Dream to Stream

Stage 3 Database Implementation and Indexing

Data Definition Language (DDL) commands

```
CREATE TABLE Platform(
        platformId INT,
        platform VARCHAR(255),
        country VARCHAR(100),

        PRIMARY KEY(platformId, platform)
);

CREATE TABLE CastGroup (
        castId INT NOT NULL,
        firstName VARCHAR(255),
        lastName VARCHAR(255),

        PRIMARY KEY (castId)
);

CREATE TABLE Shows (
        name VARCHAR(255),
        country VARCHAR(100),
        releaseYear INT,
        tvRating VARCHAR(20),
        seasons REAL,
        genre VARCHAR(255),
        platform VARCHAR(255),

        PRIMARY KEY (name)
);

CREATE TABLE Movies(
        name VARCHAR(255),
        country VARCHAR(100),
        releaseYear INT,
        tvRating VARCHAR(20),
        length REAL,
        genre VARCHAR(255),
        platform VARCHAR(255),

        PRIMARY KEY (name)
```

```
);

CREATE TABLE Users(
        id INT PRIMARY KEY,
        firstName VARCHAR(255),
        lastName VARCHAR(255),
        birthDate DATE,
        country VARCHAR(60)
);

CREATE TABLE ShowRating(
        id INT NOT NULL,
        name VARCHAR(255) NOT NULL,
        date DATE,
        value REAL,
        comments VARCHAR(255),

        PRIMARY KEY(id, name),
        FOREIGN KEY (id) REFERENCES Users(id)
                ON DELETE CASCADE,
        FOREIGN KEY (name) REFERENCES Shows(name)
                ON DELETE CASCADE
);

CREATE TABLE MovieRating(
        id INT NOT NULL,
        name VARCHAR(255) NOT NULL,
        date DATE,
        value REAL,
        comments VARCHAR(255),

        PRIMARY KEY(id, name),
        FOREIGN KEY (id) REFERENCES Users(id)
                ON DELETE CASCADE,
        FOREIGN KEY (name) REFERENCES Movies(name)
                ON DELETE CASCADE
);

CREATE TABLE CustomerGroups(
        groupID INT,
        groupName VARCHAR(255),
        contact VARCHAR (255),
```

```
        PRIMARY KEY (groupID, groupName)
);

CREATE TABLE ShowIncludes(
        castId INT NOT NULL,
        name VARCHAR(255),
        role VARCHAR(255),
        PRIMARY KEY (castId, name),

        FOREIGN KEY (castId) REFERENCES CastGroup(castId)
                ON DELETE CASCADE,
        FOREIGN KEY (name) REFERENCES Shows(name)
                ON DELETE CASCADE
);

CREATE TABLE MovieIncludes(
        castId INT NOT NULL,
        name VARCHAR(255),
        role VARCHAR(255),
        PRIMARY KEY (castId, name),

        FOREIGN KEY (castId) REFERENCES CastGroup(castId)
                ON DELETE CASCADE,
        FOREIGN KEY (name) REFERENCES Movies(name)
                ON DELETE CASCADE
);
```

1000 Entries Into 3 Tables

Movies Table

```
Database changed
mysql> SELECT COUNT(*) FROM Movies;
+----------+
| COUNT(*) |
+----------+
|    15922 |
+----------+
1 row in set (0.02 sec)
```

Shows Table

```
mysql> SELECT COUNT(*) FROM Shows;
+----------+
| COUNT(*) |
+----------+
|     6232 |
+----------+
1 row in set (0.01 sec)
```

MovieRating Table

```
mysql> SELECT COUNT(*) FROM MovieRating;
+----------+
| COUNT(*) |
+----------+
|     4922 |
+----------+
1 row in set (0.00 sec)
```

Two Advanced Queries, Top 15 Rows Output, and Explain Analyze Output

1. Advanced Query 1: Find the Horror Movies from each year from 2005-2010 with ratings equal to the highest rating that year and sort them by name.

