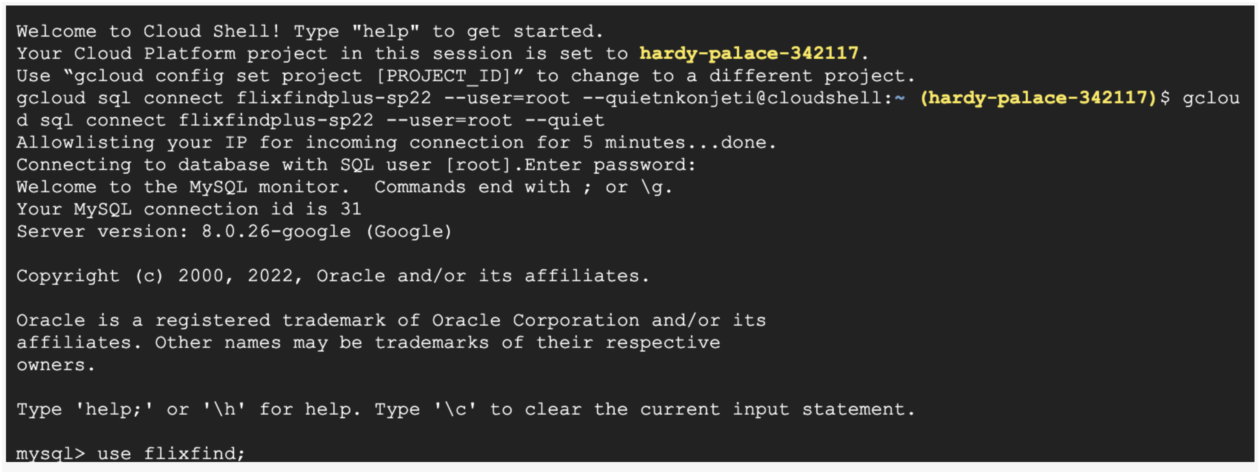
PT1 Stage 3

Database Implementation:

CONNECTION:



```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to hardy-palace-342117.
Use “gcloud config set project [PROJECT_ID]” to change to a different project.
gcloud sql connect flixfindplus-sp22 --user=root --quietnkonjeti@cloudshell:~ (hardy-palace-342117)$ gclou
d sql connect flixfindplus-sp22 --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 31
Server version: 8.0.26-google (Google)

Copyright (c) 2000, 2022, 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> use flixfind;
```

TABLE COMMANDS:

1. *Movie:*

   CREATE TABLE Movie(

   MovieId INTEGER,

   Title VARCHAR(50),

   Year INTEGER,

   Review VARCHAR(50),

   Score INTEGER,

   PRIMARY KEY (MovieId)

   );

2. *StreamingPlatform*

   CREATE TABLE StreamingPlatform(

   PlatformName VARCHAR(20),

```
        PRIMARY KEY (PlatformName)
    );
```

3. *User*

```
CREATE TABLE User(
        UserId INTEGER,
        Username VARCHAR (30),
        Age INTEGER,
        Password VARCHAR(40),
        PRIMARY KEY (UserId)
);
```

4. *MovieList*

```
CREATE TABLE MovieList(
        ListId INTEGER,
        PRIMARY KEY (ListId)
);
```

5. *MovieListMovieAssociation*

```
CREATE TABLE MovieListMovieAssociation(
        MovieId INTEGER,
        ListId INTEGER,
        FOREIGN KEY (MovieId) REFERENCES Movie(MovieId),
        FOREIGN KEY (ListId) REFERENCES MovieList(ListId)
);
```

6. *BlackList*

```
CREATE TABLE BlackList(
        ListId INTEGER,
        UserId INTEGER,
        AvgRating REAL,
                FOREIGN KEY (ListId) REFERENCES MovieList(ListId),
                FOREIGN KEY (UserId) REFERENCES User(UserId),
                PRIMARY KEY (ListId, UserId)
        );
```

7. *MoviePlatformAssociation*

```
CREATE TABLE MoviePlatformAssociation(
        MovieId INTEGER,
        PlatformName VARCHAR(20),
        FOREIGN KEY (MovieId) REFERENCES Movie(MovieId),
        FOREIGN KEY (PlatformName) REFERENCES
        StreamingPlatform(PlatformName)
);
```

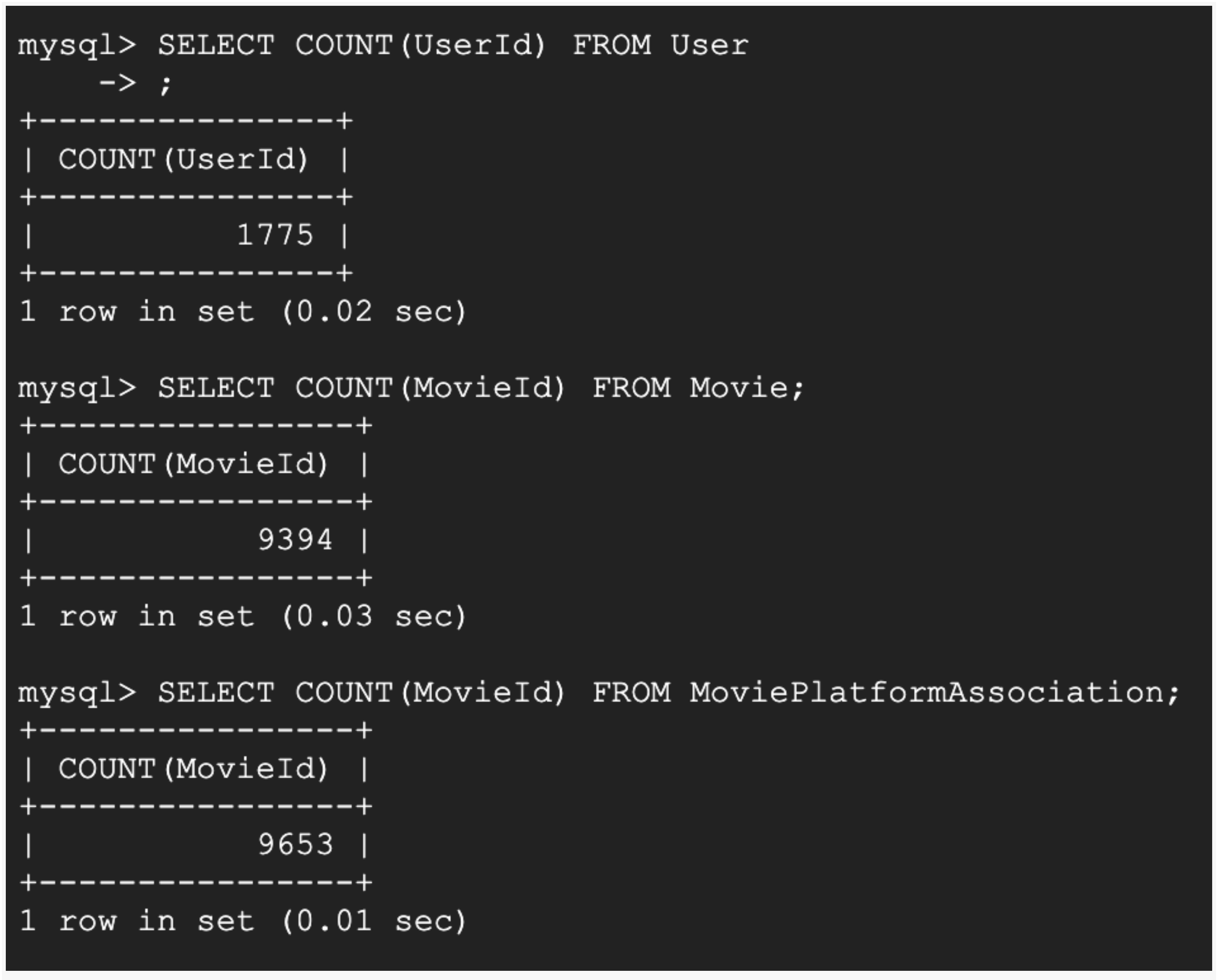
8. *WatchList*

```
CREATE TABLE WatchList(
        ListId INTEGER,
        UserId INTEGER,
        TotalRuntime VARCHAR(20),
        FOREIGN KEY (ListId) REFERENCES MovieList(ListId),
        FOREIGN KEY (UserId) REFERENCES User(UserId),
        PRIMARY KEY (ListId, UserId)
);
```

