DDL Commands

```
CREATE TABLE User (
        UserId INT PRIMARY KEY,
        Password INT,
        FirstName VARCHAR(255),
        LastName VARCHAR(255),
        Age INT,
        Email VARCHAR(255),
        PhoneNumber INT
);

CREATE TABLE Rating (
        UserId INT PRIMARY KEY,
        SongId INT PRIMARY KEY,
        Rating INT,
        FOREIGN KEY(UserId) REFERENCES User(UserId)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
        FOREIGN KEY(SongId) REFERENCES Song(SongId)
        ON DELETE SET NULL
        ON UPDATE CASCADE
);

CREATE TABLE Playlist (
        PlaylistId INT PRIMARY KEY,
        PlaylistName VARCHAR(255),
        CreationDate DATE
);


CREATE TABLE Song (
        SongId INT PRIMARY KEY,
        SongName VARCHAR(255),
        ReleaseDate DATE,
        Duration FLOAT,
        Popularity INT,
        Explicit INT,
        Mode INT,
```

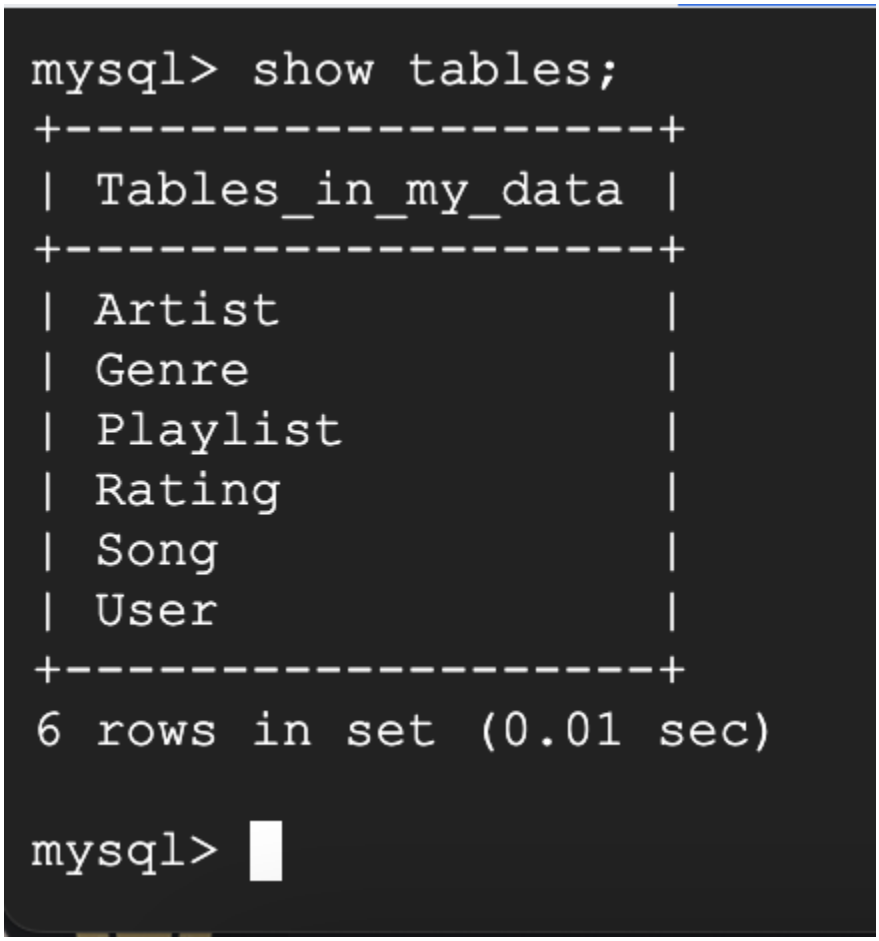
```
        Energy FLOAT,
        Liveliness FLOAT,
        ArtistID INT,
        FOREIGN KEY(ArtistID) REFERENCES Artist(ArtistID)
        ON DELETE SET NULL
        ON UPDATE CASCADE

);


CREATE TABLE Artist (
        ArtistID INT PRIMARY KEY,
        ArtistName VARCHAR(255)
);


CREATE TABLE Genre (
        GenreName VARCHAR(255) PRIMARY KEY,
        Danceability FLOAT,
        Popularity INT,
        Energy FLOAT,
        Tempo FLOAT,
        ArtistID INT,
        SongID INT,
);
```

DDL COMMAND GCP CONNECTION



Screenshots for 1000>=Rows Tables

```
mysql> select count(*) from User
    -> ;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql>
```

```
mysql> select count(*) from Artist
    -> ;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.00 sec)

mysql>
```

```
mysql> select count(*) from Song
    -> ;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
1 row in set (0.01 sec)

mysql>
```

Advanced Query 1

Impression on Categories Query: Retrieve top 15 popular artists based on their songs' popularity. In our application popularity of an artist is attained by the getting the SUM of the popularity of each of their songs.

