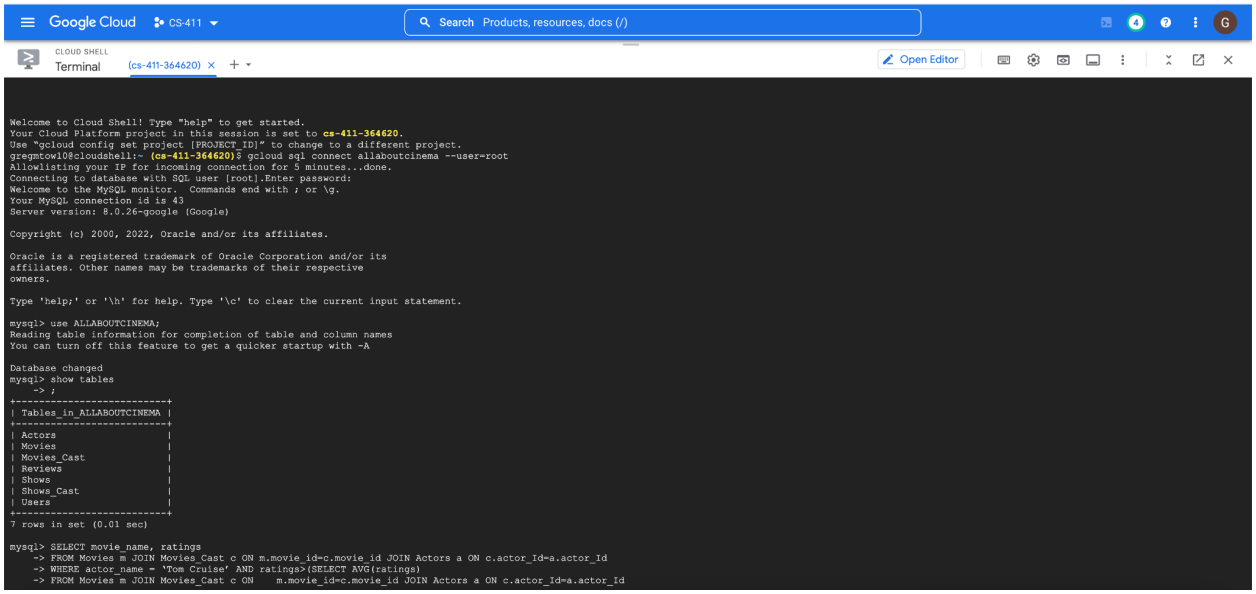
GCP Connection



DDL Commands For Each Table

```
mysql> use ALLABOUTCINEMA;
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables
    -> ;
+--------------------------+
| Tables_in_ALLABOUTCINEMA |
+--------------------------+
| Actors                   |
| Movies                   |
| Movies_Cast              |
| Reviews                  |
| Shows                    |
| Shows_Cast               |
| Users                    |
+--------------------------+
7 rows in set (0.01 sec)
```

Movies Table

```
CREATE TABLE Movies(
	`movie_id` VARCHAR(100) PRIMARY KEY,
	`movie_name` VARCHAR(255),
	`country` VARCHAR(100),
	`director` VARCHAR(255),
	`release_year` DOUBLE,
	`length` DOUBLE,
	`genre` VARCHAR(100),
	`ratings` DOUBLE
);
```

Actors Table

```
CREATE TABLE Actors(
	`actor_Id` VARCHAR(100) PRIMARY KEY,
	`actor_name` VARCHAR(100),
	`birthYear` DOUBLE
);
```

Movies Cast Table

```
CREATE TABLE Movies_Cast(
	`unique_id` DOUBLE PRIMARY KEY,
	`movie_id` VARCHAR(100),
	`actor_Id` VARCHAR(100),
	FOREIGN KEY (movie_id) references Movies(movie_id) ON DELETE CASCADE,
	FOREIGN KEY (actor_Id) references Actors(actor_Id) ON DELETE CASCADE
);
```

Shows Table

```
CREATE TABLE Shows(
	`show_id` VARCHAR(100) PRIMARY KEY,
	`titleType` VARCHAR(100),
	`release_year` DOUBLE,
	`episode_length` DOUBLE,
	`genre` VARCHAR(100),
	`show_name` VARCHAR(100),
	`country` VARCHAR(100),
	`director` VARCHAR(100),
	`ratings` DOUBLE
);
```

Shows Cast Table

```
CREATE TABLE Shows_Cast(
	`unique_id` DOUBLE PRIMARY KEY,
	`show_id` VARCHAR(100),
	`actor_Id` VARCHAR(100),
	FOREIGN KEY (show_id) references Shows(show_id) ON DELETE CASCADE,
	FOREIGN KEY (actor_Id) references Actors(actor_Id) ON DELETE CASCADE

);
```