9. *Rating*

```
CREATE TABLE Rating(
        UserId INTEGER,
        MovieId INTEGER,
        DateTime VARCHAR(20),
        Score INTEGER,
        FOREIGN KEY (UserId) REFERENCES User(UserId),
        FOREIGN KEY (MovieId) REFERENCES Movie(MovieId),
                    PRIMARY KEY (UserId, MovieId)
          );
```

Number of rows in User, Movie, and MoviePlatformAssociation:



```
mysql> SELECT COUNT(UserId) FROM User
    -> ;
+---------------+
| COUNT(UserId) |
+---------------+
|          1775 |
+---------------+
1 row in set (0.02 sec)

mysql> SELECT COUNT(MovieId) FROM Movie;
+----------------+
| COUNT(MovieId) |
+----------------+
|           9394 |
+----------------+
1 row in set (0.03 sec)

mysql> SELECT COUNT(MovieId) FROM MoviePlatformAssociation;
+----------------+
| COUNT(MovieId) |
+----------------+
|           9653 |
+----------------+
1 row in set (0.01 sec)
```

QUERIES:

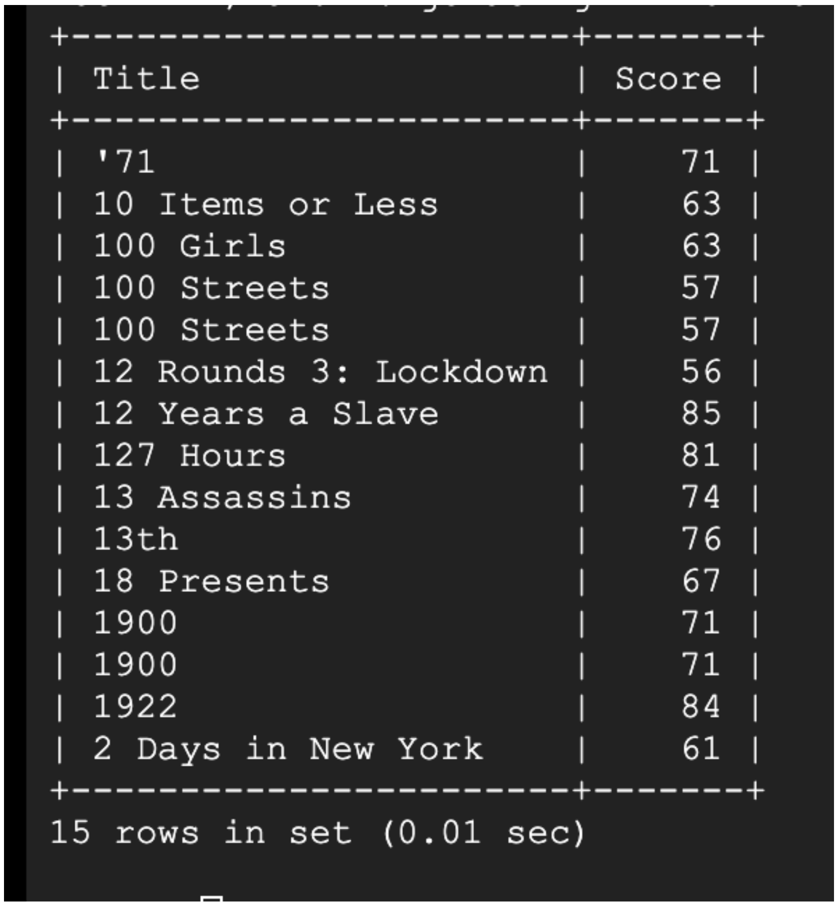
1. **Find the movies on Netflix that have a score higher than the average score on Netflix and is rated 18+:**

SELECT m.Title, m.Score

FROM Movie m JOIN MoviePlatformAssociation a USING (MovieId)

