lookup on d using HospitalId (HospitalId=h.HospitalId)  (cost=1.00 rows=9) (actual time=0.073..0.075 rows=9 loops=6022)
 |
```

Analysis of Increased Cost

Why Performance Degraded:

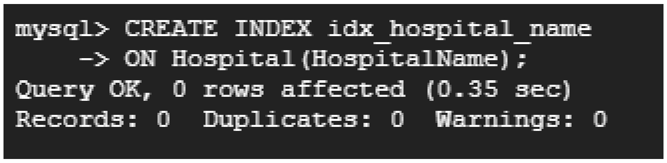
1. The total cost increased from 12504.21 to 18154.86 because:
    - The optimizer uses a different execution plan

- The group aggregate operation is now more expensive (18154.86 rows=56507)
- The temporary table creation and sorting still occurs

Current Execution Path:

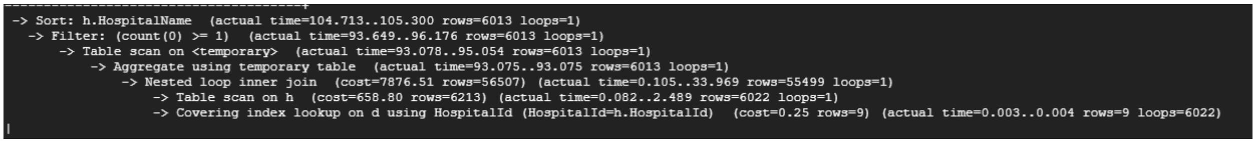
1. Filter (count(*) >= 1): cost=18154.86, rows=56507
2. Group aggregate: cost=18154.86, rows=56507
3. Nested loop join: cost=12504.21, rows=56507
4. Covering index scan on Hospital: cost=658.80, rows=6213
5. Index lookup on Doctor: cost=1.00, rows=9

Now let’s create an index for hospital name



```
mysql> CREATE INDEX idx_hospital_name
    -> ON Hospital(HospitalName);
Query OK, 0 rows affected (0.35 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

And now the cost is



```
-> Sort: h.HospitalName  (actual time=104.713..105.300 rows=6013 loops=1)
  -> Filter: (count(0) >= 1)  (actual time=93.649..96.176 rows=6013 loops=1)
      -> Table scan on <temporary>  (actual time=93.078..95.054 rows=6013 loops=1)
          -> Aggregate using temporary table  (actual time=93.075..93.075 rows=6013 loops=1)
              -> Nested loop inner join  (cost=7876.51 rows=56507) (actual time=0.105..33.969 rows=55499 loops=1)
                  -> Table scan on h  (cost=658.80 rows=6213) (actual time=0.082..2.489 rows=6022 loops=1)
                  -> Covering index lookup on d using HospitalId (HospitalId=h.HospitalId)  (cost=0.25 rows=9) (actual time=0.003..0.004 rows=9 loops=6022)
```

Performance Analysis

Improvements:

1. Total cost reduced from 12504.21 to 7876.51 (37% improvement)
2. Nested loop join cost decreased significantly
3. The index is being utilized for sorting and grouping operations

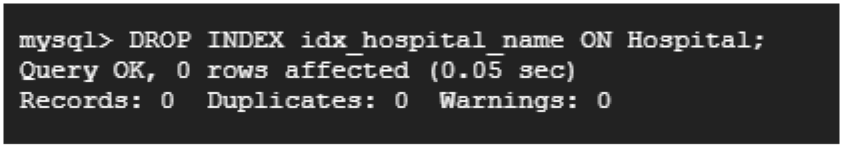
Cost Breakdown:

- Sort operation: 681.894 (rows=6013)
- Filter operation: 673.626 (rows=6013)
- Aggregate using temporary table: 672.447 (rows=6013)
- Nested loop inner join: 7876.51 (rows=55499)
- Table scan: 658.80 (rows=6213)

Why It Helped:

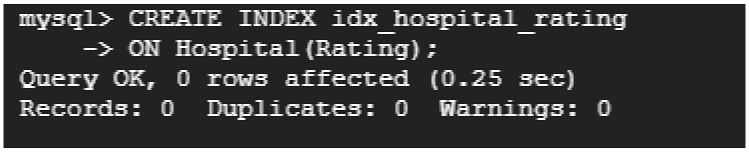
1. The index supports:
    - ORDER BY HospitalName ASC clause
    - GROUP BY HospitalName operation
    - Reduced sorting overhead
2. Better join performance due to ordered access to HospitalName

