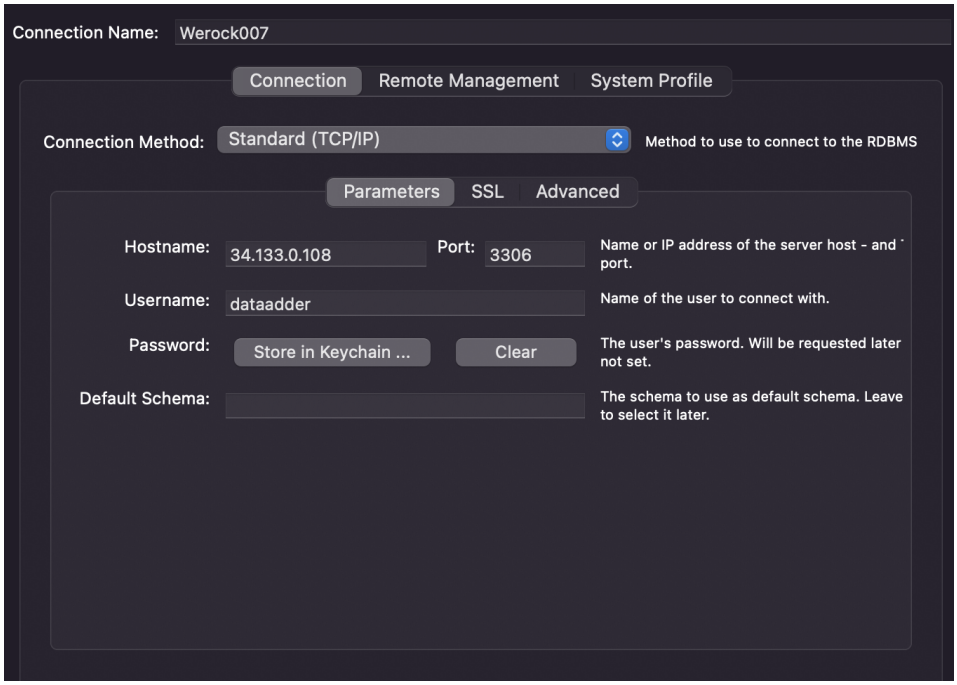
Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

Local databases + screenshot of connection




```
Welcome to Cloud Shell! Type "help" to get started.
Your Cloud Platform project in this session is set to werock007.
Use “gcloud config set project [PROJECT_ID]” to change to a different project.
aarushi_biswas28@cloudshell:~ (werock007)$ gcloud sql connect kartik-instance --user=dataadder
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [dataadder].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 765
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> SHOW DATABASES
    -> ;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| performance_schema |
| sys                |
| test               |
+--------------------+
5 rows in set (0.00 sec)

mysql> USE test
```

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

```
mysql> USE test
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_test |
+----------------+
| AKAS           |
| AKAS2          |
| Actors         |
| Actors2        |
| Acts           |
| Acts2          |
| Crew           |
| Crew2          |
| Episodes       |
| Episodes2      |
| Movies         |
| Principals     |
| Principals2    |
| Ratings        |
| Ratings2       |
| TV             |
| UserLogin      |
| UserProfile    |
| titleBasics    |
| titleBasics2   |
+----------------+
20 rows in set (0.00 sec)
```

This screenshot shows both our raw tables and our entity tables. We used our raw tables (which we had to create twice due to technical difficulties) to create our ‘actual’ entity tables. The entity tables are: Movies, TV, Actors, Acts, UserLogin and UserProfile