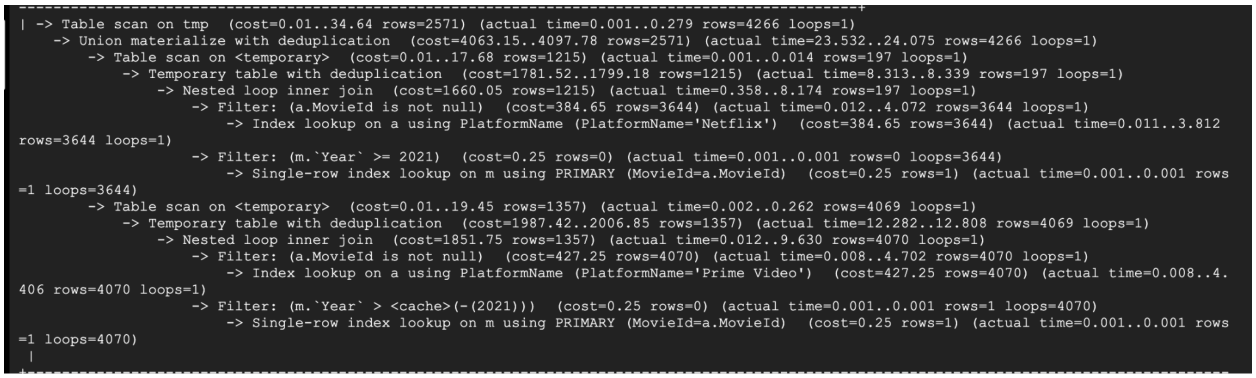
WHERE m.Score > (Select AVG(m1.Score) FROM Movie m1 JOIN MoviePlatformAssociation a1 USING (MovieId) WHERE a1.PlatformName = ‘Netflix’) and m.AgeRating = ‘18+’

ORDER BY m.Title



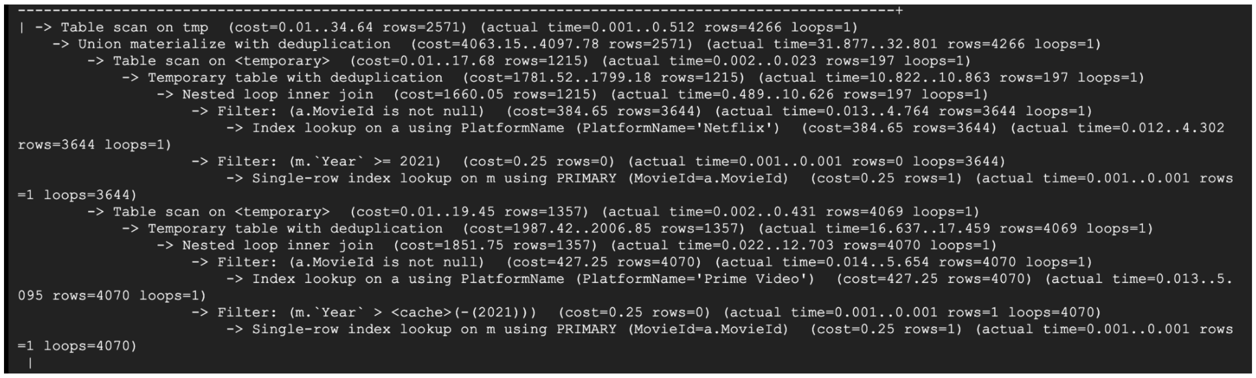
```
+------------------------+-------+
| Title                  | Score |
+------------------------+-------+
| '71                    |    71 |
| 10 Items or Less       |    63 |
| 100 Girls              |    63 |
| 100 Streets            |    57 |
| 100 Streets            |    57 |
| 12 Rounds 3: Lockdown  |    56 |
| 12 Years a Slave       |    85 |
| 127 Hours              |    81 |
| 13 Assassins           |    74 |
| 13th                   |    76 |
| 18 Presents            |    67 |
| 1900                   |    71 |
| 1900                   |    71 |
| 1922                   |    84 |
| 2 Days in New York     |    61 |
+------------------------+-------+
15 rows in set (0.01 sec)
```

Without index



```
-----------------------------------------------------------------------------------------------+
| -> Table scan on tmp  (cost=0.01..34.64 rows=2571) (actual time=0.001..0.279 rows=4266 loops=1)
    -> Union materialize with deduplication  (cost=4063.15..4097.78 rows=2571) (actual time=23.532..24.075 rows=4266 loops=1)
        -> Table scan on <temporary>  (cost=0.01..17.68 rows=1215) (actual time=0.001..0.014 rows=197 loops=1)
            -> Temporary table with deduplication  (cost=1781.52..1799.18 rows=1215) (actual time=8.313..8.339 rows=197 loops=1)
                -> Nested loop inner join  (cost=1660.05 rows=1215) (actual time=0.358..8.174 rows=197 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=384.65 rows=3644) (actual time=0.012..4.072 rows=3644 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Netflix')  (cost=384.65 rows=3644) (actual time=0.011..3.812
rows=3644 loops=1)
                    -> Filter: (m.`Year` >= 2021)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=3644)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows
=1 loops=3644)
        -> Table scan on <temporary>  (cost=0.01..19.45 rows=1357) (actual time=0.002..0.262 rows=4069 loops=1)
            -> Temporary table with deduplication  (cost=1987.42..2006.85 rows=1357) (actual time=12.282..12.808 rows=4069 loops=1)
                -> Nested loop inner join  (cost=1851.75 rows=1357) (actual time=0.012..9.630 rows=4070 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=427.25 rows=4070) (actual time=0.008..4.702 rows=4070 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Prime Video')  (cost=427.25 rows=4070) (actual time=0.008..4.
406 rows=4070 loops=1)
                    -> Filter: (m.`Year` > <cache>(-(2021)))  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=4070)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows
=1 loops=4070)
 |
+-----------------------------------------------------------------------------------------------------------------------------------
```

