- ☑ ~~Implement at least four main tables (i.e., tables that include core application information, not auxiliary information, such as user profiles and login information).~~
- ☑ ~~In the Database Design markdown or pdf, provide the Data Definition Language (DDL) commands you all used to create each of these tables in the database. Here's the syntax of the CREATE TABLE DDL command:~~
  ~~CREATE TABLE table_name (column1 datatype, column2 datatype, column3 datatype,...);~~
- ☑ ~~Insert data into these tables. You should insert at least 1000 rows **each in** three of the tables. Try to use real data, but if you cannot find a good dataset for a particular table, you may use auto-generated data.~~
- ☑ ~~As a group, develop your application's advanced SQL queries. These queries should be relevant to the functionality of your application and are expected to be part of your final application. The queries should **each** involve **at least two** of the following SQL concepts:~~
  - Join of multiple relations
  - Set operations
  - Aggregation via GROUP BY
  - Subqueries
- ☑ ~~Execute your advanced SQL queries and provide a screenshot of the top 15 rows of each query result (you can use the LIMIT clause to select the top 15 rows). If your output is less then 15 rows, say that in your output.~~

Part 2 Indexing: As a team, for each advanced query:

- ☑ ~~Use the EXPLAIN ANALYZE command to measure your advanced query performance before adding indexes.~~
- ☑ ~~Explore adding different indices to different attributes on the advanced query. For each indexing design you try, use the EXPLAIN ANALYZE command to measure the query performance after adding the indices.~~
- ☑ ~~Report on the index design you all select and explain why you chose it, referencing the analysis you performed in (b).~~
- ☑ ~~Note that if you did not find any difference in your results, report that as well. Explain why you think this change in indexing did not bring a better effect to your query.~~

Rubric: This stage is worth 30%. You are graded by completeness and correctness. For completeness, each justification should be **at least one paragraph long**. The rubric is as follows:

- ☐ Create a release with the correct tag for your submission and submit it on canvas (-2% for incorrect release)
- ☐ Does not have a submission located within the doc folder (-0.5%)
- ☑ ~~Database implementation is worth 10% and is graded (as a group) as follows:~~

- ☑ ~~+4% for implementing the database tables locally or on GCP, you should provide a screenshot of the connection (i.e. showing your terminal/command-line information)~~
- ☑ ~~+4% for providing the DDL commands for your tables. (-0.5% for each mistake)~~
- ☑ ~~+2% for inserting at least 1000 rows in the tables. (You should do a count query to show this, -1% for each missing table)~~

☑ ~~Advanced Queries are worth 10% and are graded (as a group) as follows:~~

- ☑ ~~+8% for developing two advanced queries (see point 4 for this stage, 4% each)~~
- ☑ ~~+2% for providing screenshots with the top 15 rows of the advanced query results (1% each)~~

☑ ~~Indexing Analysis is worth 10% and is graded (as a group) as follows:~~

- ☑ ~~+3% on trying at least three different indexing designs (excluding the default index) for each advanced query.~~
- ☑ ~~+5% on the indexing analysis reports which includes screenshots of the EXPLAIN ANALYZE commands.~~
- ☑ ~~+2% on the accuracy and thoroughness of the analyses.~~