I’m dropping the Hospital_name index for now

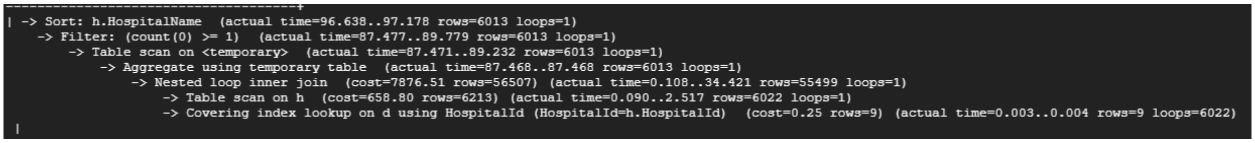


```
mysql> DROP INDEX idx_hospital_name ON Hospital;
Query OK, 0 rows affected (0.05 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Let’s create an index for rating.



```
mysql> CREATE INDEX idx_hospital_rating
    -> ON Hospital(Rating);
Query OK, 0 rows affected (0.25 sec)
Records: 0  Duplicates: 0  Warnings: 0
```



```
-----------------------------------+
| -> Sort: h.HospitalName  (actual time=96.638..97.178 rows=6013 loops=1)
    -> Filter: (count(0) >= 1)  (actual time=87.477..89.779 rows=6013 loops=1)
        -> Table scan on <temporary>  (actual time=87.471..89.232 rows=6013 loops=1)
            -> Aggregate using temporary table  (actual time=87.468..87.468 rows=6013 loops=1)
                -> Nested loop inner join  (cost=7876.51 rows=56507) (actual time=0.108..34.421 rows=55499 loops=1)
                    -> Table scan on h  (cost=658.80 rows=6213) (actual time=0.090..2.517 rows=6022 loops=1)
                    -> Covering index lookup on d using HospitalId (HospitalId=h.HospitalId)  (cost=0.25 rows=9) (actual time=0.003..0.004 rows=9 loops=6022)
 |
```

Cost Analysis

Identical Costs Observed:

1. The nested loop inner join cost remains constant at 7876.51
2. Table scan cost stays at 658.80
3. The index lookup cost remains at 0.25
4. Number of rows examined is consistent:
    - Hospital table scan: 6213 rows
    - Doctor table: 9 rows per lookup
    - Total rows processed: 55499

Why Costs Are Identical:

1. The Rating index doesn't affect:
    - The join operation cost
    - The table scan cost
    - The overall query execution plan
2. The optimizer is using the same execution strategy:
    - The same nested loop join approach
    - The same table scan on the Hospital
    - The same index lookup on Doctor

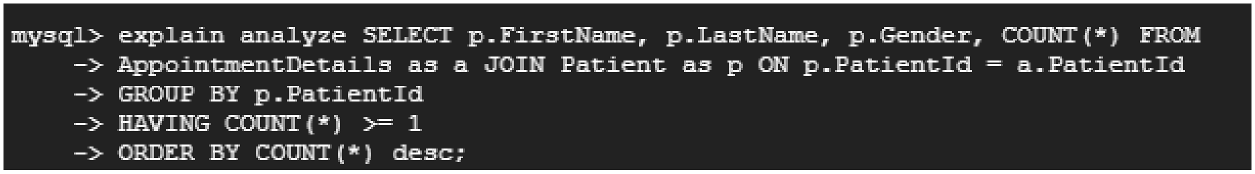
Operation	**No Index**	**HospitalName Index**	**Both (Name+Rating)**	**Rating Index**
Total Cost	12504.21	7876.51	18154.86	12504.21
Sort Time	96.638s	104.713s	70.114s	Similar to base
Filter Time	87.477s	93.649s	70.055s	Similar to base

Aggregate Time	87.468s	93.075s	70.055s	Similar to base
Rows Examined	6013	6013	56507	6013

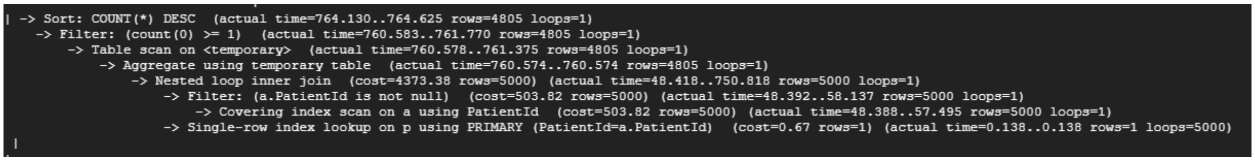
The query 3 is