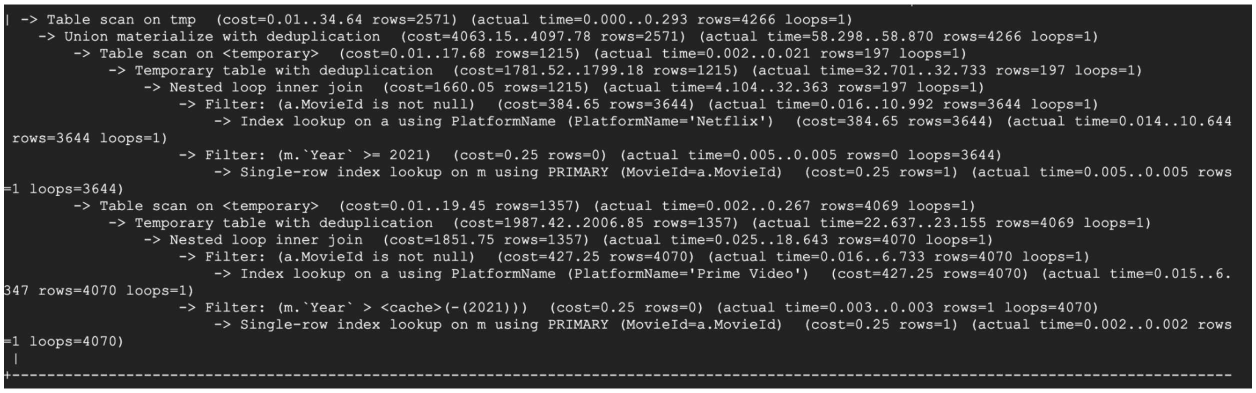
Create index on title



```
-----------------------------------------------------------------------------------------------------+
| -> Table scan on tmp  (cost=0.01..34.64 rows=2571) (actual time=0.001..0.512 rows=4266 loops=1)
    -> Union materialize with deduplication  (cost=4063.15..4097.78 rows=2571) (actual time=31.877..32.801 rows=4266 loops=1)
        -> Table scan on <temporary>  (cost=0.01..17.68 rows=1215) (actual time=0.002..0.023 rows=197 loops=1)
            -> Temporary table with deduplication  (cost=1781.52..1799.18 rows=1215) (actual time=10.822..10.863 rows=197 loops=1)
                -> Nested loop inner join  (cost=1660.05 rows=1215) (actual time=0.489..10.626 rows=197 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=384.65 rows=3644) (actual time=0.013..4.764 rows=3644 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Netflix')  (cost=384.65 rows=3644) (actual time=0.012..4.302
rows=3644 loops=1)
                    -> Filter: (m.`Year` >= 2021)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=3644)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows
=1 loops=3644)
        -> Table scan on <temporary>  (cost=0.01..19.45 rows=1357) (actual time=0.002..0.431 rows=4069 loops=1)
            -> Temporary table with deduplication  (cost=1987.42..2006.85 rows=1357) (actual time=16.637..17.459 rows=4069 loops=1)
                -> Nested loop inner join  (cost=1851.75 rows=1357) (actual time=0.022..12.703 rows=4070 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=427.25 rows=4070) (actual time=0.014..5.654 rows=4070 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Prime Video')  (cost=427.25 rows=4070) (actual time=0.013..5.
095 rows=4070 loops=1)
                    -> Filter: (m.`Year` > <cache>(-(2021)))  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=4070)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows
=1 loops=4070)
 |
```

We created an index on the movie title because it is used in the ordering of our query output. An indexing structure on the movie title would seem to work better because we are selecting movie titles for each movie and ordering by them. However, it seems that the cost of creating and handling a b-tree index does not outweigh the improvement of indexing it is offering. This may be due to the select and order by being a one time loop on a very small subset of data. While the index on the title does create a quick ordering of data on the disk, it is not worth the cost of maintaining it as it is not being used enough in this query.

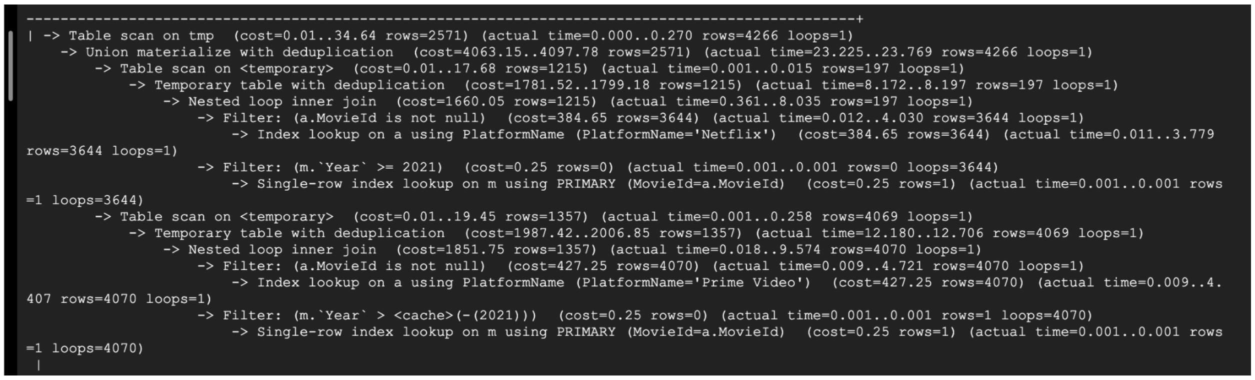
Create index on movie score



```
| -> Table scan on tmp  (cost=0.01..34.64 rows=2571) (actual time=0.000..0.293 rows=4266 loops=1)
    -> Union materialize with deduplication  (cost=4063.15..4097.78 rows=2571) (actual time=58.298..58.870 rows=4266 loops=1)
        -> Table scan on <temporary>  (cost=0.01..17.68 rows=1215) (actual time=0.002..0.021 rows=197 loops=1)
            -> Temporary table with deduplication  (cost=1781.52..1799.18 rows=1215) (actual time=32.701..32.733 rows=197 loops=1)
                -> Nested loop inner join  (cost=1660.05 rows=1215) (actual time=4.104..32.363 rows=197 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=384.65 rows=3644) (actual time=0.016..10.992 rows=3644 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Netflix')  (cost=384.65 rows=3644) (actual time=0.014..10.644
 rows=3644 loops=1)
                    -> Filter: (m.`Year` >= 2021)  (cost=0.25 rows=0) (actual time=0.005..0.005 rows=0 loops=3644)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.005..0.005 rows
=1 loops=3644)
        -> Table scan on <temporary>  (cost=0.01..19.45 rows=1357) (actual time=0.002..0.267 rows=4069 loops=1)
            -> Temporary table with deduplication  (cost=1987.42..2006.85 rows=1357) (actual time=22.637..23.155 rows=4069 loops=1)
                -> Nested loop inner join  (cost=1851.75 rows=1357) (actual time=0.025..18.643 rows=4070 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=427.25 rows=4070) (actual time=0.016..6.733 rows=4070 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Prime Video')  (cost=427.25 rows=4070) (actual time=0.015..6.
347 rows=4070 loops=1)
                    -> Filter: (m.`Year` > <cache>(-(2021)))  (cost=0.25 rows=0) (actual time=0.003..0.003 rows=1 loops=4070)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows
=1 loops=4070)
 |
+-------------------------------------------------------------------------------------------------------------------------------------
```

We created an index on the movie score because movie score is a vital part of this query. An indexing structure on the movie score works better because we create pointers to where the movie stores are stored in the database. Instead of having to sequentially traverse every movie score in the Movie table, an indexing structure on the movie score provides a quick ordering of data on the disk or tells the SQL engine exactly where to go via pointers.