Tables used: Artist, Song

SQL Concepts Used: Join of multiple relations, Aggregation via GROUP BY

Query:

```
SELECT ArtistName, Sum(Popularity) AS total_pop
FROM Artist JOIN Song ON Artist.ArtistID = Song.ArtistID
GROUP BY ArtistName
ORDER BY total_pop DESC
LIMIT 15;
```

```
mysql> SELECT ArtistName, Sum(Popularity) AS total_pop
    -> FROM Artist JOIN Song ON Artist.ArtistID = Song.ArtistID
    -> GROUP BY ArtistName
    -> ORDER BY total_pop DESC
    -> LIMIT 15;
+------------------+-----------+
| ArtistName       | total_pop |
+------------------+-----------+
| Rigel Lewis      |        34 |
| Fleur Waters     |        33 |
| Leonard Roach    |        32 |
| Kameko Murray    |        30 |
| Melanie Meyers   |        26 |
| Callum Andrews   |        24 |
| Laith George     |        23 |
| Thaddeus Perkins |        23 |
| Wendy Rush       |        22 |
| Hiram Boone      |        22 |
| Melanie Mccarty  |        22 |
| Vera Stanley     |        22 |
| Kaye Santana     |        22 |
| Angela Combs     |        22 |
| Quinlan Ortiz    |        21 |
+------------------+-----------+
15 rows in set (0.01 sec)
```

Advanced Query 2

Impression on Categories Query: Retrieve popular songs with less explicitly (<5).

Tables used: Artist, Song

SQL Concepts Used: Join of multiple relations, Aggregation via GROUP BY

Query:
SELECT SongName, MAX(Popularity) AS popularity, Explicit
FROM Artist JOIN Song ON Artist.ArtistID = Song.ArtistID
WHERE Explicit < 5
GROUP BY SongName, Explicit
ORDER BY popularity DESC
LIMIT 15;

```
mysql> SELECT SongName, MAX(Popularity) AS popularity, Explicit
    -> FROM Artist JOIN Song ON Artist.ArtistID = Song.ArtistID
    -> WHERE Explicit < 5
    -> GROUP BY SongName, Explicit
    -> ORDER BY popularity DESC
    -> LIMIT 15;
+-----------------------+------------+----------+
| SongName              | popularity | Explicit |
+-----------------------+------------+----------+
| neque non quam.       |         10 |        4 |
| turpis. Nulla aliquet. |         10 |        4 |
| lectus pede, ultrices  |         10 |        3 |
| convallis convallis    |         10 |        3 |
| odio. Phasellus at     |         10 |        2 |
| libero. Proin          |         10 |        4 |
| ornare, libero at      |         10 |        1 |
| congue turpis. In      |         10 |        3 |
| dictum. Proin eget     |         10 |        2 |
| nec, mollis vitae,     |         10 |        4 |
| egestas a, dui.        |         10 |        3 |
| non massa non          |         10 |        2 |
| eu enim.               |         10 |        2 |
| nibh. Quisque          |         10 |        3 |
| lacus. Etiam           |         10 |        2 |
+-----------------------+------------+----------+
15 rows in set (0.00 sec)
```

Indexing

Advanced Query 1

This is our initial state when we run EXPLAIN ANALYZE.

```
| EXPLAIN
                                                                                                                        |
+----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=3.222..3.224 rows=15 loops=1)
    -> Sort: total_pop DESC, limit input to 15 row(s) per chunk  (actual time=3.221..3.222 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.961..3.062 rows=633 loops=1)
            -> Aggregate using temporary table  (actual time=2.960..2.960 rows=633 loops=1)
                -> Nested loop inner join  (cost=451.75 rows=1000) (actual time=0.092..1.876 rows=1000 loops=1)
                    -> Filter: (Song.ArtistID is not null)  (cost=101.75 rows=1000) (actual time=0.073..0.543 rows=1000 loops=1)
                        -> Table scan on Song  (cost=101.75 rows=1000) (actual time=0.071..0.453 rows=1000 loops=1)
                    -> Single-row index lookup on Artist using PRIMARY (ArtistID=Song.ArtistID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
+----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Initial Cost: 101.75, Initial Time Taken: 0.073 ~ 0.543

DESIGN 1: Index by ArtistName

We indexed ArtistName in the Artist table because it was used to group each artist based on their ArtistName to find the sum of popularity for each artist.

CREATE INDEX idx_artist ON Artist (ArtistName);

This is our result after indexing.