TABLES:
iMDB_Titles, iMDB_Names, Genres, KnownFor

```
mysql> show tables
    -> ;
+---------------+
| Tables_in_bcg |
+---------------+
| Genres        |
| KnownFor      |
| YT_Trending   |
| iMDB_Names    |
| iMDB_Titles   |
+---------------+
5 rows in set (0.00 sec)
```

```
mysql> SELECT COUNT(*) FROM Genres;
+----------+
| COUNT(*) |
+----------+
|    13297 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT COUNT(*) FROM KnownFor;
+----------+
| COUNT(*) |
+----------+
|     9113 |
+----------+
1 row in set (0.00 sec)

mysql> SELECT COUNT(*) FROM YT_Trending;
+----------+
| COUNT(*) |
+----------+
|   220919 |
+----------+
1 row in set (4.79 sec)
```

```
mysql> SELECT COUNT(*) FROM iMDB_Names;
+----------+
| COUNT(*) |
+----------+
|   695994 |
+----------+
1 row in set (0.74 sec)

mysql> SELECT COUNT(*) FROM iMDB_Titles;
+----------+
| COUNT(*) |
+----------+
|     5000 |
+----------+
1 row in set (0.00 sec)
```

```
CREATE TABLE iMDB_Titles(
    titleID VARCHAR(255),
    PRIMARY KEY (titleID),
    titleType VARCHAR(255),
    primaryTitle VARCHAR(255),
    originalTitle VARCHAR(255),
    isAdult INT,
    startYear INT,
    endYear INT,
    runtimeMinutes INT,
    averageRating FLOAT,
    numVotes INT
);

CREATE TABLE iMDB_Names(
    nameID VARCHAR(255),
    PRIMARY KEY (nameID),
    nameDetails VARCHAR(255),
    primaryName VARCHAR(255),
    birthYear INT,
    deathYear INT
);

CREATE TABLE YT_Trending(
    VideoId VARCHAR(255),
    TrendingDate DATE,
    PRIMARY KEY(VideoId, TrendingDate),
    title VARCHAR(255),
    publishedAt DATETIME,
    channelId VARCHAR(255),
    channelTitle VARCHAR(255),
    categoryId VARCHAR(255),
    tags VARCHAR(500),
    view_count INT,
    likes INT,
    dislikes INT,
    comment_count INT,
    thumbnail_link VARCHAR(255),
    comments_disabled BOOL,
    ratings_disabled BOOL,
    description VARCHAR(5000)
);

CREATE TABLE Genres(
```

```
    titleID VARCHAR(255),
    FOREIGN KEY (titleID) REFERENCES iMDB_Titles(titleID),
    genreName VARCHAR(255),
    PRIMARY KEY (titleID,genreName)
    );

CREATE TABLE KnownFor(
    nameID VARCHAR(255),
    titleID VARCHAR(255),
    PRIMARY KEY (titleID, nameID),
    FOREIGN KEY (titleID) REFERENCES iMDB_Titles(titleID),
    FOREIGN KEY (nameID) REFERENCES iMDB_Names(nameID)
    );
```

input1 = First Input

```
DROP TABLE IF EXISTS titleStart;

CREATE TEMPORARY TABLE titleStart
select * from iMDB_Titles t
NATURAL JOIN KnownFor kf
where t.primaryTitle = 'Fight Club' or t.originalTitle = 'Fight Club';


INNER JOIN (Genres gn)
ON (t.titleID = gn.titleID)
```

input2 = Second Input

```
CREATE VIEW genreSecond AS
SELECT *
FROM titleStart ts
WHERE genreName = input2
INNER JOIN KnownFor kf
ON (titleStart.titleID = kf.titleID)
```

input3 = Third Input

```
CREATE VIEW knownThird AS
SELECT *
FROM genreSecond
INNER JOIN (SELECT primaryName, nameID FROM Names WHERE primaryName == Third
Input) namer
ON (genreSecond.nameID = namer.nameID)
WHERE input3 = primaryName
```

Advanced Queries

(1)

```
SELECT FLOOR(startYear/10)*10 AS Decade, gn.genreName AS Genre,
COUNT(it.primaryTitle) AS NumberOfTitles
FROM iMDB_Titles it
NATURAL JOIN Genres gn
GROUP BY Decade, Genre
ORDER BY Decade;
```

First 15 Rows:

Decade	Genre	NumberOfTitles
1900	Action	1
1900	Adventure	1
1900	Comedy	1
1920	Action	2
1920	Adventure	2
1920	Biography	1
1920	Comedy	4
1920	Drama	6
1920	Family	1
1920	Fantasy	2
1920	History	2
1920	Horror	3
1920	Mystery	1
1920	Romance	2
1920	Sci-Fi	1