Create index of age rating:



```
---------------------------------------------------------------------------------------------------+
| -> Table scan on tmp  (cost=0.01..34.64 rows=2571) (actual time=0.000..0.270 rows=4266 loops=1)
    -> Union materialize with deduplication  (cost=4063.15..4097.78 rows=2571) (actual time=23.225..23.769 rows=4266 loops=1)
        -> Table scan on <temporary>  (cost=0.01..17.68 rows=1215) (actual time=0.001..0.015 rows=197 loops=1)
            -> Temporary table with deduplication  (cost=1781.52..1799.18 rows=1215) (actual time=8.172..8.197 rows=197 loops=1)
                -> Nested loop inner join  (cost=1660.05 rows=1215) (actual time=0.361..8.035 rows=197 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=384.65 rows=3644) (actual time=0.012..4.030 rows=3644 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Netflix')  (cost=384.65 rows=3644) (actual time=0.011..3.779
rows=3644 loops=1)
                    -> Filter: (m.`Year` >= 2021)  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=0 loops=3644)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows
=1 loops=3644)
        -> Table scan on <temporary>  (cost=0.01..19.45 rows=1357) (actual time=0.001..0.258 rows=4069 loops=1)
            -> Temporary table with deduplication  (cost=1987.42..2006.85 rows=1357) (actual time=12.180..12.706 rows=4069 loops=1)
                -> Nested loop inner join  (cost=1851.75 rows=1357) (actual time=0.018..9.574 rows=4070 loops=1)
                    -> Filter: (a.MovieId is not null)  (cost=427.25 rows=4070) (actual time=0.009..4.721 rows=4070 loops=1)
                        -> Index lookup on a using PlatformName (PlatformName='Prime Video')  (cost=427.25 rows=4070) (actual time=0.009..4.
407 rows=4070 loops=1)
                    -> Filter: (m.`Year` > <cache>(-(2021)))  (cost=0.25 rows=0) (actual time=0.001..0.001 rows=1 loops=4070)
                        -> Single-row index lookup on m using PRIMARY (MovieId=a.MovieId)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows
=1 loops=4070)
 |
```

We indexed using the age rating for the movie because it was in the where clause of our query. We thought performance would increase by shortening the time to find the age rating for each movie due to pointers being created for each age rating. However, indexing on the age rating didn't improve the query because the age rating is checked for each movie in the table.

Find the movies that were released in the past 1 year from Netflix and Prime Video and are for ages 7+:

SELECT *

FROM ((SELECT DISTINCT m.Title, a.PlatformName

FROM Movie m JOIN MoviePlatformAssociation a USING (MovieId)

WHERE m.Year >= 2021 AND a.PlatformName = ‘Netflix’ AND m.AgeRating = ‘13+’)

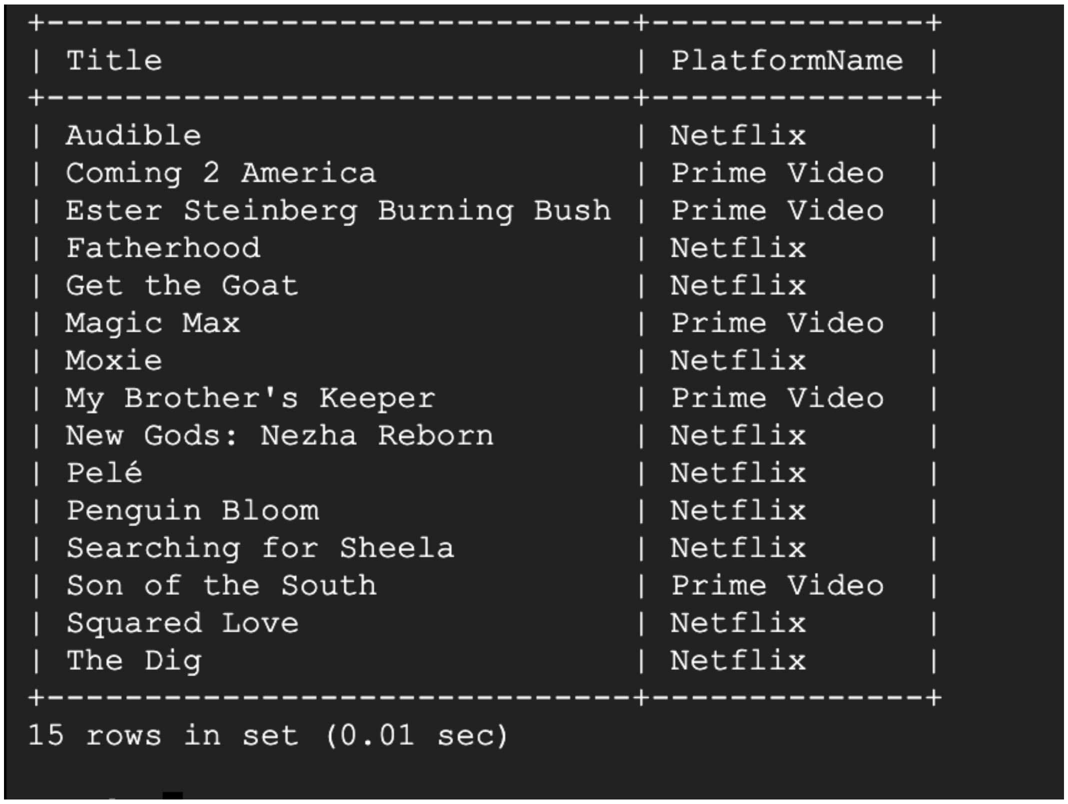
UNION

(SELECT DISTINCT m.Title, a.PlatformName

FROM Movie m JOIN MoviePlatformAssociation a USING (MovieId)

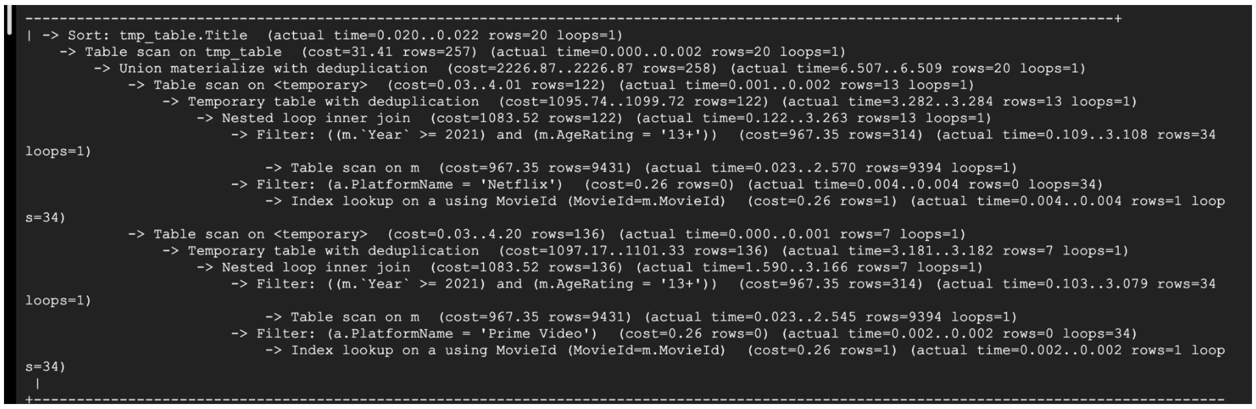
WHERE m.Year >= 2021 AND a.PlatformName = ‘Prime Video’ AND m.AgeRating = ‘13+’)) as tmp_table