```
| EXPLAIN
                                                                                                                                          |
+------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=3.024..3.026 rows=15 loops=1)
    -> Sort: total_pop DESC, limit input to 15 row(s) per chunk  (actual time=3.023..3.024 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=2.799..2.884 rows=633 loops=1)
            -> Aggregate using temporary table  (actual time=2.796..2.796 rows=633 loops=1)
                -> Nested loop inner join  (cost=451.75 rows=1000) (actual time=0.071..1.757 rows=1000 loops=1)
                    -> Filter: (Song.ArtistID is not null)  (cost=101.75 rows=1000) (actual time=0.057..0.483 rows=1000 loops=1)
                        -> Table scan on Song  (cost=101.75 rows=1000) (actual time=0.055..0.398 rows=1000 loops=1)
                    -> Single-row index lookup on Artist using PRIMARY (ArtistID=Song.ArtistID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
+------------------------------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Result cost: 101.75, Result Time Taken: 0.055 ~ 0.398

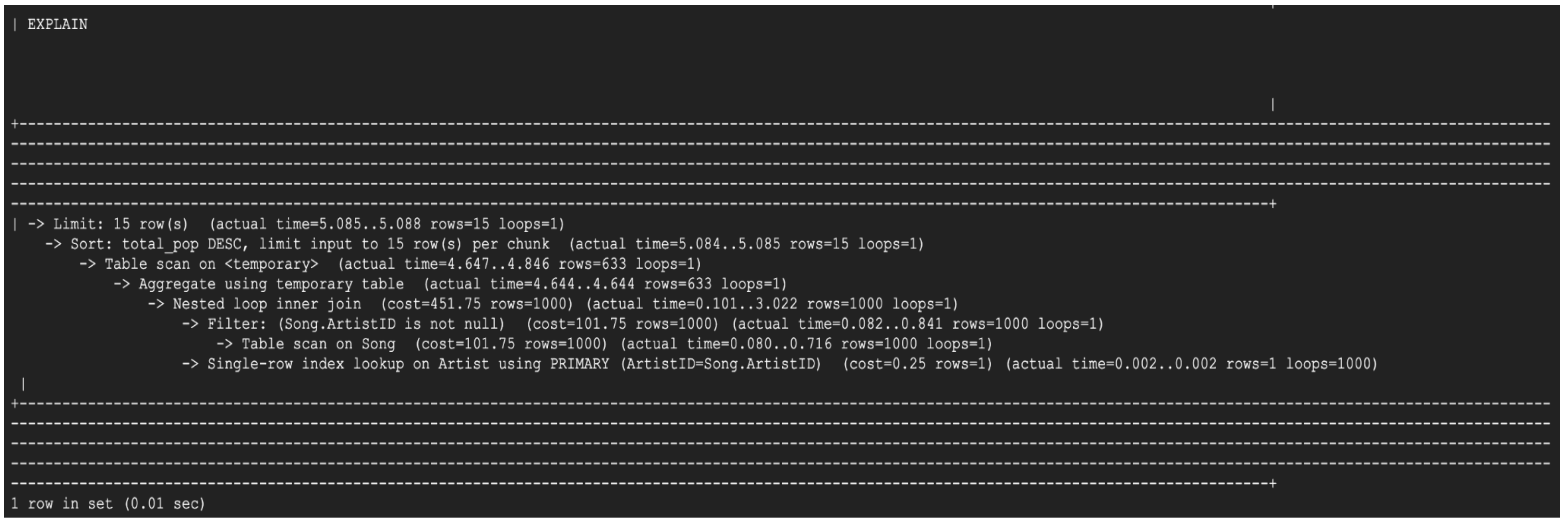
It slightly reduced the actual time by about 0.02 ~ 0.145 but it didn't change the cost. This is because indexing ArtistName helps to optimize the query since it reduces time when we iterate through ArtistName to find the sum of each artist. However, it is not a big change in time taken.

DESIGN 2: Index by Popularity

We indexed Popularity in the Popularity table because it was used in a sum aggregate function to determine the artist with the highest popularity in sum of his or her songs.

CREATE INDEX idx_pop ON Song(Popularity);

This is our result after indexing.