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

Create Table Commands

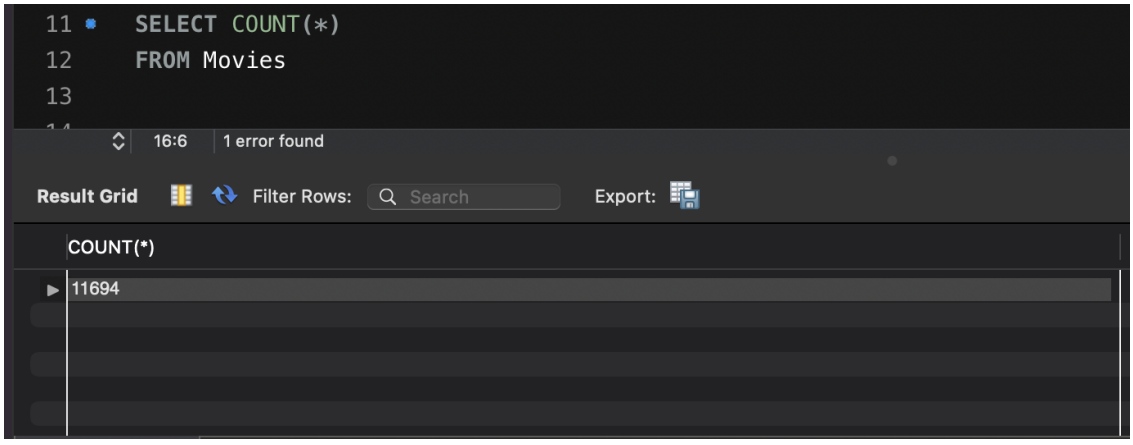
1) Movies table

```
CREATE TABLE Movies(
        titleId VARCHAR(250) PRIMARY KEY,
        title VARCHAR(250),
        genres VARCHAR(250),
        startYear INT,
        languages VARCHAR(250),
        averageRating FLOAT,
        directors VARCHAR(250),
        runtimeMinutes INT,
        isAdult INT
);
```

Populating table

```
INSERT INTO Movies(titleId, title, genres, startYear, languages, averageRating,
directors, runtimeMinutes, isAdult)

SELECT akas.titleId, akas.title, tb.genres, tb.startYear, akas.language, r.averageRating,
c.directors, tb.runtimeMinutes, tb.isAdult
FROM Crew c NATURAL JOIN Ratings r NATURAL JOIN titleBasics tb JOIN AKAS
akas ON (akas.titleId = titleBasics.tconst)
WHERE tb.titleType = 'movie';
```



2) TV Shows table

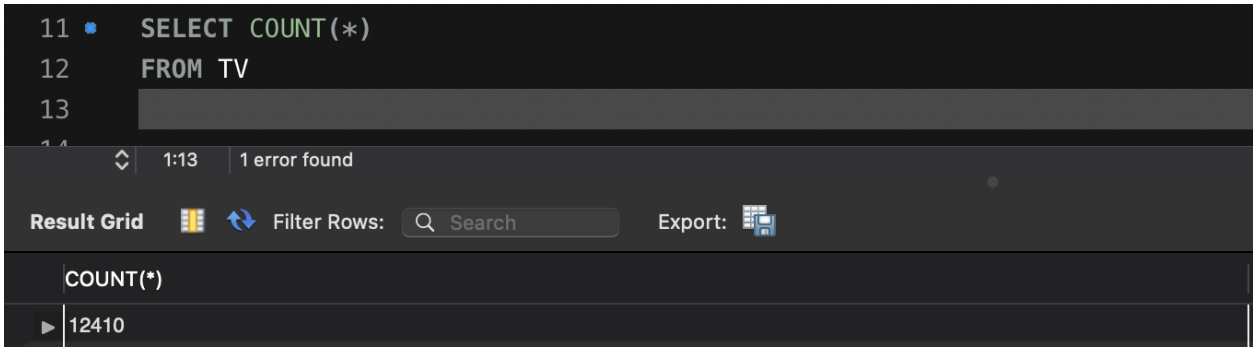
Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

```
CREATE TABLE TV(
        titleId VARCHAR(250) PRIMARY KEY,
        title VARCHAR(250),
        genres VARCHAR(250),
        startYear INT,
        averageRating FLOAT,
        directors VARCHAR(250),
        runtimeMinutes INT,
        isAdult INT,
        seasonNum INT,
        episodeNum INT
);
```

Populating table

```
INSERT INTO TV(titleId, title, genres, startYear, languages, averageRating, directors,
runtimeMinutes, isAdult, seasonNum, episodeNum)

SELECT akas.titleId, akas.title, tb.genres, tb.startYear, akas.language, r.averageRating,
c.directors, tb.runtimeMinutes, tb.isAdult, ep.seasonNum, ep.episodeNum
FROM Crew2 c NATURAL JOIN Ratings2 r NATURAL JOIN titleBasics2 tb JOIN
AKAS2 akas ON (akas.titleId = tb.tconst) NATURAL JOIN
        (SELECT te.tconst, COUNT(seasonNumber) as seasonNum,
        COUNT(episodeNumber) as episodeNum FROM Episodes2 te GROUP BY
        te.parentTconst) as ep
WHERE tb.titleType LIKE 'tv%' AND akas.isOriginalTitle = 1;
```