ORDER BY tmp_table.Title



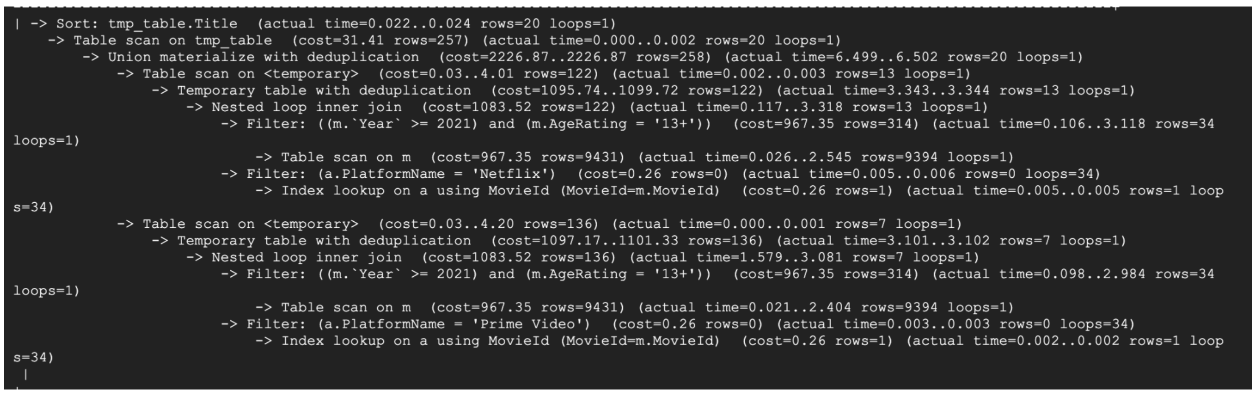
```
+-------------------------------+--------------+
| Title                         | PlatformName |
+-------------------------------+--------------+
| Audible                       | Netflix      |
| Coming 2 America              | Prime Video  |
| Ester Steinberg Burning Bush  | Prime Video  |
| Fatherhood                    | Netflix      |
| Get the Goat                  | Netflix      |
| Magic Max                     | Prime Video  |
| Moxie                         | Netflix      |
| My Brother's Keeper           | Prime Video  |
| New Gods: Nezha Reborn        | Netflix      |
| Pelé                          | Netflix      |
| Penguin Bloom                 | Netflix      |
| Searching for Sheela          | Netflix      |
| Son of the South              | Prime Video  |
| Squared Love                  | Netflix      |
| The Dig                       | Netflix      |
+-------------------------------+--------------+
15 rows in set (0.01 sec)
```

Explain analyze without index



```
--------------------------------------------------------------------------------------------------------------------------+
| -> Sort: tmp_table.Title  (actual time=0.020..0.022 rows=20 loops=1)
    -> Table scan on tmp_table  (cost=31.41 rows=257) (actual time=0.000..0.002 rows=20 loops=1)
        -> Union materialize with deduplication  (cost=2226.87..2226.87 rows=258) (actual time=6.507..6.509 rows=20 loops=1)
            -> Table scan on <temporary>  (cost=0.03..4.01 rows=122) (actual time=0.001..0.002 rows=13 loops=1)
                -> Temporary table with deduplication  (cost=1095.74..1099.72 rows=122) (actual time=3.282..3.284 rows=13 loops=1)
                    -> Nested loop inner join  (cost=1083.52 rows=122) (actual time=0.122..3.263 rows=13 loops=1)
                        -> Filter: ((m.`Year` >= 2021) and (m.AgeRating = '13+'))  (cost=967.35 rows=314) (actual time=0.109..3.108 rows=34
loops=1)
                            -> Table scan on m  (cost=967.35 rows=9431) (actual time=0.023..2.570 rows=9394 loops=1)
                        -> Filter: (a.PlatformName = 'Netflix')  (cost=0.26 rows=0) (actual time=0.004..0.004 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.26 rows=1) (actual time=0.004..0.004 rows=1 loop
s=34)
            -> Table scan on <temporary>  (cost=0.03..4.20 rows=136) (actual time=0.000..0.001 rows=7 loops=1)
                -> Temporary table with deduplication  (cost=1097.17..1101.33 rows=136) (actual time=3.181..3.182 rows=7 loops=1)
                    -> Nested loop inner join  (cost=1083.52 rows=136) (actual time=1.590..3.166 rows=7 loops=1)
                        -> Filter: ((m.`Year` >= 2021) and (m.AgeRating = '13+'))  (cost=967.35 rows=314) (actual time=0.103..3.079 rows=34
loops=1)
                            -> Table scan on m  (cost=967.35 rows=9431) (actual time=0.023..2.545 rows=9394 loops=1)
                        -> Filter: (a.PlatformName = 'Prime Video')  (cost=0.26 rows=0) (actual time=0.002..0.002 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=1 loop
s=34)
 |
+--------------------------------------------------------------------------------------------------------------------------------------
```