Advanced Query:

```
(SELECT "Movies", m.releaseYear as yr, COUNT(*)
        FROM Movies as m
        WHERE releaseYear > 2000 AND genre LIKE "%horror%" OR
        genre LIKE "%Horror%"
        GROUP BY m.releaseYear
)
UNION
(SELECT "Shows", s.releaseYear as yr, COUNT(*), AVG(s.rating)
        FROM Shows as s
        WHERE releaseYear > 2000 AND genre LIKE "%horror%" OR
        genre LIKE "%Horror%"
        GROUP BY s.releaseYear
)
ORDER BY yr DESC
LIMIT 15;
```

Output:

```
mysql> (SELECT "Movies", m.releaseYear as yr, COUNT(*)
    -> FROM Movies as m
    -> WHERE releaseYear > 2000 AND genre LIKE "%horror%" OR genre LIKE "%Horror%"
    -> GROUP BY m.releaseYear
    -> )
    -> UNION
    -> (SELECT "Shows", s.releaseYear as yr, COUNT(*)
    -> FROM Shows as s
    -> WHERE releaseYear > 2000 AND genre LIKE "%horror%" OR genre LIKE "%Horror%"
    -> GROUP BY s.releaseYear
    -> )
    -> ORDER BY yr DESC
    -> LIMIT 15;
+--------+------+----------+
| Movies | yr   | COUNT(*) |
+--------+------+----------+
| Shows  | 2021 |       15 |
| Movies | 2021 |      226 |
| Movies | 2020 |      138 |
| Shows  | 2020 |       25 |
| Shows  | 2019 |       22 |
| Movies | 2019 |      144 |
| Shows  | 2018 |       19 |
| Movies | 2018 |      134 |
| Shows  | 2017 |        7 |
| Movies | 2017 |       98 |
| Movies | 2016 |       81 |
| Shows  | 2016 |       10 |
| Shows  | 2015 |       11 |
| Movies | 2015 |       58 |
| Shows  | 2014 |       10 |
+--------+------+----------+
15 rows in set (0.03 sec)
```

a. EXPLAIN ANALYZE – actual time = 33.429 - 33.432
    i. 33.429 to first row

ii. 33.432 milliseconds to 15 rows

```
Limit: 15 row(s)  (actual time=33.429..33.432 rows=15 loops=1)
-> Sort: m.`name`, limit input to 15 row(s) per chunk  (actual time=33.428..33.430 rows=15 loops=1)
   -> Stream results  (cost=3591.75 rows=0) (actual time=19.929..33.386 rows=27 loops=1)
      -> Nested loop inner join  (cost=3591.75 rows=0) (actual time=19.924..33.365 rows=27 loops=1)
         -> Nested loop inner join  (cost=2222.15 rows=547) (actual time=0.133..14.521 rows=448 loops=1)
            -> Filter: (mr.`value` is not null)  (cost=499.45 rows=4922) (actual time=0.065..2.110 rows=4922 loops=1)
               -> Table scan on mr  (cost=499.45 rows=4922) (actual time=0.063..1.686 rows=4922 loops=1)
            -> Filter: ((m.releaseYear <= 2010) and (m.releaseYear >= 2005) and (m.releaseYear is not null))  (cost=0.25 rows=0) (actual time=0.002..0.002 rows=0 loops=4922)
               -> Single-row index lookup on m using PRIMARY (name=mr.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
         -> Index lookup on maxes using <auto_key0> (yr=m.releaseYear, maxr=mr.`value`)  (actual time=0.000..0.000 rows=0 loops=448)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=18.747..18.754 rows=31 loops=1)
               -> Table scan on <temporary>  (actual time=0.001..0.003 rows=31 loops=1)
                  -> Aggregate using temporary table  (actual time=18.439..18.443 rows=31 loops=1)
                     -> Nested loop inner join  (cost=2222.15 rows=1033) (actual time=0.193..18.331 rows=144 loops=1)
                        -> Table scan on mr1  (cost=499.45 rows=4922) (actual time=0.051..1.574 rows=4922 loops=1)
                        -> Filter: ((m1.genre like '%Horror') or (m1.genre like '%horror'))  (cost=0.25 rows=0) (actual time=0.003..0.003 rows=0 loops=4922)
                           -> Single-row index lookup on m1 using PRIMARY (name=mr1.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
```