3) Actors

```
CREATE TABLE Actors (
```

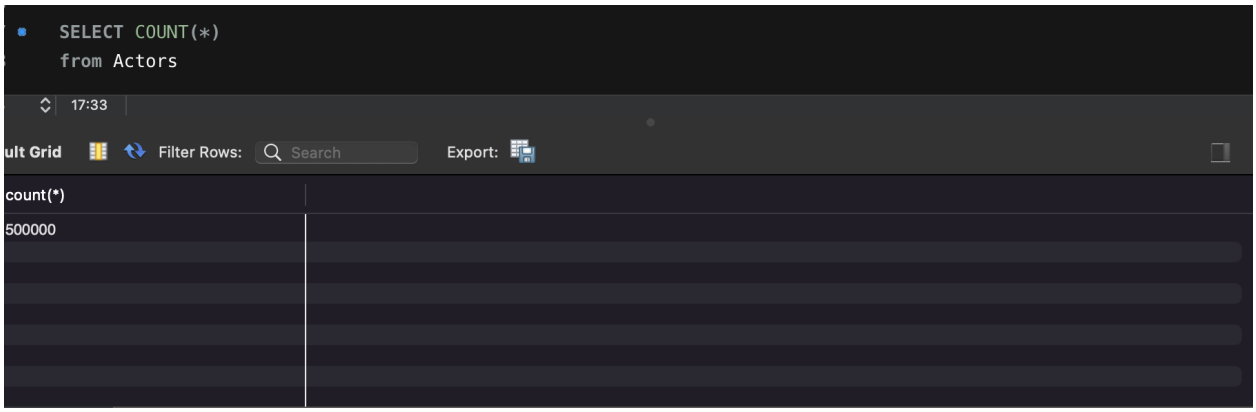
Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

```
    actorId VARCHAR(250) PRIMARY KEY,
    actorName VARCHAR (250),
    birthYear INT
);
```

Populating table

```
INSERT INTO Actors

SELECT nconst, primaryName, birthYear FROM Actors2;
```



4) Acts Table - relational table between Movies and Actors

```
CREATE TABLE Acts(
titleId VARCHAR(250),
Ordering INT,
ActorId VARCHAR(250),
Category VARCHAR(250),
primary key (titleId, ActorId)
);
```

Populating table

```
INSERT INTO Acts(titleId, Ordering, ActorId, Category)

SELECT tconst, ordering, nconst, category
FROM Principals2
WHERE category LIKE "actor" or category LIKE "actress"
ORDER BY tconst, ordering;
```

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

ADVANCED SUBQUERIES

1.
GroupBy, Joining multiple relations

```
SELECT DISTINCT genres, avgRuntime
FROM Movies JOIN(
   SELECT AVG(runtimeMinutes) AS avgRuntime, genres
   FROM Movies m1
   GROUP BY m1.genres
   ) as bob using(genres)
ORDER BY avgRuntime DESC;
```

Output:

genres	avgRuntime
War	78.6087
Action	77.2504
Crime	74.8443
Drama	72.3065
Horror	71.3182
Musical	69.8182
Adult	67.1538
Film-Noir	66.5000
Sci-Fi	66.2083
Adventure	64.5711
Mystery	64.0536
Western	63.3221
Family	60.0000
History	56.8000
Romance	56.4242
Comedy	54.6091
Fantasy	38.9821
Docume…	33.0944
Music	31.5088
Short	10.9799
Animation	7.1491
News	2.4615

Default Indexes

```
SHOW INDEX FROM Movies;
```

	Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
▶	Movies	0	PRIMARY	1	titleId	A	11763	NULL	NULL		BTREE			YES	NULL