(2)

```
SELECT primaryName, primaryTitle
FROM iMDB_Names NATURAL JOIN (iMDB_Titles NATURAL JOIN KnownFor)
WHERE titleID IN(
    SELECT titleID
    FROM Genres
    WHERE genreName = "Horror") limit 15;
```

First 15 Rows:

primaryName	primaryTitle
James Bernard	Nosferatu
F.W. Murnau	Nosferatu
Bela Lugosi	Dracula
James Whale	Frankenstein
Franz Waxman	The Bride of Frankenstein
Boris Karloff	The Bride of Frankenstein
James Whale	The Bride of Frankenstein
Sam Peckinpah	Invasion of the Body Snatchers
Kevin McCarthy	Invasion of the Body Snatchers
Mary Badham	The Twilight Zone
Sue Randall	The Twilight Zone
Alfred Hitchcock	Psycho
Anthony Perkins	Psycho
Martin Balsam	Psycho
John Gavin	Psycho

Indexing

Indexing Designs on query (1):

No new indexing EXPLAIN ANALYZE

```
| EXPLAIN

                                    |
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
-------------------------------------+
| -> Sort: Decade, gn.genreName  (actual time=35.853..35.867 rows=215 loops=1)
    -> Table scan on <temporary>  (actual time=35.708..35.762 rows=215 loops=1)
        -> Aggregate using temporary table  (actual time=35.706..35.706 rows=215 loops=1)
            -> Nested loop inner join  (cost=3529.53 rows=11627) (actual time=0.076..21.587 rows=13297 loops=1)
                -> Table scan on it  (cost=539.35 rows=5151) (actual time=0.055..2.165 rows=5000 loops=1)
                -> Covering index lookup on gn using PRIMARY (titleID=it.titleID)  (cost=0.35 rows=2) (actual time=0.003..0.004 rows=3 loops=5000)
 |
```

(1)

```
mysql> CREATE INDEX genreName_idx on Genres(genreName)
```

```
| EXPLAIN

                                    |
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
-------------------------------------+
| -> Sort: Decade, gn.genreName  (actual time=38.535..38.550 rows=215 loops=1)
    -> Table scan on <temporary>  (actual time=38.393..38.446 rows=215 loops=1)
        -> Aggregate using temporary table  (actual time=38.391..38.391 rows=215 loops=1)
            -> Nested loop inner join  (cost=3529.53 rows=11627) (actual time=0.136..24.557 rows=13297 loops=1)
                -> Table scan on it  (cost=539.35 rows=5151) (actual time=0.057..4.020 rows=5000 loops=1)
                -> Covering index lookup on gn using PRIMARY (titleID=it.titleID)  (cost=0.35 rows=2) (actual time=0.003..0.004 rows=3 loops=5000)
 |
```

(2)

```
CREATE INDEX startYear_idx ON iMDB_Titles(startYear);
```

```
| EXPLAIN

                                    |
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
-------------------------------------+
| -> Sort: Decade, gn.genreName  (actual time=37.121..37.135 rows=215 loops=1)
    -> Table scan on <temporary>  (actual time=36.950..37.028 rows=215 loops=1)
        -> Aggregate using temporary table  (actual time=36.947..36.947 rows=215 loops=1)
            -> Nested loop inner join  (cost=3529.53 rows=11627) (actual time=0.124..22.461 rows=13297 loops=1)
                -> Table scan on it  (cost=539.35 rows=5151) (actual time=0.095..2.317 rows=5000 loops=1)
                -> Covering index lookup on gn using PRIMARY (titleID=it.titleID)  (cost=0.35 rows=2) (actual time=0.003..0.004 rows=3 loops=5000)
 |
```