b. 3 Indexing Designs

i. Indexing on only MovieRating values

1. CREATE INDEX idx_value ON MovieRating(value); - actual time = 31.596 - 31.599
2. 31.596 to first row
3. 31.599 milliseconds to 15 rows

```
Limit: 15 row(s)  (actual time=31.596..31.599 rows=15 loops=1)
-> Sort: m.`name`, limit input to 15 row(s) per chunk  (actual time=31.595..31.597 rows=15 loops=1)
   -> Stream results  (cost=3591.75 rows=0) (actual time=26.838..31.561 rows=27 loops=1)
      -> Nested loop inner join  (cost=3591.75 rows=0) (actual time=26.832..31.542 rows=27 loops=1)
         -> Nested loop inner join  (cost=2222.15 rows=547) (actual time=0.105..14.224 rows=448 loops=1)
            -> Filter: (mr.`value` is not null)  (cost=499.45 rows=4922) (actual time=0.053..1.793 rows=4922 loops=1)
               -> Index scan on mr using idx_value  (cost=499.45 rows=4922) (actual time=0.051..1.386 rows=4922 loops=1)
            -> Filter: ((m.releaseYear <= 2010) and (m.releaseYear >= 2005) and (m.releaseYear is not null))  (cost=0.25 rows=0) (actual time=0.002..0.002 rows=0 loops=4922)
               -> Single-row index lookup on m using PRIMARY (name=mr.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
         -> Index lookup on maxes using <auto_key0> (yr=m.releaseYear, maxr=mr.`value`)  (actual time=0.000..0.001 rows=0 loops=448)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=17.228..17.234 rows=31 loops=1)
               -> Table scan on <temporary>  (actual time=0.001..0.003 rows=31 loops=1)
                  -> Aggregate using temporary table  (actual time=16.923..16.927 rows=31 loops=1)
                     -> Nested loop inner join  (cost=2222.15 rows=1033) (actual time=0.400..16.825 rows=144 loops=1)
                        -> Index scan on mr1 using idx_value  (cost=499.45 rows=4922) (actual time=0.028..1.400 rows=4922 loops=1)
                        -> Filter: ((m1.genre like '%Horror') or (m1.genre like '%horror'))  (cost=0.25 rows=0) (actual time=0.003..0.003 rows=0 loops=4922)
                           -> Single-row index lookup on m1 using PRIMARY (name=mr1.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
```

ii. Indexing on Movies releaseYear

1. CREATE INDEX idx_year ON Movies(releaseYear); - 19.298..27.588
2. 19.298 to first row
3. 27.588 milliseconds to 15 rows

```
Limit: 15 row(s)  (cost=5008.46 rows=0) (actual time=19.298..27.588 rows=15 loops=1)
-> Nested loop inner join  (cost=5008.46 rows=0) (actual time=19.283..27.571 rows=15 loops=1)
   -> Nested loop inner join  (cost=454.91 rows=1) (actual time=0.092..8.623 rows=277 loops=1)
      -> Filter: ((m.releaseYear <= 2010) and (m.releaseYear >= 2005) and (m.releaseYear is not null))  (cost=0.33 rows=1) (actual time=0.061..4.215 rows=938 loops=1)
         -> Index scan on m using PRIMARY  (cost=0.33 rows=10) (actual time=0.029..3.421 rows=9979 loops=1)
      -> Filter: (mr.`value` is not null)  (cost=0.29 rows=1) (actual time=0.004..0.005 rows=0 loops=938)
         -> Index lookup on mr using name (name=m.`name`)  (cost=0.29 rows=1) (actual time=0.004..0.004 rows=0 loops=938)
   -> Index lookup on maxes using <auto_key0> (yr=m.releaseYear, maxr=mr.`value`)  (actual time=0.000..0.001 rows=0 loops=277)
      -> Materialize  (cost=0.00..0.00 rows=0) (actual time=18.882..18.888 rows=31 loops=1)
         -> Table scan on <temporary>  (actual time=0.001..0.003 rows=31 loops=1)
            -> Aggregate using temporary table  (actual time=18.673..18.677 rows=31 loops=1)
               -> Nested loop inner join  (cost=2222.15 rows=1033) (actual time=0.142..18.535 rows=144 loops=1)
                  -> Table scan on mr1  (cost=499.45 rows=4922) (actual time=0.037..1.672 rows=4922 loops=1)
                  -> Filter: ((m1.genre like '%Horror') or (m1.genre like '%horror'))  (cost=0.25 rows=0) (actual time=0.003..0.003 rows=0 loops=4922)
                     -> Single-row index lookup on m1 using PRIMARY (name=mr1.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
```