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

EXPLAIN

```
-> Sort: bob.avgRuntime DESC  (actual time=83.282..83.286 rows=55 loops=1)
   -> Table scan on <temporary>  (cost=2.50..2.50 rows=0) (actual time=0.002..0.008 rows=55 loops=1)
      -> Temporary table with deduplication  (cost=8070.34..8070.34 rows=0) (actual time=83.226..83.236 rows=55 loops=1)
         -> Filter: (bob.genres = Movies.genres)  (cost=8067.84 rows=0) (actual time=68.654..78.925 rows=11694 loops=1)
            -> Inner hash join (<hash>(bob.genres)=<hash>(Movies.genres))  (cost=8067.84 rows=0) (actual time=68.651..76.241 rows=11694 loops=1)
               -> Table scan on bob  (cost=2.50..2.50 rows=0) (actual time=0.001..0.016 rows=55 loops=1)
                  -> Materialize  (cost=2.50..2.50 rows=0) (actual time=12.020..12.041 rows=55 loops=1)
                     -> Table scan on <temporary>  (actual time=0.002..0.008 rows=55 loops=1)
                        -> Aggregate using temporary table  (actual time=11.918..11.928 rows=55 loops=1)
                           -> Table scan on m1  (cost=1199.55 rows=11753) (actual time=0.047..3.541 rows=11694 loops=1)
               -> Hash
                  -> Table scan on Movies  (cost=1199.55 rows=11753) (actual time=0.067..4.275 rows=11694 loops=1)
```

18:56:27 EXPLAIN ANALYZE SELECT distinct genres, avgRuntime FROM Movies JOIN(... 1 row(s) returned 0.105 sec / 0.000075...

Duration: 0.105 s

Indexing 1: Index on Movies(genres)

CREATE INDEX idx1 on Movies(genres);
SHOW INDEX FROM Movies;

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
Movies	0	PRIMARY	1	titleId	A	11753	NULL	NULL		BTREE			YES
Movies	1	idx1	1	genres	A	55	NULL	NULL	YES	BTREE			YES

EXPLAIN

```
-> Sort: bob.avgRuntime DESC  (actual time=32.820..32.824 rows=55 loops=1)
   -> Table scan on <temporary>  (cost=0.01..31396.36 rows=2511509) (actual time=0.002..0.007 rows=55 loops=1)
      -> Temporary table with deduplication  (cost=994009.79..1025406.14 rows=2511509) (actual time=32.772..32.785 rows=55 loops=1)
         -> Nested loop inner join  (cost=742858.86 rows=2511509) (actual time=23.790..28.856 rows=11694 loops=1)
            -> Filter: (bob.genres is not null)  (cost=3549.96..1324.71 rows=11753) (actual time=23.741..23.760 rows=55 loops=1)
               -> Table scan on bob  (cost=0.01..149.41 rows=11753) (actual time=0.003..0.012 rows=55 loops=1)
                  -> Materialize  (cost=3550.16..3699.56 rows=11753) (actual time=23.738..23.751 rows=55 loops=1)
                     -> Group aggregate: avg(m1.runtimeMinutes)  (cost=2374.85 rows=11753) (actual time=2.013..23.600 rows=55 loops=1)
                        -> Index scan on m1 using idx1  (cost=1199.55 rows=11753) (actual time=0.308..20.977 rows=11694 loops=1)
            -> Index lookup on Movies using idx1 (genres=bob.genres)  (cost=41.73 rows=214) (actual time=0.008..0.079 rows=213 loops=55)
```

18:58:43 EXPLAIN ANALYZE SELECT distinct genres, avgRuntime FROM Movies JOIN(... 1 row(s) returned 0.052 sec / 0.000051...

Duration: 0.052

We can observe that the time is reduced by 50% and this is because we are grouping by genres and hence having an index pointing to the genres allows for faster access to all the entries with a specific genre as we no longer have to index via titleId.

Indexing 2: Index on Movies(Runtime)

CREATE INDEX idx2 on Movies(runtimeMinutes);
SHOW INDEX FROM Movies;

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
Movies	0	PRIMARY	1	titleId	A	11753	NULL	NULL		BTREE			YES
Movies	1	idx2	1	runtimeMinutes	A	212	NULL	NULL	YES	BTREE			YES