```
SELECT p.FirstName, p.LastName, p.Gender, COUNT(*) FROM
AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
GROUP BY p.PatientId
HAVING COUNT(*) >= 1
ORDER BY COUNT(*) desc;
```

And when we analyze it, it was



```
mysql> explain analyze SELECT p.FirstName, p.LastName, p.Gender, COUNT(*) FROM
    -> AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
    -> GROUP BY p.PatientId
    -> HAVING COUNT(*) >= 1
    -> ORDER BY COUNT(*) desc;
```



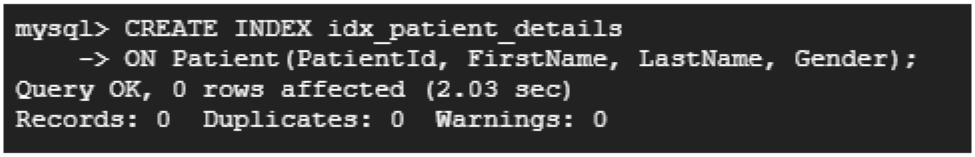
```
| -> Sort: COUNT(*) DESC  (actual time=764.130..764.625 rows=4805 loops=1)
    -> Filter: (count(0) >= 1)  (actual time=760.583..761.770 rows=4805 loops=1)
        -> Table scan on <temporary>  (actual time=760.578..761.375 rows=4805 loops=1)
            -> Aggregate using temporary table  (actual time=760.574..760.574 rows=4805 loops=1)
                -> Nested loop inner join  (cost=4373.38 rows=5000) (actual time=48.418..750.818 rows=5000 loops=1)
                    -> Filter: (a.PatientId is not null)  (cost=503.82 rows=5000) (actual time=48.392..58.137 rows=5000 loops=1)
                        -> Covering index scan on a using PatientId  (cost=503.82 rows=5000) (actual time=48.388..57.495 rows=5000 loops=1)
                    -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.67 rows=1) (actual time=0.138..0.138 rows=1 loops=5000)
|
```

Current Cost Analysis

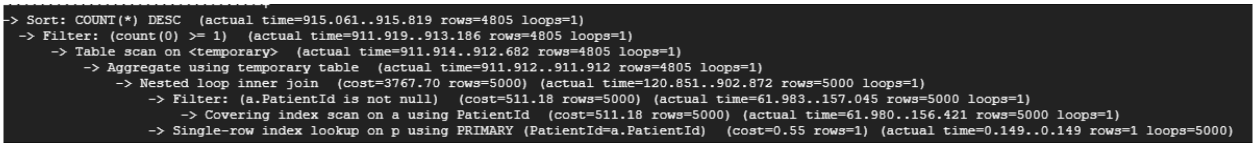
Operation	**Cost**	**Rows**	**Time**
Total Cost	4373.38	5000	-

Sort (COUNT(*) DESC)	-	4805	764.130s
Filter (COUNT(*) >= 1)	-	4805	760.583s
Nested Loop Join	4373.38	5000	-
Filter (PatientId not null)	503.82	5000	48.392s
Index Lookup (PRIMARY)	0.67	1	0.138s

Now we create a complex index like



```
mysql> CREATE INDEX idx_patient_details
    -> ON Patient(PatientId, FirstName, LastName, Gender);
Query OK, 0 rows affected (2.03 sec)
Records: 0  Duplicates: 0  Warnings: 0
```