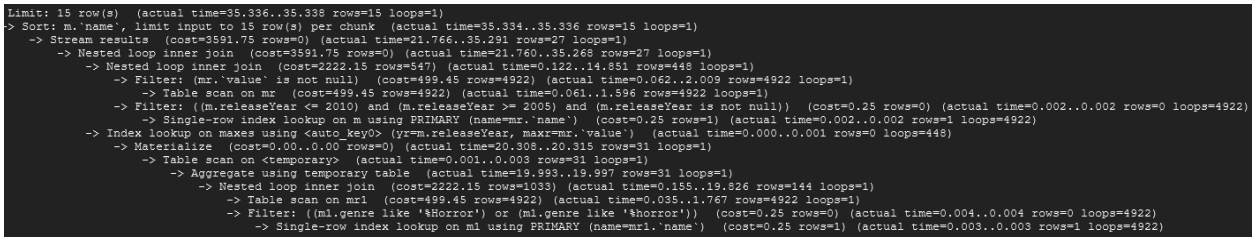
iii. Indexing on Movies name

1. CREATE INDEX idx_name ON Movies(name); - 32.949..32.951
2. 32.949 to first row
3. 32.951 milliseconds to 15 rows

```
Limit: 15 row(s)  (actual time=32.949..32.951 rows=15 loops=1)
-> Sort: m.`name`, limit input to 15 row(s) per chunk  (actual time=32.948..32.950 rows=15 loops=1)
   -> Stream results  (cost=3591.75 rows=0) (actual time=18.409..32.889 rows=27 loops=1)
      -> Nested loop inner join  (cost=3591.75 rows=0) (actual time=18.405..32.863 rows=27 loops=1)
         -> Nested loop inner join  (cost=2222.15 rows=547) (actual time=0.120..15.376 rows=448 loops=1)
            -> Filter: (mr.`value` is not null)  (cost=499.45 rows=4922) (actual time=0.045..2.248 rows=4922 loops=1)
               -> Table scan on mr  (cost=499.45 rows=4922) (actual time=0.043..1.804 rows=4922 loops=1)
            -> Filter: ((m.releaseYear <= 2010) and (m.releaseYear >= 2005) and (m.releaseYear is not null))  (cost=0.25 rows=0) (actual time=0.003..0.003 rows=0 loops=4922)
               -> Single-row index lookup on m using PRIMARY (name=mr.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
         -> Index lookup on maxes using <auto_key0> (yr=m.releaseYear, maxr=mr.`value`)  (actual time=0.001..0.001 rows=0 loops=448)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=17.369..17.377 rows=31 loops=1)
               -> Table scan on <temporary>  (actual time=0.001..0.003 rows=31 loops=1)
                  -> Aggregate using temporary table  (actual time=16.995..16.999 rows=31 loops=1)
                     -> Nested loop inner join  (cost=2222.15 rows=1033) (actual time=0.130..16.897 rows=144 loops=1)
                        -> Table scan on mr1  (cost=499.45 rows=4922) (actual time=0.041..1.498 rows=4922 loops=1)
                        -> Filter: ((m1.genre like '%Horror') or (m1.genre like '%horror'))  (cost=0.25 rows=0) (actual time=0.003..0.003 rows=0 loops=4922)
                           -> Single-row index lookup on m1 using PRIMARY (name=mr1.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
```