EXPLAIN

```
-> Sort: bob.avgRuntime DESC  (actual time=47.168..47.172 rows=55 loops=1)
  -> Table scan on <temporary>  (cost=2.50..2.50 rows=0) (actual time=0.002..0.008 rows=55 loops=1)
    -> Temporary table with deduplication  (cost=8070.34..8070.34 rows=0) (actual time=47.131..47.140 rows=55 loops=1)
      -> Filter: (bob.genres = Movies.genres)  (cost=8067.84 rows=0) (actual time=31.898..43.112 rows=11694 loops=1)
        -> Inner hash join (<hash>(bob.genres)=<hash>(Movies.genres))  (cost=8067.84 rows=0) (actual time=31.895..40.582 rows=11694 loops=1)
          -> Table scan on bob  (cost=2.50..2.50 rows=0) (actual time=0.001..0.006 rows=55 loops=1)
            -> Materialize  (cost=2.50..2.50 rows=0) (actual time=11.247..11.257 rows=55 loops=1)
              -> Table scan on <temporary>  (actual time=0.001..0.009 rows=55 loops=1)
                -> Aggregate using temporary table  (actual time=11.107..11.117 rows=55 loops=1)
                  -> Table scan on m1  (cost=1199.55 rows=11753) (actual time=0.178..3.665 rows=11694 loops=1)
          -> Hash
            -> Table scan on Movies  (cost=1199.55 rows=11753) (actual time=0.069..3.165 rows=11694 loops=1)
```

19:09:24 EXPLAIN ANALYZE SELECT distinct genres, avgRuntime FROM Movies JOIN(... 1 row(s) returned 0.069 sec / 0.000016...

Duration: 0.069

This is an interesting decrease as we didn't expect such a large difference. The decrease is about 30%. We think since we are averaging the runtime minutes, having an index quickens this query up as we think that the runtime minutes for each entry could necessarily not be in the same block and hence having an index directly to it in memory allows us to access it quicker then if we were to search for it by going to each titleId.

Indexing 3: Index on Movies(genres) and Movies(runtimeMinutes)

```
CREATE INDEX idx1 on Movies(genres);
CREATE INDEX idx2 on Movies(runtimeMinutes);
SHOW INDEX FROM Movies;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible
Movies	0	PRIMARY	1	titleId	A	11753	NULL	NULL		BTREE			YES
Movies	1	idx2	1	runtimeMinutes	A	212	NULL	NULL	YES	BTREE			YES
Movies	1	idx1	1	genres	A	55	NULL	NULL	YES	BTREE			YES

EXPLAIN

```
-> Sort: bob.avgRuntime DESC  (actual time=31.661..31.665 rows=55 loops=1)
  -> Table scan on <temporary>  (cost=0.01..31396.36 rows=2511509) (actual time=0.002..0.006 rows=55 loops=1)
    -> Temporary table with deduplication  (cost=994009.79..1025406.14 rows=2511509) (actual time=31.615..31.623 rows=55 loops=1)
      -> Nested loop inner join  (cost=742858.86 rows=2511509) (actual time=22.537..27.675 rows=11694 loops=1)
        -> Filter: (bob.genres is not null)  (cost=3549.96..1324.71 rows=11753) (actual time=22.490..22.508 rows=55 loops=1)
          -> Table scan on bob  (cost=0.01..149.41 rows=11753) (actual time=0.002..0.010 rows=55 loops=1)
            -> Materialize  (cost=3550.16..3699.56 rows=11753) (actual time=22.488..22.500 rows=55 loops=1)
              -> Group aggregate: avg(m1.runtimeMinutes)  (cost=2374.85 rows=11753) (actual time=1.325..22.413 rows=55 loops=1)
                -> Index scan on m1 using idx1  (cost=1199.55 rows=11753) (actual time=0.173..19.798 rows=11694 loops=1)
        -> Index lookup on Movies using idx1 (genres=bob.genres)  (cost=41.73 rows=214) (actual time=0.007..0.080 rows=213 loops=55)
```

19:11:56 EXPLAIN ANALYZE SELECT distinct genres, avgRuntime FROM Movies JOIN(... 1 row(s) returned 0.050 sec / 0.000017...

Duration: 0.050

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

Here we can see the time is lowest and this is the case where we both have the indexes to the runtime minutes and the genre. This is bound to run the fastest as it has the advantages of both the indexes. However, the time doesn't decrease as much as the indexes decrease the time separately and we believe this is due to some overlap as in some cases the genre pointer is pointing to the same block of memory as the runtime minutes block of memory and in this case we don't save as much time as having both indexes in this case is redundant.

2.

GroupBy, Set operations