Reviews Table

```
CREATE TABLE Reviews(
   `review_id` DOUBLE PRIMARY KEY,
   `user_id` DOUBLE,
   `rating` DOUBLE,
   `content_id` VARCHAR(100),
   `review_text` TEXT,
   FOREIGN KEY (user_id) references Users(user_id) ON DELETE CASCADE
);
```

Users Table

```
CREATE TABLE Users(
	`user_id` DOUBLE PRIMARY KEY,
	`first_name` VARCHAR(100),
	`last_name` VARCHAR(100),
	`password` VARCHAR(100)
);
```

Proof of 1000 rows in tables

```
mysql> SELECT Count(show_id) FROM Shows;
+----------------+
| Count(show_id) |
+----------------+
|           3627 |
+----------------+
1 row in set (0.00 sec)

mysql> SELECT Count(movie_id) FROM Movies;
+-----------------+
| Count(movie_id) |
+-----------------+
|           44729 |
+-----------------+
1 row in set (0.02 sec)

mysql> SELECT Count(actor_Id) FROM Actors;
+-----------------+
| Count(actor_Id) |
+-----------------+
|          566164 |
+-----------------+
1 row in set (0.26 sec)

mysql> SELECT Count(unique_id) FROM Movies_Cast;
+------------------+
| Count(unique_id) |
+------------------+
|           434696 |
+------------------+
1 row in set (0.20 sec)

mysql> SELECT Count(unique_id) FROM Shows_Cast;
+------------------+
| Count(unique_id) |
+------------------+
|            35050 |
+------------------+
1 row in set (0.01 sec)

mysql>
```

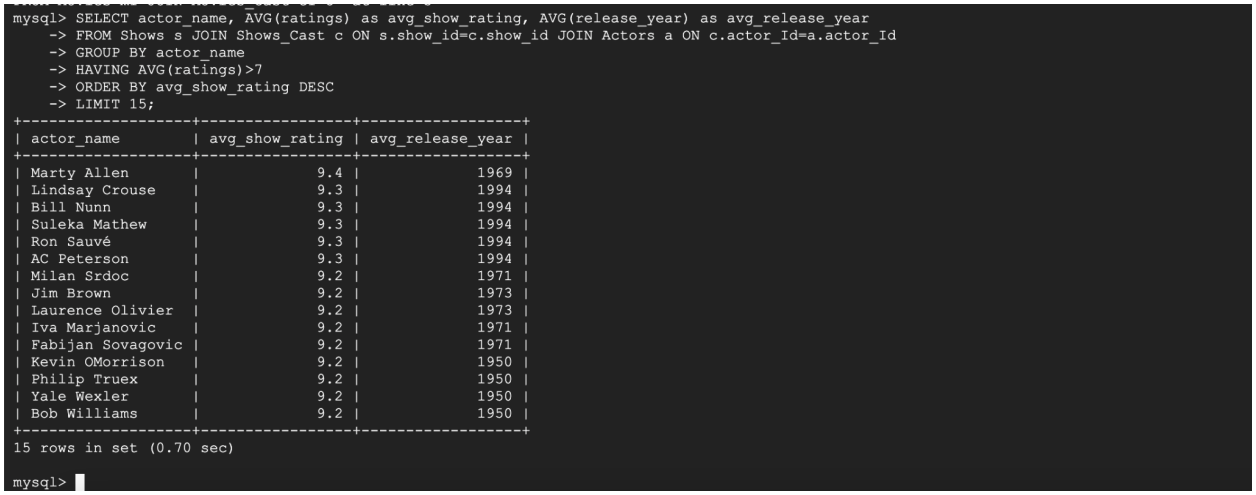
*** Reviews Table and Users Table are for user input, so they have 2 rows each from our inputs.**

2 Advanced Queries with Screenshots of top 15 rows

Find all of the actors with average show rating greater than 7.0. Order in descending order by avg_show_rating, and limit the number of results to 15.