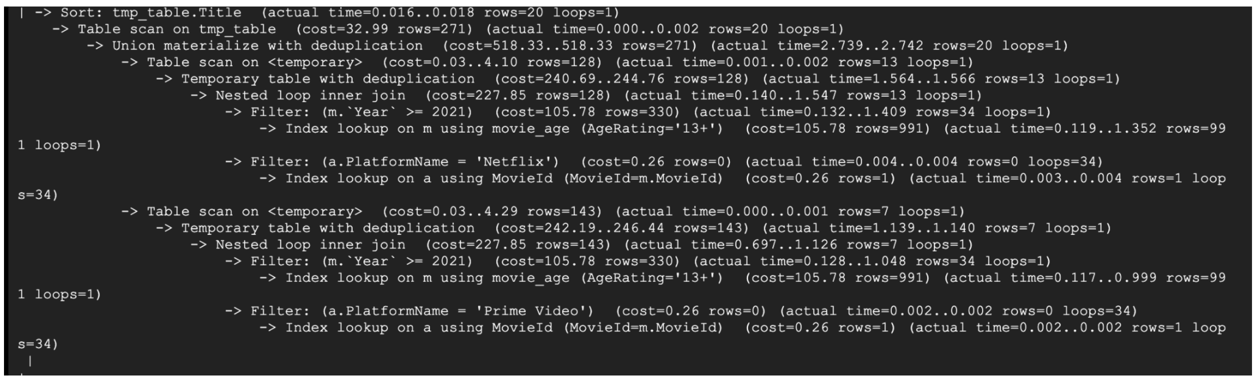
Temp_table title index



```
| -> Sort: tmp_table.Title  (actual time=0.022..0.024 rows=20 loops=1)
    -> Table scan on tmp_table  (cost=31.41 rows=257) (actual time=0.000..0.002 rows=20 loops=1)
        -> Union materialize with deduplication  (cost=2226.87..2226.87 rows=258) (actual time=6.499..6.502 rows=20 loops=1)
            -> Table scan on <temporary>  (cost=0.03..4.01 rows=122) (actual time=0.002..0.003 rows=13 loops=1)
                -> Temporary table with deduplication  (cost=1095.74..1099.72 rows=122) (actual time=3.343..3.344 rows=13 loops=1)
                    -> Nested loop inner join  (cost=1083.52 rows=122) (actual time=0.117..3.318 rows=13 loops=1)
                        -> Filter: ((m.`Year` >= 2021) and (m.AgeRating = '13+'))  (cost=967.35 rows=314) (actual time=0.106..3.118 rows=34
loops=1)
                            -> Table scan on m  (cost=967.35 rows=9431) (actual time=0.026..2.545 rows=9394 loops=1)
                        -> Filter: (a.PlatformName = 'Netflix')  (cost=0.26 rows=0) (actual time=0.005..0.006 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.26 rows=1) (actual time=0.005..0.005 rows=1 loop
s=34)
            -> Table scan on <temporary>  (cost=0.03..4.20 rows=136) (actual time=0.000..0.001 rows=7 loops=1)
                -> Temporary table with deduplication  (cost=1097.17..1101.33 rows=136) (actual time=3.101..3.102 rows=7 loops=1)
                    -> Nested loop inner join  (cost=1083.52 rows=136) (actual time=1.579..3.081 rows=7 loops=1)
                        -> Filter: ((m.`Year` >= 2021) and (m.AgeRating = '13+'))  (cost=967.35 rows=314) (actual time=0.098..2.984 rows=34
loops=1)
                            -> Table scan on m  (cost=967.35 rows=9431) (actual time=0.021..2.404 rows=9394 loops=1)
                        -> Filter: (a.PlatformName = 'Prime Video')  (cost=0.26 rows=0) (actual time=0.003..0.003 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=1 loop
s=34)
 |
```

We created an index on the temp table title because it is used in the ordering of our query output. This index does not have a significant impact on the runtime of the query as the title is only used to order the output, and is not seen anywhere else in the query. This may be due to the select and order by being a one time loop on a very small subset of data. While the index on the title does create a quick ordering of data on the disk, it is not worth the cost of maintaining it as it is not being used enough in this query.

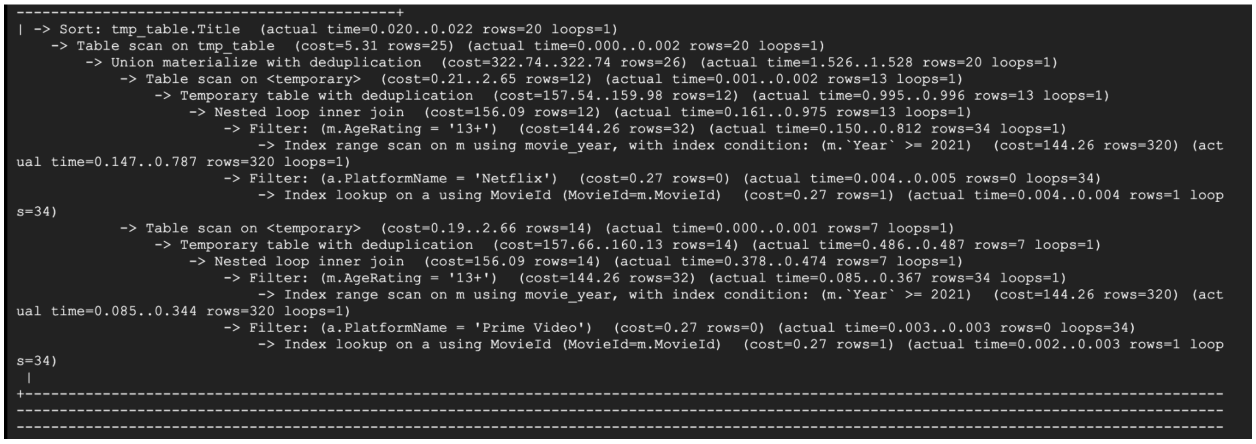
Movie age rating index



```
| -> Sort: tmp_table.Title  (actual time=0.016..0.018 rows=20 loops=1)
    -> Table scan on tmp_table  (cost=32.99 rows=271) (actual time=0.000..0.002 rows=20 loops=1)
        -> Union materialize with deduplication  (cost=518.33..518.33 rows=271) (actual time=2.739..2.742 rows=20 loops=1)
            -> Table scan on <temporary>  (cost=0.03..4.10 rows=128) (actual time=0.001..0.002 rows=13 loops=1)
                -> Temporary table with deduplication  (cost=240.69..244.76 rows=128) (actual time=1.564..1.566 rows=13 loops=1)
                    -> Nested loop inner join  (cost=227.85 rows=128) (actual time=0.140..1.547 rows=13 loops=1)
                        -> Filter: (m.`Year` >= 2021)  (cost=105.78 rows=330) (actual time=0.132..1.409 rows=34 loops=1)
                            -> Index lookup on m using movie_age (AgeRating='13+')  (cost=105.78 rows=991) (actual time=0.119..1.352 rows=99
1 loops=1)
                        -> Filter: (a.PlatformName = 'Netflix')  (cost=0.26 rows=0) (actual time=0.004..0.004 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.26 rows=1) (actual time=0.003..0.004 rows=1 loop
s=34)
            -> Table scan on <temporary>  (cost=0.03..4.29 rows=143) (actual time=0.000..0.001 rows=7 loops=1)
                -> Temporary table with deduplication  (cost=242.19..246.44 rows=143) (actual time=1.139..1.140 rows=7 loops=1)
                    -> Nested loop inner join  (cost=227.85 rows=143) (actual time=0.697..1.126 rows=7 loops=1)
                        -> Filter: (m.`Year` >= 2021)  (cost=105.78 rows=330) (actual time=0.128..1.048 rows=34 loops=1)
                            -> Index lookup on m using movie_age (AgeRating='13+')  (cost=105.78 rows=991) (actual time=0.117..0.999 rows=99
1 loops=1)
                        -> Filter: (a.PlatformName = 'Prime Video')  (cost=0.26 rows=0) (actual time=0.002..0.002 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.26 rows=1) (actual time=0.002..0.002 rows=1 loop
s=34)
 |
```

We indexed using the movie age rating for the movie because it was in the WHERE clause of our query. By indexing on the age rating, the query was slightly improved as we were only choosing movies that had a rating of ‘13+’. Indexing by the age rating creates pointers for the age ratings and the lookup time for each rating is faster.