```
(SELECT genres, COUNT(*) as FREQ
 FROM Movies m
 GROUP BY m.genres)
UNION
(SELECT genres, COUNT(*) as FREQ
 FROM TV t
 GROUP BY t.genres) ;
```

Output:

genres	FREQ
Animation	1556
Biography	130
Comedy	2630
Crime	623
Documentary	657
Drama	2493
Family	38
Fantasy	56
Film-Noir	4
History	15
Horror	220
Music	57
Musical	77
Mystery	56
News	13
Romance	99
Sci-Fi	24
Short	448
Thriller	28

Default Indexes

Default index for Movies

```
SHOW INDEX FROM Movies;
```

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
Movies	0	PRIMARY	1	titleId	A	11763	NULL	NULL		BTREE			YES	NULL

Default index for TV

```
SHOW INDEX FROM TV;
```

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
TV	0	PRIMARY	1	titleId	A	13014	NULL	NULL		BTREE			YES	NULL

EXPLAIN

```
-> Table scan on <union temporary>  (cost=2.50..2.50 rows=0) (actual time=0.001..0.011 rows=143 loops=1)
    -> Union materialize with deduplication  (cost=2.50..2.50 rows=0) (actual time=19.286..19.305 rows=143 loops=1)
        -> Table scan on <temporary>  (actual time=0.001..0.006 rows=55 loops=1)
            -> Aggregate using temporary table  (actual time=9.205..9.213 rows=55 loops=1)
                -> Table scan on m  (cost=1271.16 rows=11753) (actual time=0.042..3.149 rows=11694 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.013 rows=88 loops=1)
            -> Aggregate using temporary table  (actual time=9.977..9.993 rows=88 loops=1)
                -> Table scan on t  (cost=1341.65 rows=13014) (actual time=0.039..3.496 rows=12410 loops=1)
```

215 17:34:49 EXPLAIN ANALYZE (SELECT genres, COUNT(*) as FREQ FROM Movies m GROUP BY m.gen... 1 row(s) returned 0.037 sec / 0.000010...

Duration: 0.037s

Indexing 1: Index on genres on Movie

CREATE INDEX idx1 on Movies(genres);
SHOW INDEX FROM Movies;

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
Movies	0	PRIMARY	1	titleId	A	11753	NULL	NULL		BTREE			YES	NULL
Movies	1	idx1	1	genres	A	55	NULL	NULL	YES	BTREE			YES	NULL

EXPLAIN ANALYZE

EXPLAIN

```
-> Table scan on <union temporary>  (cost=0.01..149.41 rows=11753) (actual time=0.001..0.010 rows=143 loops=1)
    -> Union materialize with deduplication  (cost=3621.78..3771.18 rows=11753) (actual time=14.343..14.360 rows=143 loops=1)
        -> Group aggregate: count(0)  (cost=2446.46 rows=11753) (actual time=0.161..4.432 rows=55 loops=1)
            -> Index scan on m using idx1  (cost=1271.16 rows=11753) (actual time=0.037..2.471 rows=11694 loops=1)
        -> Table scan on <temporary>  (actual time=0.002..0.014 rows=88 loops=1)
            -> Aggregate using temporary table  (actual time=9.809..9.826 rows=88 loops=1)
                -> Table scan on t  (cost=1341.65 rows=13014) (actual time=0.300..3.698 rows=12410 loops=1)
```

212 17:33:58 EXPLAIN ANALYZE (SELECT genres, COUNT(*) as FREQ FROM Movies m GROUP BY m.gen... 1 row(s) returned 0.032 sec / 0.000051...

Duration: 0.032s

We can see this is an approximate 13% decrease in the time from the default indexes. We believe this slight change is attributed by the fact that we are grouping by genres and at least for the part of the query that goes through the Movies table it has an index to the genre and hence it is faster in finding movies with a certain genre. However, the

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

tv table still doesn't have an index to the genre hence it still needs to go over each tv show and find out the genre for that show.

Indexing 2: Index on genres on Tv

CREATE INDEX idx1 on TV(genres);
SHOW INDEX FROM TV;

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
TV	0	PRIMARY	1	titleId	A	13014	NULL	NULL		BTREE			YES	NULL
TV	1	idx1	1	genres	A	88	NULL	NULL	YES	BTREE			YES	NULL

EXPLAIN ANALYZE

EXPLAIN