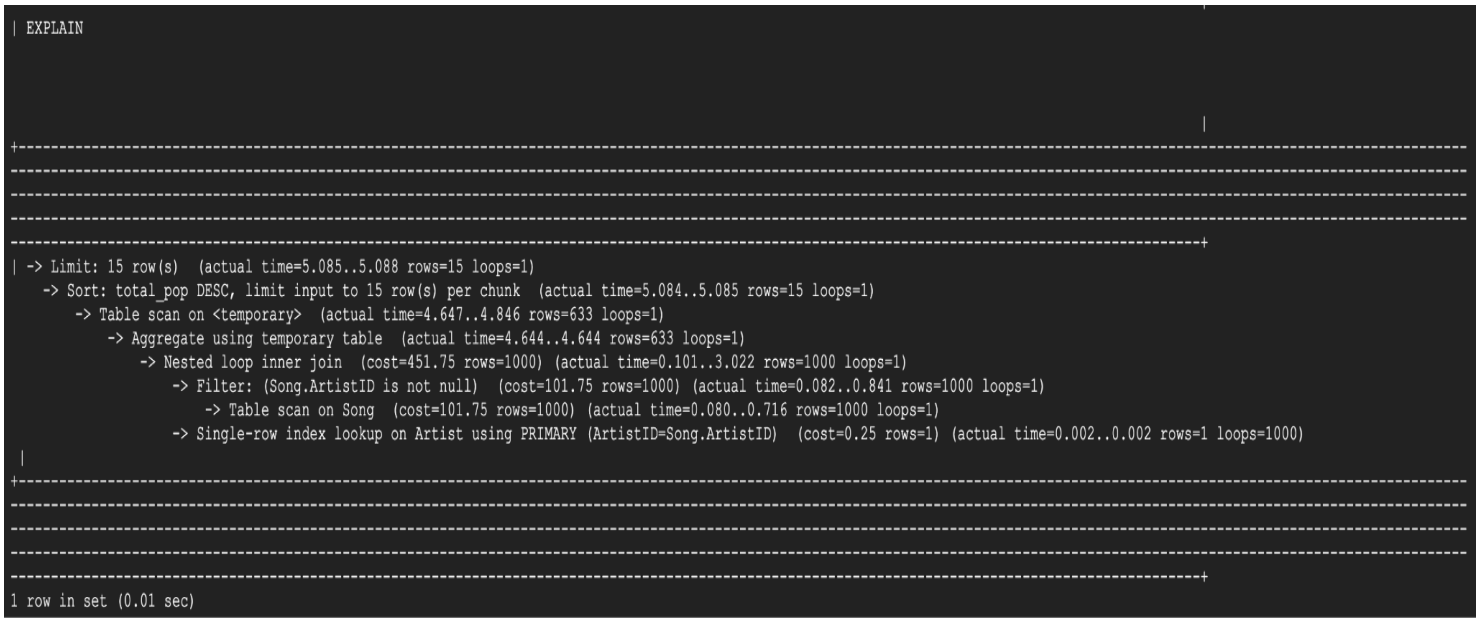
Result cost: 101.75, Result Time Taken: 0.080 ~ 0.716

Resulting cost and time taken doesn’t seem to be improved by indexing popularity. We think it is because the sum of popularity is sorted in ORDER BY, so indexing doesn’t affect the cost and time.

DESIGN 3: Index by both ArtistName and Popularity

We indexed both ArtistName and Popularity to see if there is an improvement in cost and time taken.

CREATE INDEX idx_artist ON Artist(ArtistName);
CREATE INDEX idx_pop ON Song(Popularity);



Result cost: 101.75, Result Time Taken: 0.080 ~ 0.716

As we indexed for ArtistName and Popularity separately and got no major improvement in cost and time taken, indexing both ArtistName and Popularity doesn’t affect the cost and time taken.

Advanced Query 2

This is our initial state when we run EXPLAIN ANALYZE.

```
| EXPLAIN
| -> Limit: 15 row(s)  (actual time=3.707..3.709 rows=15 loops=1)
    -> Sort: total_pop DESC, limit input to 15 row(s) per chunk  (actual time=3.706..3.707 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=3.474..3.568 rows=633 loops=1)
            -> Aggregate using temporary table  (actual time=3.472..3.472 rows=633 loops=1)
                -> Nested loop inner join  (cost=451.75 rows=1000) (actual time=0.056..2.092 rows=1000 loops=1)
                    -> Filter: (Song.ArtistID is not null)  (cost=101.75 rows=1000) (actual time=0.046..0.564 rows=1000 loops=1)
                        -> Table scan on Song  (cost=101.75 rows=1000) (actual time=0.045..0.465 rows=1000 loops=1)
                    -> Single-row index lookup on Artist using PRIMARY (ArtistID=Song.ArtistID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1000)
 |
1 row in set (0.01 sec)
```

Initial Cost: 101.75, Initial Time Taken: 0.045 ~ 0.465

DESIGN 1: Index by SongName

We indexed SongName in the Song table because it was used to group each song based on their SongName to find the maximum popularity among less explicit songs in the Song table.