```
-> Sort: COUNT(*) DESC  (actual time=915.061..915.819 rows=4805 loops=1)
 -> Filter: (count(0) >= 1)  (actual time=911.919..913.186 rows=4805 loops=1)
     -> Table scan on <temporary>  (actual time=911.914..912.682 rows=4805 loops=1)
         -> Aggregate using temporary table  (actual time=911.912..911.912 rows=4805 loops=1)
             -> Nested loop inner join  (cost=3767.70 rows=5000) (actual time=120.851..902.872 rows=5000 loops=1)
                 -> Filter: (a.PatientId is not null)  (cost=511.18 rows=5000) (actual time=61.983..157.045 rows=5000 loops=1)
                     -> Covering index scan on a using PatientId  (cost=511.18 rows=5000) (actual time=61.980..156.421 rows=5000 loops=1)
                 -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.55 rows=1) (actual time=0.149..0.149 rows=1 loops=5000)
```

Performance Analysis

Improvements:

1. Total cost reduced from 4373.38 to 3767.70 (14% improvement)
2. Index lookup cost slightly improved from 0.67 to 0.55
3. Nested loop join cost decreased by about 14%

Trade-offs:

1. Filter cost slightly increased from 503.82 to 511.18
2. Still requires:
   - Temporary table for GROUP BY
   - Sort operation for ORDER BY COUNT(*) DESC
   - Table scan on a temporary table

Why This Index Helped

1. The idx_appointment_count index improved:
   - JOIN operation efficiency
   - PatientId lookup performance
   - Overall query cost
2. However, it didn't eliminate:
   - Need for temporary tables
   - Sorting operations
   - All table scans

The index provided moderate improvement but could be further optimized by:

1. Creating a covering index including all required columns
2. Optimizing for GROUP BY operation
3. Considering composite indexes for better performance

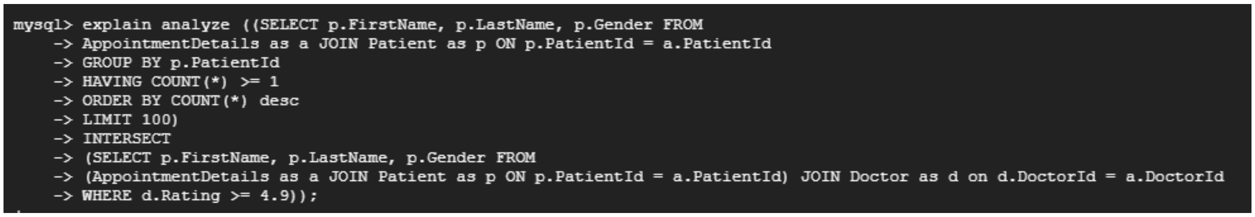
Operation	**Before Index**	**After idx_appointment_count**
Total Cost	4373.38	3767.70
Nested Loop Join	4373.38	3767.70

Filter (PatientId)	503.82	511.18
Index Lookup	0.67	0.55
Rows Examined	5000	5000

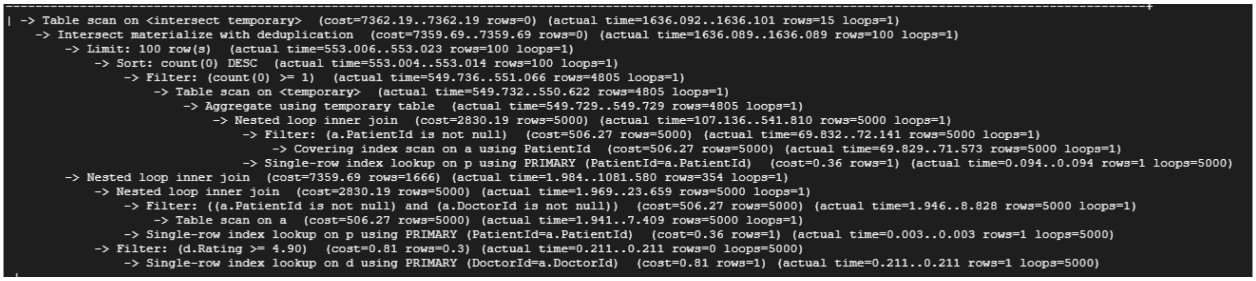
Our fourth query