```
-> Table scan on <union temporary>  (cost=0.01..165.18 rows=13014) (actual time=0.001..0.012 rows=143 loops=1)
   -> Union materialize with deduplication  (cost=3944.46..4109.62 rows=13014) (actual time=13.955..13.973 rows=143 loops=1)
      -> Table scan on <temporary>  (actual time=0.001..0.007 rows=55 loops=1)
         -> Aggregate using temporary table  (actual time=9.134..9.143 rows=55 loops=1)
            -> Table scan on m  (cost=1271.16 rows=11753) (actual time=0.040..3.216 rows=11694 loops=1)
      -> Group aggregate: count(0)  (cost=2643.05 rows=13014) (actual time=0.059..4.737 rows=88 loops=1)
         -> Index scan on t using idx1  (cost=1341.65 rows=13014) (actual time=0.050..2.599 rows=12410 loops=1)
```

207 17:29:00 EXPLAIN ANALYZE (SELECT genres, COUNT(*) as FREQ FROM Movies m GROUP BY m.gen... 1 row(s) returned 0.032 sec / 0.000018...

Duration: 0.032s

We can see this is again a 13% decrease in the time from the default indexes. We believe this slight change is attributed to the fact that we are grouping by genres and at least for the part of the query that goes through the TV table it has an index to the genre and hence it is faster in finding TV shows with a certain genre. However, the movie table still doesn't have an index to the genre hence it still needs to go over each movie and find out the genre for that movie.

Indexing 3: Index on genres on Tv and Movies
CREATE INDEX idx1 on Movies(genres);
SHOW INDEX FROM Movies;

Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
Movies	0	PRIMARY	1	titleId	A	11753	NULL	NULL		BTREE			YES	NULL
Movies	1	idx1	1	genres	A	55	NULL	NULL	YES	BTREE			YES	NULL

CREATE INDEX idx1 on TV(genres);
SHOW INDEX FROM TV;

Note: While populating our entity tables we are using queries which pull from raw tables wemade using the IMDB database which are not shown.

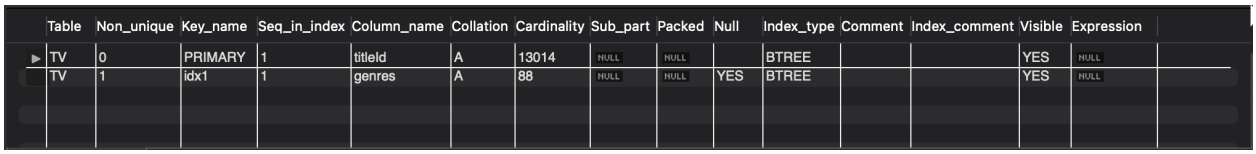


Table	Non_unique	Key_name	Seq_in_index	Column_name	Collation	Cardinality	Sub_part	Packed	Null	Index_type	Comment	Index_comment	Visible	Expression
TV	0	PRIMARY	1	titleId	A	13014	NULL	NULL		BTREE			YES	NULL
TV	1	idx1	1	genres	A	88	NULL	NULL	YES	BTREE			YES	NULL

EXPLAIN ANALYZE

EXPLAIN

```
-> Table scan on <union temporary>  (cost=0.01..312.09 rows=24767) (actual time=0.003..0.011 rows=143 loops=1)
   -> Union materialize with deduplication  (cost=7566.23..7878.30 rows=24767) (actual time=9.088..9.105 rows=143 loops=1)
      -> Group aggregate: count(0)  (cost=2446.46 rows=11753) (actual time=0.129..4.358 rows=55 loops=1)
         -> Index scan on m using idx1  (cost=1271.16 rows=11753) (actual time=0.039..2.423 rows=11694 loops=1)
      -> Group aggregate: count(0)  (cost=2643.05 rows=13014) (actual time=0.035..4.647 rows=88 loops=1)
         -> Index scan on t using idx1  (cost=1341.65 rows=13014) (actual time=0.034..2.567 rows=12410 loops=1)
```

Duration: 0.027s

We can see this is again a 24% decrease in the time from the default indexes. We believe this slight change is attributed by the fact that we are grouping by genres and now since both parts of the query the movie and TV table have indexes to the genre and hence it can easily search by index for both parts and group them faster.