(3)

```
CREATE INDEX primarytitle_idx ON iMDB_Titles(primaryTitle);
```

```
| EXPLAIN

                                    |
+--------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
---------------------------------------------------------------------------------------------------------------------------------
-------------------------------------+
| -> Sort: Decade, gn.genreName  (actual time=36.791..36.805 rows=215 loops=1)
    -> Table scan on <temporary>  (actual time=36.638..36.694 rows=215 loops=1)
        -> Aggregate using temporary table  (actual time=36.636..36.636 rows=215 loops=1)
            -> Nested loop inner join  (cost=3529.53 rows=11627) (actual time=0.409..22.452 rows=13297 loops=1)
                -> Table scan on it  (cost=539.35 rows=5151) (actual time=0.390..2.468 rows=5000 loops=1)
                -> Covering index lookup on gn using PRIMARY (titleID=it.titleID)  (cost=0.35 rows=2) (actual time=0.003..0.004 rows=3 loops=5000)
 |
```

For each indexing design used, the costs for the nested loop inner join did not change. There weren't any noticeable changes in the total actual time as well. This is most likely because the attributes involved in the relevant operations already had indices associated with them since

they are primary keys. So the indexing design we chose to optimize this query's performance is to just stick with the default indexing.

Indexing design on query (2):
Default Indexing EXPLAIN ANALYZE

```
| -> Nested loop inner join  (cost=4192436.01 rows=17845199) (actual time=6.887..17.629 rows=756 loops=1)
    -> Nested loop inner join  (cost=2392786.80 rows=17845199) (actual time=6.875..15.738 rows=756 loops=1)
        -> Nested loop inner join  (cost=606142.43 rows=6050880) (actual time=6.831..12.596 rows=565 loops=1)
            -> Filter: (iMDB_Titles.primaryTitle is not null)  (cost=539.35 rows=5151) (actual time=0.058..2.315 rows=5000 loops=1)
                -> Table scan on iMDB_Titles  (cost=539.35 rows=5151) (actual time=0.057..1.930 rows=5000 loops=1)
            -> Single-row index lookup on <subquery2> using <auto_distinct_key> (primaryTitle=iMDB_Titles.primaryTitle)  (actual time=0.002..0.002 rows=0 loops=5000)
                -> Materialize with deduplication  (cost=1727.57..1727.57 rows=1175) (actual time=6.760..6.760 rows=529 loops=1)
                    -> Filter: (iMDB_Titles.primaryTitle is not null)  (cost=1610.10 rows=1175) (actual time=0.063..6.464 rows=552 loops=1)
                        -> Nested loop inner join  (cost=1610.10 rows=1175) (actual time=0.063..6.406 rows=552 loops=1)
                            -> Filter: (Genres.genreName = 'Horror')  (cost=1198.95 rows=1175) (actual time=0.045..5.041 rows=552 loops=1)
                                -> Covering index scan on Genres using PRIMARY  (cost=1198.95 rows=11747) (actual time=0.038..3.771 rows=13297 loops=1)
                            -> Single-row index lookup on iMDB_Titles using PRIMARY (titleID=Genres.titleID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=552)
        -> Covering index lookup on KnownFor using PRIMARY (titleID=iMDB_Titles.titleID)  (cost=0.41 rows=3) (actual time=0.005..0.005 rows=1 loops=565)
    -> Single-row index lookup on iMDB_Names using PRIMARY (nameID=KnownFor.nameID)  (cost=1.00 rows=1) (actual time=0.002..0.002 rows=1 loops=756)
|
```