SELECT actor_name, AVG(ratings) as avg_show_rating, AVG(release_year) as avg_release_year
FROM Shows s JOIN Shows_Cast c ON s.show_id=c.show_id JOIN Actors a ON c.actor_Id=a.actor_Id
GROUP BY actor_name
HAVING AVG(ratings)>7
ORDER BY avg_show_rating DESC
LIMIT 15;

*included join of multiple tables and Aggregation via Group By



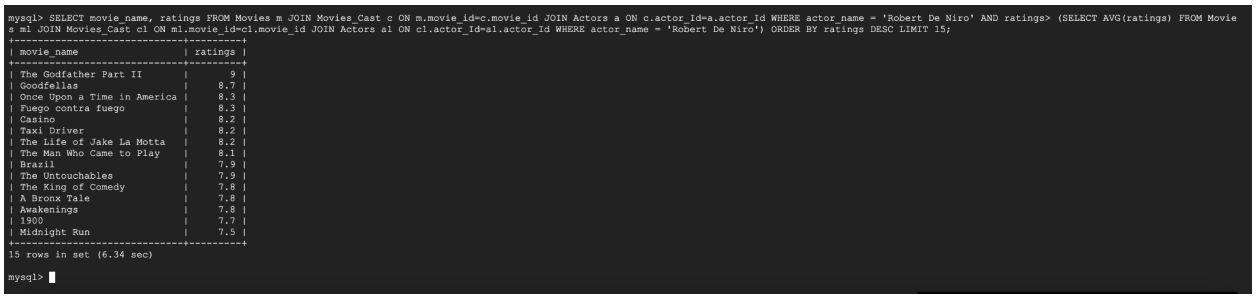
```
mysql> SELECT actor_name, AVG(ratings) as avg_show_rating, AVG(release_year) as avg_release_year
    -> FROM Shows s JOIN Shows_Cast c ON s.show_id=c.show_id JOIN Actors a ON c.actor_Id=a.actor_Id
    -> GROUP BY actor_name
    -> HAVING AVG(ratings)>7
    -> ORDER BY avg_show_rating DESC
    -> LIMIT 15;
+------------------+-----------------+------------------+
| actor_name       | avg_show_rating | avg_release_year |
+------------------+-----------------+------------------+
| Marty Allen      |             9.4 |             1969 |
| Lindsay Crouse   |             9.3 |             1994 |
| Bill Nunn        |             9.3 |             1994 |
| Suleka Mathew    |             9.3 |             1994 |
| Ron Sauvé        |             9.3 |             1994 |
| AC Peterson      |             9.3 |             1994 |
| Milan Srdoc      |             9.2 |             1971 |
| Jim Brown        |             9.2 |             1973 |
| Laurence Olivier |             9.2 |             1973 |
| Iva Marjanovic   |             9.2 |             1971 |
| Fabijan Sovagovic|             9.2 |             1971 |
| Kevin OMorrison  |             9.2 |             1950 |
| Philip Truex     |             9.2 |             1950 |
| Yale Wexler      |             9.2 |             1950 |
| Bob Williams     |             9.2 |             1950 |
+------------------+-----------------+------------------+
15 rows in set (0.70 sec)

mysql>
```

Find all movies starring Robert De Niro with ratings higher than his average movie rating. Order by ratings in descending order, and limit the number of results to 15.

```
SELECT movie_name, ratings
FROM Movies m JOIN Movies_Cast c ON m.movie_id=c.movie_id JOIN Actors a ON
c.actor_Id=a.actor_Id
WHERE actor_name = 'Robert De Niro' AND ratings>(SELECT AVG(ratings)
                                                 FROM Movies m1 JOIN Movies_Cast c1
                                                 ON m1.movie_id=c1.movie_id JOIN Actors
                                                 a1 ON c1.actor_Id=a1.actor_Id
                                                 WHERE actor_name = 'Robert De Niro')
ORDER BY ratings DESC
LIMIT 15;
```

*included join of multiple tables and subqueries



```
mysql> SELECT movie_name, ratings FROM Movies m JOIN Movies_Cast c ON m.movie_id=c.movie_id JOIN Actors a ON c.actor_Id=a.actor_Id WHERE actor_name = 'Robert De Niro' AND ratings> (SELECT AVG(ratings) FROM Movie
s m1 JOIN Movies_Cast c1 ON m1.movie_id=c1.movie_id JOIN Actors a1 ON c1.actor_Id=a1.actor_Id WHERE actor_name = 'Robert De Niro') ORDER BY ratings DESC LIMIT 15;
+------------------------------+---------+
| movie_name                   | ratings |
+------------------------------+---------+
| The Godfather Part II        |       9 |
| Goodfellas                   |     8.7 |
| Once Upon a Time in America  |     8.3 |
| Fuego contra fuego           |     8.3 |
| Casino                       |     8.2 |
| Taxi Driver                  |     8.2 |
| The Life of Jake La Motta    |     8.2 |
| The Man Who Came to Play     |     8.1 |
| Brazil                       |     7.9 |
| The Untouchables             |     7.9 |
| The King of Comedy           |     7.8 |
| A Bronx Tale                 |     7.8 |
| Awakenings                   |     7.8 |
| 1900                         |     7.7 |
| Midnight Run                 |     7.5 |
+------------------------------+---------+
15 rows in set (6.34 sec)

mysql>
```