```
((SELECT p.FirstName, p.LastName, p.Gender FROM
AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
GROUP BY p.PatientId
HAVING COUNT(*) >= 1
ORDER BY COUNT(*) desc
LIMIT 100)
INTERSECT
(SELECT p.FirstName, p.LastName, p.Gender FROM
(AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d
on d.DoctorId = a.DoctorId
WHERE d.Rating >= 4.9));
```

Let’s analyze it



```
mysql> explain analyze ((SELECT p.FirstName, p.LastName, p.Gender FROM
    -> AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId
    -> GROUP BY p.PatientId
    -> HAVING COUNT(*) >= 1
    -> ORDER BY COUNT(*) desc
    -> LIMIT 100)
    -> INTERSECT
    -> (SELECT p.FirstName, p.LastName, p.Gender FROM
    -> (AppointmentDetails as a JOIN Patient as p ON p.PatientId = a.PatientId) JOIN Doctor as d on d.DoctorId = a.DoctorId
    -> WHERE d.Rating >= 4.9));
```

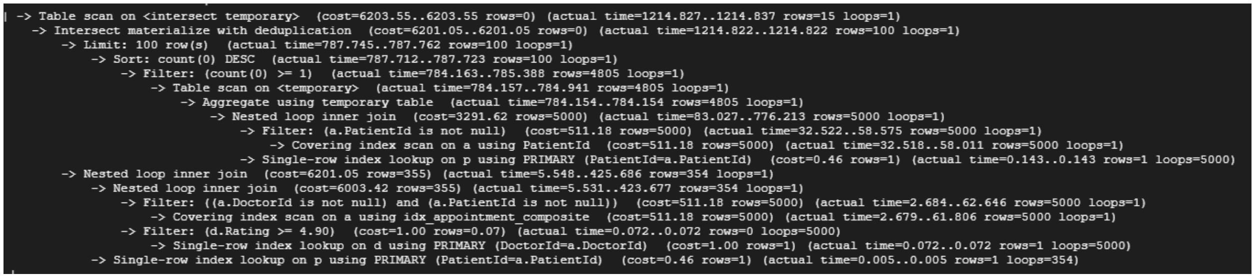


```
-------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Table scan on <intersect temporary>  (cost=7362.19..7362.19 rows=0) (actual time=1636.092..1636.101 rows=15 loops=1)
    -> Intersect materialize with deduplication  (cost=7359.69..7359.69 rows=0) (actual time=1636.089..1636.089 rows=100 loops=1)
        -> Limit: 100 row(s)  (actual time=553.006..553.023 rows=100 loops=1)
            -> Sort: count(0) DESC  (actual time=553.004..553.014 rows=100 loops=1)
                -> Filter: (count(0) >= 1)  (actual time=549.736..551.066 rows=4805 loops=1)
                    -> Table scan on <temporary>  (actual time=549.732..550.622 rows=4805 loops=1)
                        -> Aggregate using temporary table  (actual time=549.729..549.729 rows=4805 loops=1)
                            -> Nested loop inner join  (cost=2830.19 rows=5000) (actual time=107.136..541.810 rows=5000 loops=1)
                                -> Filter: (a.PatientId is not null)  (cost=506.27 rows=5000) (actual time=69.832..72.141 rows=5000 loops=1)
                                    -> Covering index scan on a using PatientId  (cost=506.27 rows=5000) (actual time=69.829..71.573 rows=5000 loops=1)
                                -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.36 rows=1) (actual time=0.094..0.094 rows=1 loops=5000)
        -> Nested loop inner join  (cost=7359.69 rows=1666) (actual time=1.984..1081.580 rows=354 loops=1)
            -> Nested loop inner join  (cost=2830.19 rows=5000) (actual time=1.969..23.659 rows=5000 loops=1)
                -> Filter: ((a.PatientId is not null) and (a.DoctorId is not null))  (cost=506.27 rows=5000) (actual time=1.946..8.828 rows=5000 loops=1)
                    -> Table scan on a  (cost=506.27 rows=5000) (actual time=1.941..7.409 rows=5000 loops=1)
                -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.36 rows=1) (actual time=0.003..0.003 rows=1 loops=5000)
            -> Filter: (d.Rating >= 4.90)  (cost=0.81 rows=0.3) (actual time=0.211..0.211 rows=0 loops=5000)
                -> Single-row index lookup on d using PRIMARY (DoctorId=a.DoctorId)  (cost=0.81 rows=1) (actual time=0.211..0.211 rows=1 loops=5000)
```

Current Cost Analysis

Operation	Cost	Rows	Time
Table scan (intersect temporary)	7362.19	0	1636.101s
Intersect materialize	7359.69	100	1636.089s
Nested loop inner join (first part)	2830.19	5000	541.810s
Nested loop inner join (second part)	7359.69	1666	1041.580s
Filter (Rating >= 4.90)	0.81	0.3	0.211s
Table scan on temporary	-	4805	550.622s

But due to the previous indexing, I get