(1)

```
mysql> CREATE INDEX genreName_idx ON Genres(genreName);
```

```
| -> Nested loop inner join  (cost=454495.82 rows=838559) (actual time=2.292..10.100 rows=756 loops=1)
    -> Nested loop inner join  (cost=369127.01 rows=838559) (actual time=2.281..8.311 rows=756 loops=1)
        -> Filter: (iMDB_Titles.primaryTitle = `<subquery2>`.primaryTitle)  (cost=285058.68 rows=284335) (actual time=2.259..5.279 rows=565 loops=1)
            -> Inner hash join (<hash>(iMDB_Titles.primaryTitle)=<hash>(`<subquery2>`.primaryTitle))  (cost=285058.68 rows=284335) (actual time=2.257..5.072 rows=565 loops=1)
                -> Table scan on iMDB_Titles  (cost=284.61 rows=5151) (actual time=0.027..1.804 rows=5000 loops=1)
                -> Hash
                    -> Table scan on <subquery2>  (cost=480.99..490.38 rows=552) (actual time=1.968..2.039 rows=529 loops=1)
                        -> Materialize with deduplication  (cost=480.98..480.98 rows=552) (actual time=1.964..1.964 rows=529 loops=1)
                            -> Filter: (iMDB_Titles.primaryTitle is not null)  (cost=425.78 rows=552) (actual time=0.075..1.701 rows=552 loops=1)
                                -> Nested loop inner join  (cost=425.78 rows=552) (actual time=0.074..1.648 rows=552 loops=1)
                                    -> Covering index lookup on Genres using genreName_idx (genreName='Horror')  (cost=232.58 rows=552) (actual time=0.056..0.271 rows=552 loops=1)
                                    -> Single-row index lookup on iMDB_Titles using PRIMARY (titleID=Genres.titleID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=552)
        -> Covering index lookup on KnownFor using PRIMARY (titleID=iMDB_Titles.titleID)  (cost=0.41 rows=3) (actual time=0.005..0.005 rows=1 loops=565)
    -> Single-row index lookup on iMDB_Names using PRIMARY (nameID=KnownFor.nameID)  (cost=1.00 rows=1) (actual time=0.002..0.002 rows=1 loops=756)
|
```

(2)

```
mysql> CREATE INDEX genre_idx ON Genres(genreName,titleID);
```

```
| -> Nested loop inner join  (cost=2600.34 rows=1628) (actual time=0.136..6.432 rows=724 loops=1)
    -> Nested loop inner join  (cost=816.24 rows=1628) (actual time=0.089..4.619 rows=724 loops=1)
        -> Nested loop inner join  (cost=425.78 rows=552) (actual time=0.059..1.713 rows=552 loops=1)
            -> Covering index lookup on Genres using genre_idx (genreName='Horror')  (cost=232.58 rows=552) (actual time=0.043..0.257 rows=552 loops=1)
            -> Single-row index lookup on iMDB_Titles using PRIMARY (titleID=Genres.titleID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=552)
        -> Covering index lookup on KnownFor using PRIMARY (titleID=Genres.titleID)  (cost=0.41 rows=3) (actual time=0.005..0.005 rows=1 loops=552)
    -> Single-row index lookup on iMDB_Names using PRIMARY (nameID=KnownFor.nameID)  (cost=1.00 rows=1) (actual time=0.002..0.002 rows=1 loops=724)
|
```

(3)

```
mysql> CREATE INDEX genre_idx ON Genres(genreName,titleID);
```

AND

```
mysql> CREATE INDEX title_idx ON iMDBTitles(primaryTitle);
```

```
| -> Nested loop inner join  (cost=2600.34 rows=1628) (actual time=0.084..6.337 rows=724 loops=1)
    -> Nested loop inner join  (cost=816.24 rows=1628) (actual time=0.076..4.635 rows=724 loops=1)
        -> Nested loop inner join  (cost=425.78 rows=552) (actual time=0.054..1.653 rows=552 loops=1)
            -> Covering index lookup on Genres using genre_idx (genreName='Horror')  (cost=232.58 rows=552) (actual time=0.038..0.262 rows=552 loops=1)
            -> Single-row index lookup on iMDB_Titles using PRIMARY (titleID=Genres.titleID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=552)
        -> Covering index lookup on KnownFor using PRIMARY (titleID=Genres.titleID)  (cost=0.41 rows=3) (actual time=0.005..0.005 rows=1 loops=552)
    -> Single-row index lookup on iMDB_Names using PRIMARY (nameID=KnownFor.nameID)  (cost=1.00 rows=1) (actual time=0.002..0.002 rows=1 loops=724)
|
```

As we can see from the indexing analysis the indexing design that yielded the lowest cost was (2). We saw an improvement in cost by just indexing the genreName, but by making a multiple-column index on genreName and titleID we were able to see a major improvement. (3) yielded the same cost as (2), so we decided to stick with indexing design (2) to optimize this query's performance.