Indexing Analysis

First Advanced Query Analysis

EXPLAIN ANALYZE
SELECT actor_name, AVG(ratings) as avg_show_rating, AVG(release_year) as avg_release_year
FROM Shows s JOIN Shows_Cast c ON s.show_id=c.show_id JOIN Actors a ON c.actor_Id=a.actor_Id
GROUP BY actor_name
HAVING AVG(ratings)>7
ORDER BY avg_show_rating DESC
LIMIT 15;

Default Index (Primary Key Indexes)

```
 BY actor_name HAVING AVG(ratings)>7 ORDER BY avg_show_rating DESC LIMIT 15;
| EXPLAIN
| -> Limit: 15 row(s)  (actual time=472.125..472.128 rows=15 loops=1)
    -> Sort: avg_show_rating DESC  (actual time=472.124..472.126 rows=15 loops=1)
        -> Filter: (avg(s.ratings) > 7)  (actual time=461.553..466.802 rows=10249 loops=1)
            -> Table scan on <temporary>  (actual time=0.002..2.995 rows=16162 loops=1)
                -> Aggregate using temporary table  (actual time=461.546..465.472 rows=16162 loops=1)
                    -> Nested loop inner join  (cost=26746.65 rows=33383) (actual time=0.119..412.712 rows=30132 loops=1)
                        -> Nested loop inner join  (cost=15062.60 rows=33383) (actual time=0.094..124.381 rows=35043 loops=1)
                            -> Filter: ((c.show_id is not null) and (c.actor_Id is not null))  (cost=3378.55 rows=33383) (actual time=0.032..25.661 rows=35050 loops=1)
                                -> Table scan on c  (cost=3378.55 rows=33383) (actual time=0.030..20.865 rows=35050 loops=1)
                            -> Index lookup on s using show_id (show_id=c.show_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=35050)
                        -> Index lookup on a using actor_id (actor_Id=c.actor_Id)  (cost=0.25 rows=1) (actual time=0.007..0.008 rows=1 loops=35043)
 |
1 row in set (0.48 sec)

mysql>
```

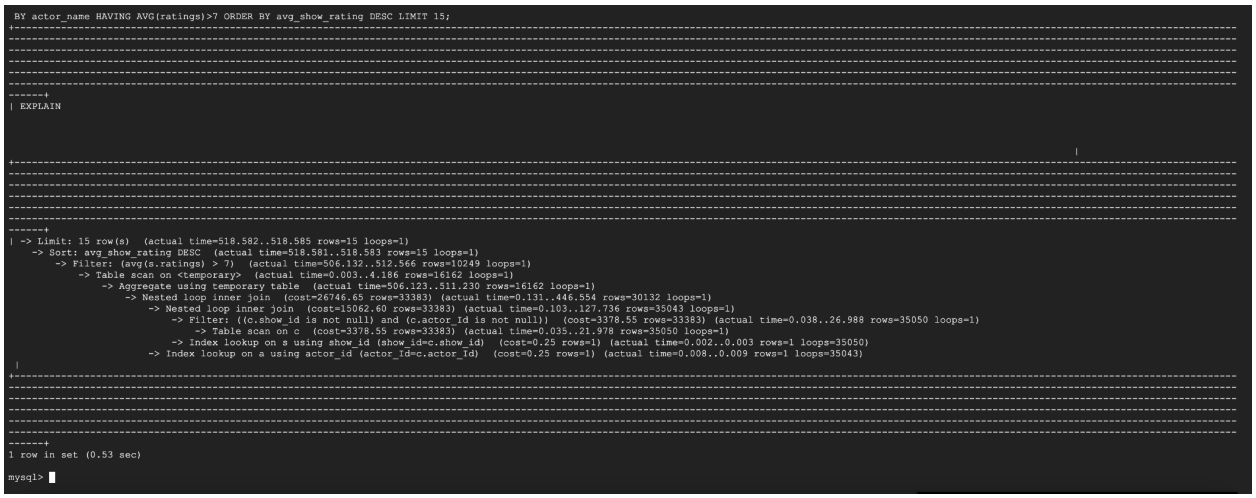
Index 1: Indexed on actor_name in Actors Table since we were grouping by it

```
mysql> EXPLAIN ANALYZE SELECT actor_name, AVG(ratings) as avg_show_rating, AVG(release_year) as avg_release_year FROM Shows s JOIN Shows_Cast c ON s.show_id=c.show_id JOIN Actors a ON c.actor_Id=a.actor_Id GROUP
 BY actor_name HAVING AVG(ratings)>7 ORDER BY avg_show_rating DESC LIMIT 15;
| EXPLAIN
| -> Limit: 15 row(s)  (actual time=449.349..449.351 rows=15 loops=1)
    -> Sort: avg_show_rating DESC  (actual time=449.348..449.350 rows=15 loops=1)
        -> Filter: (avg(s.ratings) > 7)  (actual time=438.100..443.709 rows=10249 loops=1)
            -> Table scan on <temporary>  (actual time=0.002..3.143 rows=16162 loops=1)
                -> Aggregate using temporary table  (actual time=438.093..442.201 rows=16162 loops=1)
                    -> Nested loop inner join  (cost=26746.65 rows=33383) (actual time=0.116..394.007 rows=30132 loops=1)
                        -> Nested loop inner join  (cost=15062.60 rows=33383) (actual time=0.094..122.407 rows=35043 loops=1)
                            -> Filter: ((c.show_id is not null) and (c.actor_Id is not null))  (cost=3378.55 rows=33383) (actual time=0.036..25.008 rows=35050 loops=1)
                                -> Table scan on c  (cost=3378.55 rows=33383) (actual time=0.034..20.285 rows=35050 loops=1)
                            -> Index lookup on s using show_id (show_id=c.show_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=35050)
                        -> Index lookup on a using actor_id (actor_Id=c.actor_Id)  (cost=0.25 rows=1) (actual time=0.007..0.008 rows=1 loops=35043)
 |
1 row in set (0.45 sec)
```