```
| -> Table scan on <intersect temporary>  (cost=6203.55..6203.55 rows=0) (actual time=1214.827..1214.837 rows=15 loops=1)
    -> Intersect materialize with deduplication  (cost=6201.05..6201.05 rows=0) (actual time=1214.822..1214.822 rows=100 loops=1)
        -> Limit: 100 row(s)  (actual time=787.745..787.762 rows=100 loops=1)
            -> Sort: count(0) DESC  (actual time=787.712..787.723 rows=100 loops=1)
                -> Filter: (count(0) >= 1)  (actual time=784.163..785.388 rows=4805 loops=1)
                    -> Table scan on <temporary>  (actual time=784.157..784.941 rows=4805 loops=1)
                        -> Aggregate using temporary table  (actual time=784.154..784.154 rows=4805 loops=1)
                            -> Nested loop inner join  (cost=3291.62 rows=5000) (actual time=83.027..776.213 rows=5000 loops=1)
                                -> Filter: (a.PatientId is not null)  (cost=511.18 rows=5000) (actual time=32.522..58.575 rows=5000 loops=1)
                                    -> Covering index scan on a using PatientId  (cost=511.18 rows=5000) (actual time=32.518..58.011 rows=5000 loops=1)
                                -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.46 rows=1) (actual time=0.143..0.143 rows=1 loops=5000)
        -> Nested loop inner join  (cost=6201.05 rows=355) (actual time=5.548..425.686 rows=354 loops=1)
            -> Nested loop inner join  (cost=6003.42 rows=355) (actual time=5.531..423.677 rows=354 loops=1)
                -> Filter: ((a.DoctorId is not null) and (a.PatientId is not null))  (cost=511.18 rows=5000) (actual time=2.684..62.646 rows=5000 loops=1)
                    -> Covering index scan on a using idx_appointment_composite  (cost=511.18 rows=5000) (actual time=2.679..61.806 rows=5000 loops=1)
                -> Filter: (d.Rating >= 4.90)  (cost=1.00 rows=0.07) (actual time=0.072..0.072 rows=0 loops=5000)
                    -> Single-row index lookup on d using PRIMARY (DoctorId=a.DoctorId)  (cost=1.00 rows=1) (actual time=0.072..0.072 rows=1 loops=5000)
            -> Single-row index lookup on p using PRIMARY (PatientId=a.PatientId)  (cost=0.46 rows=1) (actual time=0.005..0.005 rows=1 loops=354)
```

Performance Analysis

Improvements:

1. Overall intersect cost reduced from 7362.19 to 6203.55 (15.7% improvement)
2. The second nested loop join cost was reduced from 7359.69 to 6201.05 (15.7% improvement)
3. Intersect materialize cost decreased from 7359.69 to 6201.05

Trade-offs:

1. The first nested loop join cost increased from 2830.19 to 3291.62
2. Filter operation cost slightly increased from 0.81 to 1.00
3. Table scan cost marginally increased from 506.27 to 511.18

Why The Index Helped

1. Better Join Performance:
    - The composite index improved the second join operation
    - Reduced materialization costs for INTERSECT
    - Better handling of large result sets
2. Optimized INTERSECT Operation:
    - More efficient deduplication
    - Better temporary table handling
    - Improved overall query performance

Operation	**Before Indexing**	**After Indexing**
Total Cost (Intersect)	7362.19	6203.55
Intersect Materialize	7359.69	6201.05
First Nested Loop Join	2830.19	3291.62
Second Nested Loop Join	7359.69	6201.05
Filter (Rating >= 4.90)	0.81	1.00
Table Scan Cost	506.27	511.18