Movie year index



```
------------------------------------------+
| -> Sort: tmp_table.Title  (actual time=0.020..0.022 rows=20 loops=1)
    -> Table scan on tmp_table  (cost=5.31 rows=25) (actual time=0.000..0.002 rows=20 loops=1)
        -> Union materialize with deduplication  (cost=322.74..322.74 rows=26) (actual time=1.526..1.528 rows=20 loops=1)
            -> Table scan on <temporary>  (cost=0.21..2.65 rows=12) (actual time=0.001..0.002 rows=13 loops=1)
                -> Temporary table with deduplication  (cost=157.54..159.98 rows=12) (actual time=0.995..0.996 rows=13 loops=1)
                    -> Nested loop inner join  (cost=156.09 rows=12) (actual time=0.161..0.975 rows=13 loops=1)
                        -> Filter: (m.AgeRating = '13+')  (cost=144.26 rows=32) (actual time=0.150..0.812 rows=34 loops=1)
                            -> Index range scan on m using movie_year, with index condition: (m.`Year` >= 2021)  (cost=144.26 rows=320) (act
ual time=0.147..0.787 rows=320 loops=1)
                        -> Filter: (a.PlatformName = 'Netflix')  (cost=0.27 rows=0) (actual time=0.004..0.005 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.27 rows=1) (actual time=0.004..0.004 rows=1 loop
s=34)
            -> Table scan on <temporary>  (cost=0.19..2.66 rows=14) (actual time=0.000..0.001 rows=7 loops=1)
                -> Temporary table with deduplication  (cost=157.66..160.13 rows=14) (actual time=0.486..0.487 rows=7 loops=1)
                    -> Nested loop inner join  (cost=156.09 rows=14) (actual time=0.378..0.474 rows=7 loops=1)
                        -> Filter: (m.AgeRating = '13+')  (cost=144.26 rows=32) (actual time=0.085..0.367 rows=34 loops=1)
                            -> Index range scan on m using movie_year, with index condition: (m.`Year` >= 2021)  (cost=144.26 rows=320) (act
ual time=0.085..0.344 rows=320 loops=1)
                        -> Filter: (a.PlatformName = 'Prime Video')  (cost=0.27 rows=0) (actual time=0.003..0.003 rows=0 loops=34)
                            -> Index lookup on a using MovieId (MovieId=m.MovieId)  (cost=0.27 rows=1) (actual time=0.002..0.003 rows=1 loop
s=34)
 |
+------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------------------------------------------
```

We created an index on the movie year because movie year is a vital part of this query. An indexing structure on the movie year works better because we create pointers to where the movie years are stored in the database. Instead of having to sequentially traverse every movie year in the Movie table, an indexing structure on the movie year provides a quick ordering of data on the disk or tells the SQL engine exactly where to go via pointers.

User Trigger for hashing password and creating UserId

```
CREATE TRIGGER UserTrig BEFORE INSERT ON User FOR EACH ROW
BEGIN
SET @UserExists = EXISTS(SELECT * FROM User WHERE User.UserId = NEW.UserId);
SET NEW.Password = MD5(NEW.Password);
IF (@UserExists) THEN SET NEW.UserId = (SELECT MAX(UserId) FROM User) + 1;
END IF;
END$$

CREATE TRIGGER UserTrig2 AFTER INSERT ON User FOR EACH ROW
BEGIN
SET @NewListId = (SELECT MAX(ListId) FROM MovieList) + 1;
SET @NewListId2 = (SELECT MAX(ListId) FROM MovieList) + 2;
INSERT IGNORE INTO MovieList(ListId) VALUES(@NewListId);
INSERT IGNORE INTO MovieList(ListId) VALUES(@NewListId2);
INSERT IGNORE INTO BlackList(ListId, UserId) VALUES (@NewListId,NEW.UserId);
INSERT IGNORE INTO WatchList(ListId, UserId) VALUES (@NewListId2, NEW.UserId);
END$$


SELECT u.UserId, w.ListId as WatchListId, b.ListId as BlackListId FROM User u Join WatchList
w ON u.UserId = w.UserId JOIN BlackList b ON u.UserId = b.UserId WHERE u.userName =
"work" AND u.password = MD5("test");
```

Stored Procedure

```
CREATE PROCEDURE ratingUpdate()
BEGIN
        DECLARE varMovieId INT;
        DECLARE varAvgRating REAL;
        DECLARE varNumRatings INT;
        DECLARE varReview INT;
        DECLARE loop_exit BOOLEAN DEFAULT FALSE;

        DECLARE ratingInfo CURSOR FOR (
                SELECT m.MovieId, AVG(r.Score), COUNT(r.UserId)
                FROM Movie m INNER JOIN Rating r ON m.MovieId = r.MovieId
                GROUP BY m.MovieId
        ) ;
        DECLARE CONTINUE HANDLER FOR NOT FOUND SET loop_exit=TRUE;


        DROP TABLE IF EXISTS Review;
        CREATE TABLE Review(
             MovieId INT PRIMARY KEY,
             Review INT
        );


        OPEN ratingInfo;
        cloop: LOOP
                FETCH ratingInfo INTO varMovieId, varAvgRating, varNumRatings;
                IF loop_exit THEN
                        LEAVE cloop;
                END IF;
                IF (varNumRatings < 3) THEN
                        SET varReview = 0;
                END IF;
                IF (varNumRatings >= 3 AND varAvgRating > 2.5) THEN
                        SET varReview = 1;
                END IF;
                IF (varNumRatings >= 3 AND varAvgRating <= 2.5) THEN
                        SET varReview = 2;
                END IF;
                INSERT IGNORE INTO Review VALUES (varMovieId, varReview);
        END LOOP cloop;
        CLOSE ratingInfo;
```

END$$