Commands used:

CREATE INDEX actor_name ON Actors(actor_name)
Show index from Actors
DROP INDEX actor_name ON Actors

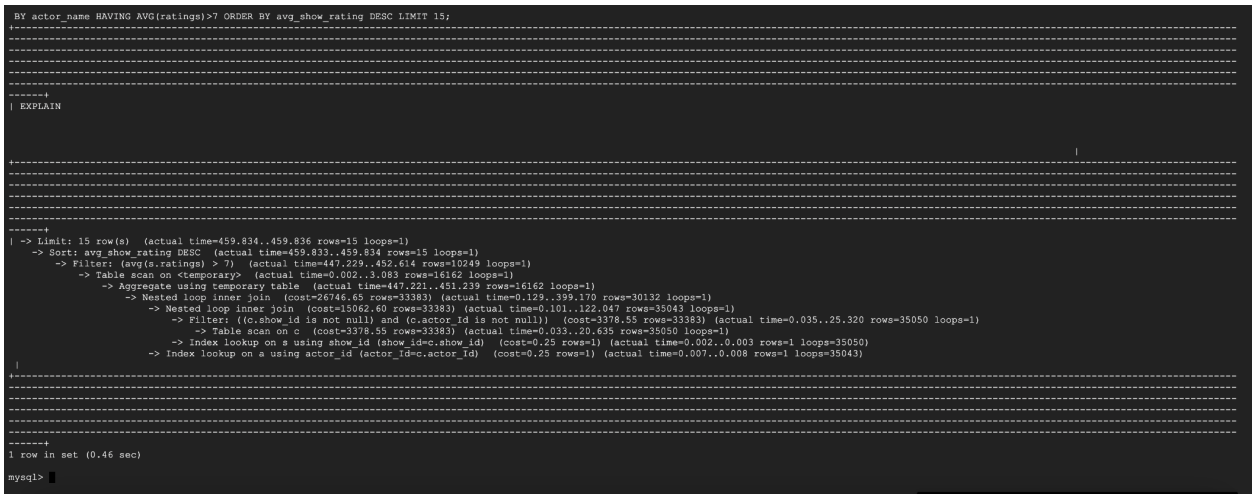
Index 2: Indexed on ratings in Shows Table because we were doing an aggregation of ratings



Commands used:

CREATE INDEX ratings ON Shows(ratings)
SHOW INDEX from Shows
DROP INDEX ratings ON Shows

Combination of Index 1 and 2: Includes indexes on ratings in the Shows Table and actor_name



Commands used:

CREATE INDEX actor_name ON Actors(actor_name)

```
Show index from Actors
CREATE INDEX ratings ON Shows(ratings)
SHOW INDEX from Shows
DROP INDEX actor_name ON Actors
DROP INDEX ratings ON Shows
```

Analysis: Overall, we can see that smart indexing does work here as indexing by actor_name which we grouped by lowered our execution time from 0.48s to 0.45s. However, the original query with the Primary Key indexes alone was already fast, so while it made it faster, it did not make it much faster. This is also due to the amount of data we have for our Shows Table as we only have about 3700 shows available in our dataset. If we had more shows on a similar level to our amount of movies, the execution would be noticeably faster. Notice that actor_name was a smart choice for indexing as we grouped the actor name compared to ratings. An index on ratings actually slowed our execution time from 0.48s to 0.53s. We chose ratings as we are finding the average ratings of all actors in tv shows, but indexing on it is not a smart decision. This may be because we are creating too much data reading and writing with this specific index. The trend follows for our third indexing design as we used both the actor_name and ratings index together which gave us a slower execution time than just using the actor_name index. This makes sense as the ratings index slows our execution time overall, so it would output a slower time than just using the actor_name index. Our second advanced query shows us how a larger dataset allows us to better see the effects of indexing.