iv. Indexing on Movies genre

1. CREATE INDEX idx_genre ON Movies(genre); - 35.336..35.338
2. 35.336 to first row
3. 35.338 milliseconds to 15 rows



```
Limit: 15 row(s)  (actual time=35.336..35.338 rows=15 loops=1)
-> Sort: m.`name`, limit input to 15 row(s) per chunk  (actual time=35.334..35.336 rows=15 loops=1)
   -> Stream results  (cost=3591.75 rows=0) (actual time=21.766..35.291 rows=27 loops=1)
      -> Nested loop inner join  (cost=3591.75 rows=0) (actual time=21.760..35.268 rows=27 loops=1)
         -> Nested loop inner join  (cost=2222.15 rows=547) (actual time=0.122..14.851 rows=448 loops=1)
            -> Filter: (mr.`value` is not null)  (cost=499.45 rows=4922) (actual time=0.062..2.009 rows=4922 loops=1)
               -> Table scan on mr  (cost=499.45 rows=4922) (actual time=0.061..1.596 rows=4922 loops=1)
            -> Filter: ((m.releaseYear <= 2010) and (m.releaseYear >= 2005) and (m.releaseYear is not null))  (cost=0.25 rows=0) (actual time=0.002..0.002 rows=0 loops=4922)
               -> Single-row index lookup on m using PRIMARY (name=mr.`name`)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=4922)
         -> Index lookup on maxes using <auto_key0> (yr=m.releaseYear, maxr=mr.`value`)  (actual time=0.000..0.001 rows=0 loops=448)
            -> Materialize  (cost=0.00..0.00 rows=0) (actual time=20.308..20.315 rows=31 loops=1)
               -> Table scan on <temporary>  (actual time=0.001..0.003 rows=31 loops=1)
                  -> Aggregate using temporary table  (actual time=19.993..19.997 rows=31 loops=1)
                     -> Nested loop inner join  (cost=2222.15 rows=1033) (actual time=0.155..19.826 rows=144 loops=1)
                        -> Table scan on mr1  (cost=499.45 rows=4922) (actual time=0.035..1.767 rows=4922 loops=1)
                        -> Filter: ((m1.genre like '%Horror') or (m1.genre like '%horror'))  (cost=0.25 rows=0) (actual time=0.004..0.004 rows=0 loops=4922)
                           -> Single-row index lookup on m1 using PRIMARY (name=mr1.`name`)  (cost=0.25 rows=1) (actual time=0.003..0.003 rows=1 loops=4922)
```

v. Summary of indexing designs

1. The default index resulted in 33.429 ms to get the first row and 33.432 ms to get the first 15 rows. Out of the 4 indexing designs, the index created on Movies.releaseYear seemed to produce the best results with a time of 19.298 ms to get the first row and 27.588 ms to get the first 15 rows. Strangely, certain indexes such as the one on movie name and genre produced worse results. This is likely due to the lack of correlation between these two values and the results. The worst performing index was created on Movies genre with a time of 35.336 ms to get the first row and 35.338 to get the first 15 rows, which was significantly worse that the default index design.Since the index created on Movies.releaseYear achieved the best results, this is the index that we chose for this query.

2. Advanced Query 2: Find movies, users, and values of ratings from users who live in a country where Netflix is available in the order of movie names.

   Advanced Query:

```
SELECT name AS movieName, firstName, lastName, value
    FROM Users
    NATURAL JOIN MovieRating
    WHERE country
    IN (
        SELECT country
        FROM Platform
        WHERE platform = 'Netflix'
    )
ORDER BY movieName
LIMIT 15;
```

   Output:

```
mysql> SELECT name AS movieName, firstName, lastName, value
    -> FROM Users NATURAL JOIN MovieRating
    -> WHERE value >= 7 AND country IN
    ->    (SELECT country
    -> FROM Platform
    -> WHERE platform = 'Netflix')
    -> ORDER BY movieName
    -> LIMIT 15;
+------------------------------------------------------------+-----------+----------+-------+
| movieName                                                  | firstName | lastName | value |
+------------------------------------------------------------+-----------+----------+-------+
| 10 Endrathukulla                                           | Maya      | Mayor    |    10 |
| 100% Halal                                                 | Leonel    | Messi    |     7 |
| 15-Minute Full Body Challenge II 6.0 Workout (with weights) | Kathy     | Gray     |     8 |
| 15-Minute HIIT 9.0 (tabata workout with weights)           | Maya      | Mayor    |     9 |
| 15-Minute Lower Body Chisel 2.0 Workout                    | Leonel    | Messi    |    10 |
| 21                                                         | Leonel    | Messi    |     7 |
| 27, el club de los malditos                                | Kathy     | Gray     |    10 |
| 28 Days Later                                              | Jacob     | Gray     |     7 |
| 3 Heroines                                                 | Leonel    | Messi    |     8 |
| 3 Pints and a Rabbi                                        | Maya      | Mayor    |     8 |
| 37 Seconds                                                 | Leonel    | Messi    |     7 |
| 6-5=2                                                      | Kathy     | Gray     |     9 |
| 7 Yards: The Chris Norton Story                            | Maya      | Mayor    |    10 |
| 911 Nightmare                                              | Jacob     | Gray     |     7 |
| A 30-Minute Journey Around the World                       | Maya      | Mayor    |     7 |
+------------------------------------------------------------+-----------+----------+-------+
15 rows in set (0.00 sec)
```