CREATE INDEX idx_songname ON Song(SongName);

```
| EXPLAIN |
+-------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=2.083..2.096 rows=15 loops=1)
    -> Sort: popularity DESC, limit input to 15 row(s) per chunk  (actual time=2.081..2.093 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=1.921..1.987 rows=354 loops=1)
            -> Aggregate using temporary table  (actual time=1.919..1.919 rows=354 loops=1)
                -> Nested loop inner join  (cost=218.40 rows=333) (actual time=0.162..1.403 rows=368 loops=1)
                    -> Filter: ((Song.Explicit < 5) and (Song.ArtistID is not null))  (cost=101.75 rows=333) (actual time=0.146..0.709 rows=368 loops=1)
                        -> Table scan on Song  (cost=101.75 rows=1000) (actual time=0.139..0.593 rows=1000 loops=1)
                    -> Single-row covering index lookup on Artist using PRIMARY (ArtistID=Song.ArtistID)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=368)
 |
+-------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

Result cost: 101.75, Result Time Taken: 0.139 ~ 0.593

Resulting cost and time taken wasn’t improved and what we think the reason for this is there is because it doesn’t iterate over songs when we try to find the less explicit songs with high popularity.

DESIGN 2: Index by Explicit

We indexed Explicit in the Song table because it was used to group each song based on their Explicit attribute to find the maximum popularity among less explicit songs in the Song table.

CREATE INDEX idx_explicit ON Song(Explicit);

```
| EXPLAIN
                                                                                                                         |
+------------------------------------------------------------------------------------------------------------------------
+
| -> Limit: 15 row(s)  (actual time=1.613..1.615 rows=15 loops=1)
    -> Sort: popularity DESC, limit input to 15 row(s) per chunk  (actual time=1.612..1.613 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=1.499..1.551 rows=354 loops=1)
            -> Aggregate using temporary table  (actual time=1.498..1.498 rows=354 loops=1)
                -> Nested loop inner join  (cost=230.55 rows=368) (actual time=0.079..1.107 rows=368 loops=1)
                    -> Filter: ((Song.Explicit < 5) and (Song.ArtistID is not null))  (cost=101.75 rows=368) (actual time=0.064..0.555 rows=368 loops=1)
                        -> Table scan on Song  (cost=101.75 rows=1000) (actual time=0.058..0.441 rows=1000 loops=1)
                    -> Single-row covering index lookup on Artist using PRIMARY (ArtistID=Song.ArtistID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=368)
 |
+------------------------------------------------------------------------------------------------------------------------
+
1 row in set (0.00 sec)
```

Result cost: 101.75, Result Time Taken: 0.058 ~ 0.441

There seems no improvement in cost and time taken and as we said for design 1, we think this is occurring because it doesn’t iterate over explicit when we find less explicit songs, but just compare if explicit is less than 5.

DESIGN 3: Index by Popularity

We indexed Popularity in the Popularity table because it was used in a max aggregate function to determine the song with the highest popularity among all the less explicit songs.

CREATE INDEX idx_pop ON Song(Popularity);

```
| EXPLAIN
                                                                                                                                |
+-------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (actual time=1.758..1.760 rows=15 loops=1)
    -> Sort: popularity DESC, limit input to 15 row(s) per chunk  (actual time=1.757..1.758 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=1.640..1.691 rows=354 loops=1)
            -> Aggregate using temporary table  (actual time=1.638..1.638 rows=354 loops=1)
                -> Nested loop inner join  (cost=218.40 rows=333) (actual time=0.094..1.231 rows=368 loops=1)
                    -> Filter: ((Song.Explicit < 5) and (Song.ArtistID is not null))  (cost=101.75 rows=333) (actual time=0.078..0.646 rows=368 loops=1)
                        -> Table scan on Song  (cost=101.75 rows=1000) (actual time=0.072..0.540 rows=1000 loops=1)
                    -> Single-row covering index lookup on Artist using PRIMARY (ArtistID=Song.ArtistID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=368)
 |
+-------------------------------------------------------------------------------------------------------------------------------+
1 row in set (0.00 sec)
```

Result cost: 101.75, Result Time Taken: 0.072 ~ 0.540

This index also didn’t improve the cost and the time taken, and we think it’s because we use the maximum of this attribute in the ORDER BY clause which doesn’t get influenced by indexing.