Second Advanced Query Analysis

```
EXPLAIN ANALYZE
SELECT movie_name, ratings
FROM Movies m JOIN Movies_Cast c ON m.movie_id=c.movie_id JOIN Actors a ON
c.actor_Id=a.actor_Id
WHERE actor_name = 'Robert De Niro' AND ratings>
    (SELECT AVG(ratings)
    FROM Movies m1 JOIN Movies_Cast c1 ON m1.movie_id=c1.movie_id JOIN Actors
    a1 ON c1.actor_Id=a1.actor_Id
    WHERE actor_name = 'Robert De Niro')
ORDER BY ratings DESC
LIMIT 15;
```

Default Index (Primary Key Indexes)

```
-----------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=7518.565..7518.568 rows=15 loops=1)
    -> Sort: m.ratings DESC, limit input to 15 row(s) per chunk  (actual time=7518.564..7518.566 rows=15 loops=1)
        -> Stream results  (cost=238997.69 rows=14035) (actual time=6621.539..7518.441 rows=23 loops=1)
            -> Nested loop inner join  (cost=238997.69 rows=14035) (actual time=6621.530..7518.329 rows=23 loops=1)
                -> Nested loop inner join  (cost=189873.80 rows=140354) (actual time=5316.068..6952.258 rows=69441 loops=1)
                    -> Filter: ((c.movie_id is not null) and (c.actor_Id is not null))  (cost=42487.40 rows=421104) (actual time=0.135..287.746 rows=434696 loops=1)
                        -> Table scan on c  (cost=42487.40 rows=421104) (actual time=0.133..235.591 rows=434696 loops=1)
                    -> Filter: (m.ratings > (select #2))  (cost=0.25 rows=0) (actual time=0.015..0.015 rows=0 loops=434696)
                        -> Index lookup on m using movie_id (movie_id=c.movie_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=434696)
                        -> Select #2 (subquery in condition; run only once)
                            -> Aggregate: avg(m1.ratings)  (cost=341471.24 rows=42110) (actual time=5315.218..5315.219 rows=1 loops=1)
                                -> Nested loop inner join  (cost=337260.20 rows=42110) (actual time=2914.712..5314.952 rows=46 loops=1)
                                    -> Nested loop inner join  (cost=189873.80 rows=421104) (actual time=0.015..1712.834 rows=434538 loops=1)
                                        -> Filter: ((c1.movie_id is not null) and (c1.actor_Id is not null))  (cost=42487.40 rows=421104) (actual time=0.007..331.047 rows=434696 loops=1)
                                            -> Table scan on c1  (cost=42487.40 rows=421104) (actual time=0.007..269.647 rows=434696 loops=1)
                                        -> Index lookup on m1 using movie_id (movie_id=c1.movie_id)  (cost=0.25 rows=1) (actual time=0.002..0.003 rows=1 loops=434696)
                                    -> Filter: (a1.actor_name = 'Robert De Niro')  (cost=0.25 rows=0) (actual time=0.008..0.008 rows=0 loops=434538)
                                        -> Index lookup on a1 using actor_id (actor_Id=c1.actor_Id)  (cost=0.25 rows=1) (actual time=0.008..0.008 rows=1 loops=434538)
                -> Filter: (a.actor_name = 'Robert De Niro')  (cost=0.25 rows=0) (actual time=0.008..0.008 rows=0 loops=69441)
                    -> Index lookup on a using actor_id (actor_Id=c.actor_Id)  (cost=0.25 rows=1) (actual time=0.007..0.008 rows=1 loops=69441)
 |
+----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------+
1 row in set (7.52 sec)

mysql>
```