   a. EXPLAIN ANALYZE – actual time = 4.095 - 4.101
      i. 4.095 milliseconds to first row
      ii. 4.101 milliseconds to 15 rows

```
| -> Limit: 15 row(s)  (actual time=4.095..4.101 rows=15 loops=1)
    -> Sort: MovieRating.`name`, limit input to 15 row(s) per chunk  (actual time=4.094..4.099 rows=15loops=1)
        -> Stream results  (cost=1405.10 rows=13388) (actual time=0.658..3.331 rows=1741 loops=1)
            -> Nested loop inner join  (cost=1405.10 rows=13388) (actual time=0.649..2.093 rows=1741 loops=1)
                -> Remove duplicate Users rows using temporary table (weedout)  (cost=65.82 rows=38) (actual time=0.568..0.616 rows=5 loops=1)
                    -> Filter: (Users.country = Platform.country)  (cost=65.82 rows=38) (actual time=0.560..0.598 rows=5 loops=1)
                        -> Inner hash join (<hash>(Users.country)=<hash>(Platform.country))  (cost=65.82 rows=38) (actual time=0.559..0.590 rows=5 loops=1)
                            -> Table scan on Users  (cost=0.43 rows=14) (actual time=0.026..0.041 rows=14 loops=1)
                            -> Hash
                                -> Filter: (Platform.platform = 'Netflix')  (cost=27.45 rows=27) (actual time=0.108..0.329 rows=189 loops=1)
                                    -> Table scan on Platform  (cost=27.45 rows=272) (actual time=0.100..0.226 rows=272 loops=1)
                -> Index lookup on MovieRating using PRIMARY (id=Users.id)  (cost=25.47 rows=352) (actual time=0.052..0.260 rows=348 loops=5)
 |
```

b. 3 Indexing designs

i. Indexing on value for MovieRating

1. CREATE INDEX idx_value ON MovieRating(value); – actual time = 2.187 - 2.190
2. 2.187 milliseconds to first row
3. 2.190 milliseconds to 15 rows

```
| -> Limit: 15 row(s)  (actual time=2.187..2.190 rows=15 loops=1)
    -> Sort: MovieRating.`name`, limit input to 15 row(s) per chunk  (actual time=2.185..2.188 rows=15 loops=1)
        -> Stream results  (cost=1405.10 rows=13388) (actual time=0.402..1.785 rows=1741 loops=1)
            -> Nested loop inner join  (cost=1405.10 rows=13388) (actual time=0.394..1.118 rows=1741 loops=1)
                -> Remove duplicate Users rows using temporary table (weedout)  (cost=65.82 rows=38) (actual time=0.343..0.359 rows=5 loops=1)
                    -> Filter: (Users.country = Platform.country)  (cost=65.82 rows=38) (actual time=0.337..0.350 rows=5 loops=1)
                        -> Inner hash join (<hash>(Users.country)=<hash>(Platform.country))  (cost=65.82 rows=38) (actual time=0.336..0.347 rows=5 loops=1)
                            -> Table scan on Users  (cost=0.43 rows=14) (actual time=0.015..0.021 rows=14 loops=1)
                            -> Hash
                                -> Filter: (Platform.platform = 'Netflix')  (cost=27.45 rows=27) (actual time=0.095..0.212 rows=189 loops=1)
                                    -> Table scan on Platform  (cost=27.45 rows=272) (actual time=0.089..0.154 rows=272 loops=1)
                -> Index lookup on MovieRating using PRIMARY (id=Users.id)  (cost=25.47 rows=352) (actual time=0.027..0.129 rows=348 loops=5)
 |
```

ii. Indexing on platform for Platform

1. CREATE INDEX idx_platform ON Platform(platform); – actual time = 2.250 - 2.253
2. 2.250 milliseconds to first row
3. 2.254 milliseconds to 15 rows

```
| -> Limit: 15 row(s)  (actual time=2.250..2.253 rows=15 loops=1)
    -> Sort: MovieRating.`name`, limit input to 15 row(s) per chunk  (actual time=2.249..2.251 rows=15loops=1)
        -> Stream results  (cost=9587.83 rows=93026) (actual time=0.543..1.861 rows=1741 loops=1)
            -> Nested loop inner join  (cost=9587.83 rows=93026) (actual time=0.539..1.198 rows=1741 loops=1)
                -> Remove duplicate Users rows using temporary table (weedout)  (cost=284.76 rows=265)(actual time=0.498..0.508 rows=5 loops=1)
                    -> Filter: (Users.country = Platform.country)  (cost=284.76 rows=265) (actual time=0.493..0.501 rows=5 loops=1)
                        -> Inner hash join (<hash>(Users.country)=<hash>(Platform.country))  (cost=284.76 rows=265) (actual time=0.491..0.498 rows=5 loops=1)
                            -> Table scan on Users  (cost=0.65 rows=14) (actual time=0.014..0.017 rows=14 loops=1)
                            -> Hash
                                -> Index lookup on Platform using idx_platform (platform='Netflix')  (cost=19.65 rows=189) (actual time=0.208..0.375 rows=189 loops=1)
                -> Index lookup on MovieRating using PRIMARY (id=Users.id)  (cost=25.47 rows=352) (actual time=0.024..0.116 rows=348 loops=5)
 |
```