Index 1: Indexed on actor_name as we were querying for Robert De Niro's movies

```
-----------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=623.996..623.998 rows=15 loops=1)
    -> Sort: m.ratings DESC, limit input to 15 row(s) per chunk  (actual time=623.995..623.996 rows=15 loops=1)
        -> Stream results  (cost=15538.25 rows=1404) (actual time=493.465..623.929 rows=23 loops=1)
            -> Nested loop inner join  (cost=15538.25 rows=1404) (actual time=493.459..623.871 rows=23 loops=1)
                -> Filter: (c.actor_Id = a.actor_Id)  (cost=4589.55 rows=4211) (actual time=169.174..321.390 rows=46 loops=1)
                    -> Inner hash join (<hash>(c.actor_Id)=<hash>(a.actor_Id))  (cost=4589.55 rows=4211) (actual time=169.169..321.325 rows=46 loops=1)
                        -> Filter: (c.movie_id is not null)  (cost=4589.20 rows=42110) (actual time=0.021..260.088 rows=434696 loops=1)
                            -> Table scan on c  (cost=4589.20 rows=421104) (actual time=0.020..222.039 rows=434696 loops=1)
                        -> Hash
                            -> Index lookup on a using actor_name (actor_name='Robert De Niro')  (cost=0.35 rows=1) (actual time=0.044..0.050 rows=1 loops=1)
                -> Filter: (m.ratings > (select #2))  (cost=0.25 rows=0) (actual time=6.575..6.575 rows=0 loops=46)
                    -> Index lookup on m using movie_id (movie_id=c.movie_id)  (cost=0.25 rows=1) (actual time=0.012..0.013 rows=1 loops=46)
                    -> Select #2 (subquery in condition; run only once)
                        -> Aggregate: avg(m1.ratings)  (cost=15959.36 rows=4211) (actual time=301.682..301.683 rows=1 loops=1)
                            -> Nested loop inner join  (cost=15538.25 rows=4211) (actual time=157.848..301.611 rows=46 loops=1)
                                -> Filter: (c1.actor_Id = a1.actor_Id)  (cost=4589.55 rows=4211) (actual time=157.808..300.986 rows=46 loops=1)
                                    -> Inner hash join (<hash>(c1.actor_Id)=<hash>(a1.actor_Id))  (cost=4589.55 rows=4211) (actual time=157.804..300.931 rows=46 loops=1)
                                        -> Filter: (c1.movie_id is not null)  (cost=4589.20 rows=42110) (actual time=0.010..241.067 rows=434696 loops=1)
                                            -> Table scan on c1  (cost=4589.20 rows=421104) (actual time=0.009..209.100 rows=434696 loops=1)
                                        -> Hash
                                            -> Index lookup on a1 using actor_name (actor_name='Robert De Niro')  (cost=0.35 rows=1) (actual time=0.026..0.028 rows=1 loops=1)
                                -> Index lookup on m1 using movie_id (movie_id=c1.movie_id)  (cost=0.25 rows=1) (actual time=0.012..0.013 rows=1 loops=46)
 |
+----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------+
1 row in set (0.63 sec)

mysql> SELECT movie_name, ratings FROM Movies m JOIN Movies_Cast c ON m.movie_id=c.movie_id JOIN Actors a ON c.actor_Id=a.actor_Id WHERE actor_name = 'Robert De Niro' AND ratings> (SELECT AVG(ratings) FROM Movie
```

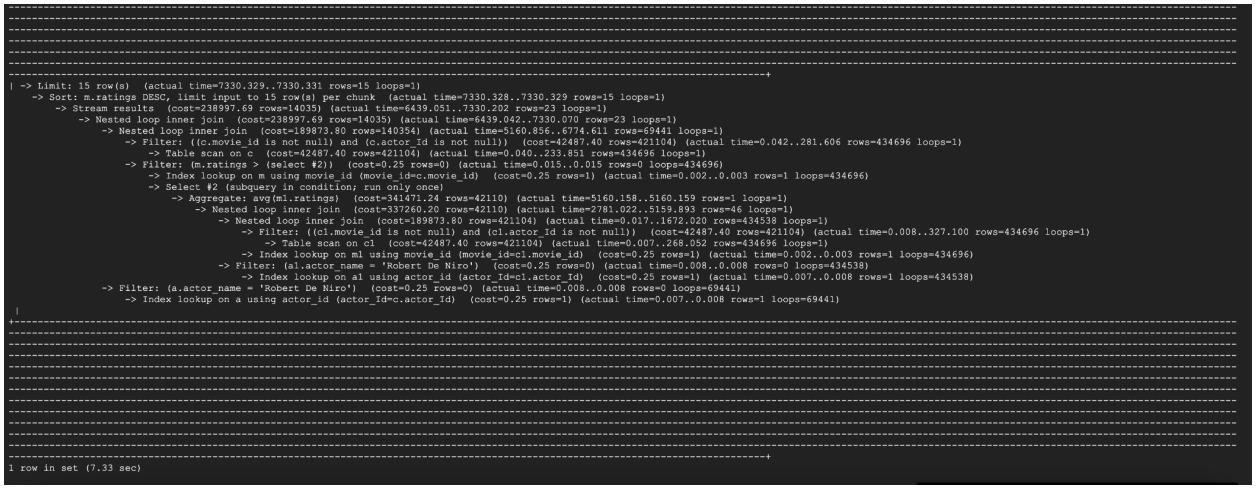
Commands used:

CREATE INDEX actor_name ON Actors(actor_name)

Show index from Actors

DROP INDEX actor_name ON Actors

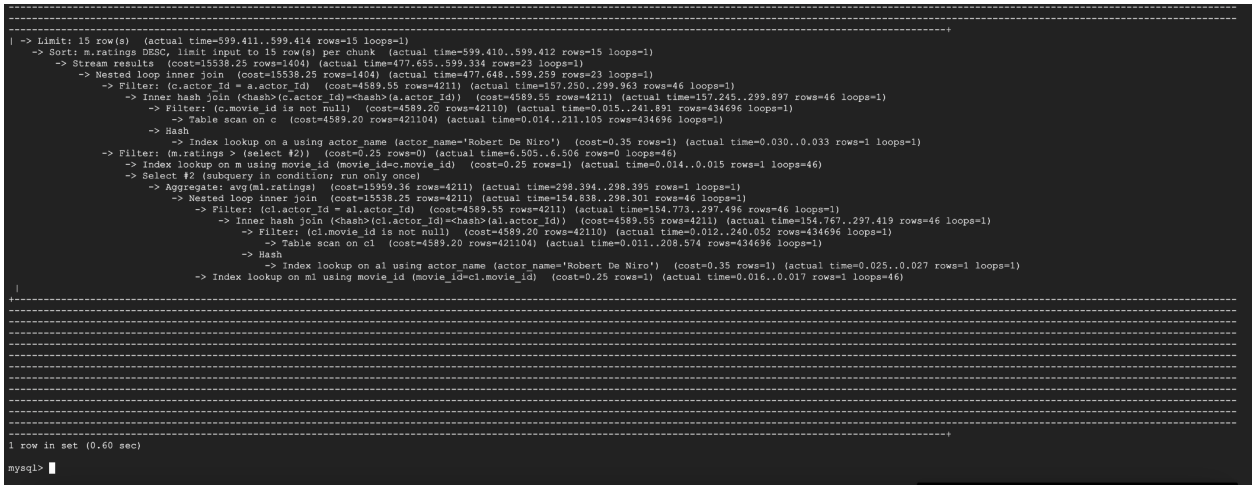
Index 2: Indexed on movie_name as we were returning the names of Robert De Niro's movies



Commands used:

CREATE INDEX movie_name ON Movies(movie_name)
SHOW INDEX from Movies
DROP INDEX movie_name ON Movies

Combination of Index 1 and 2: Includes indexes on movie_name and actor_name



Commands used:

CREATE INDEX actor_name ON Actors(actor_name)
Show index from Actors
CREATE INDEX movie_name ON Movies(movie_name)
SHOW INDEX from Movies
DROP INDEX actor_name ON Actors
DROP INDEX ratings ON Shows
DROP INDEX movie_name ON Movies

Analysis: Continuing from the last query we can see that a larger dataset does give a much longer run time as we had a 7.52s execution time using the EXPLAIN ANALYZE Command. The query itself was long as well at 6.34s as shown in our 15 row output of the query earlier in the document. We chose to first index on actor_name as we were essentially querying for Robert De Niro's movies twice with the presence of our subquery. This worked very well as our execution time lowered to 0.63s from 7.52s. Indexing on actor_name clearly is a smart decision and will help in the future to know to index on attributes you are grouping by or finding records for. Since we're looking for shows where John De Niro was a part of, it makes sense that indexing by the actor_name helps. We next tried to index on movie_name since we would need the names of the movies that starred Robert De Niro. This offered some improvement to our execution time as it dropped from 7.52s to 7.33s. We believe it helped, but did not help as much as the grouping by actor_name had already occurred before selecting movie_name, so we had our movie_name already on hand. Indexing by movie_name probably made little difference because most of the data we require in this query depends on the actor. We're not looking for movies depending on the name of the movie, but rather depending on whether or not Robert De Niro was in it. Finally, we used both the movie_name and actor_name index together and obtained expected results as it was our fastest execution time at 0.60s which was slightly better than just using actor_name alone. This was expected as both indexes on their own improved execution time, so putting them together gave the best time. It also made sense that it was only a little better than just the actor_name index as the movie_name index was only slightly effective on its own.