iii. Indexing on firstName for Users

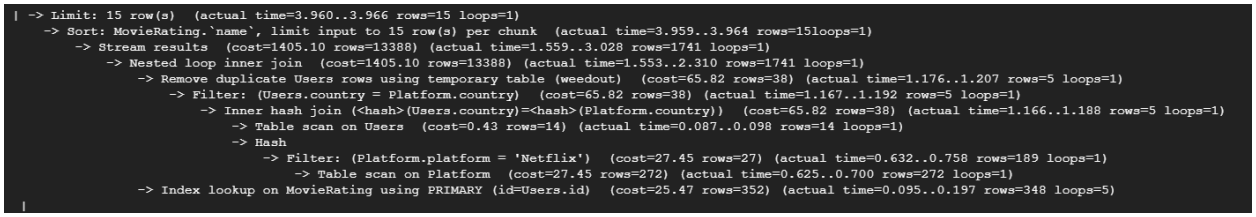
1. CREATE INDEX idx_firstName ON Users(firstName); – actual time = 2.072 - 2.075
2. 2.072 milliseconds to first row
3. 2.075 milliseconds to 15 rows

```
| -> Limit: 15 row(s)  (actual time=2.072..2.075 rows=15 loops=1)
    -> Sort: MovieRating.`name`, limit input to 15 row(s) per chunk  (actual time=2.071..2.073 rows=15 loops=1)
        -> Stream results  (cost=1405.10 rows=13388) (actual time=0.375..1.698 rows=1741 loops=1)
            -> Nested loop inner join  (cost=1405.10 rows=13388) (actual time=0.370..1.046 rows=1741 loops=1)
                -> Remove duplicate Users rows using temporary table (weedout)  (cost=65.82 rows=38) (actual time=0.323..0.334 rows=5 loops=1)
                    -> Filter: (Users.country = Platform.country)  (cost=65.82 rows=38) (actual time=0.318..0.326 rows=5 loops=1)
                        -> Inner hash join (<hash>(Users.country)=<hash>(Platform.country))  (cost=65.82 rows=38) (actual time=0.317..0.324 rows=5 loops=1)
                            -> Table scan on Users  (cost=0.43 rows=14) (actual time=0.014..0.017 rows=14 loops=1)
                            -> Hash
                                -> Filter: (Platform.platform = 'Netflix')  (cost=27.45 rows=27) (actual time=0.044..0.181 rows=189 loops=1)
                                    -> Table scan on Platform  (cost=27.45 rows=272) (actual time=0.039..0.104 rows=272 loops=1)
                -> Index lookup on MovieRating using PRIMARY (id=Users.id)  (cost=25.47 rows=352) (actual time=0.028..0.121 rows=348 loops=5)
 |
```

iv. Indexing on date for MovieRating

1. CREATE INDEX idx_date ON MovieRating(date); – actual time = 3.960 - 3.966
2. 3.960 milliseconds to first row

3. 3.966 milliseconds to 15 rows



v. Summary of indexing designs

1. The default index resulted in 4.095 ms to get the first row and 4.101 ms to get the first 15 rows. Out of the 4 indexing designs, the index created on Users.firstName seemed to produce the best results with a time of 2.072 ms to get the first row and 2.075 ms to get the first 15 rows. Additionally, all the indexing designs seem to produce better results than the default index. The worst performing index was created on MovieRating.date with a time of 3.960 ms to get the first row and 3.966 to get the first 15 rows, which was still better than the default index created.Since the index created on Users.firstName achieved the best results, this is the index that we chose for this query.