Inserting Data

We had two tables called temp and temp2 where we inserted 2 dataset into it. We then created another table where we aggregate all the relevant data from temp and temp2 and named it BIGTABLE. We then created a trigger where before inserting into BIGTABLE we create a unique artistID for each artist by checking if the artist already has an ID and if not finding the largest ID already made and adding 1. We then created tables for SongRecommendations, Favorites, Account, Song, and Artist. We inserted our data into Song and Artist from BIGTABLE. We then inserted our own names into Account, and wrote a procedure to generate 1000 favorite songs for one of our accounts. That leaves us with ~5000 items in Song, ~1200 items in Artist, and ~1000 items in Favorites. We also wrote triggers that assigns favoriteID to each favorite, and updates the favoriteCount in the account data.

```
mysql> show tables
    -> ;
+--------------------------+
| Tables_in_SpotifyNotMil  |
+--------------------------+
| Account                  |
| Artist                   |
| BIGTABLE                 |
| Favorites                |
| Song                     |
| SongRecommendations      |
| temp                     |
| temp2                    |
+--------------------------+
8 rows in set (0.00 sec)

mysql>
```

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

mysql>
```

```
mysql> SELECT COUNT(artistID) FROM Artist;
+------------------+
| COUNT(artistID)  |
+------------------+
|             1194 |
+------------------+
1 row in set (0.01 sec)

mysql>
```

```
mysql> SELECT COUNT(favoriteID) FROM Favorites;
+--------------------+
| COUNT(favoriteID)  |
+--------------------+
|               1000 |
+--------------------+
1 row in set (0.01 sec)

mysql>
```

Advanced Queries

Query 1

```
mysql> SELECT Artist.name, AVG(Song.songDuration) FROM Artist JOIN Song on Artist.artistID = Song.artistID GROUP BY Artist.name ORDER BY AVG(Song.songDuration) DESC LIMIT 15;
+-----------------------------------------------------------------+-------------------------+
| name                                                            | AVG(Song.songDuration)  |
+-----------------------------------------------------------------+-------------------------+
| Fela Kuti                                                       |               944.1667  |
| Fela Kuti;Afrika 70                                             |               793.0000  |
| Tony Allen With Africa 70                                       |               684.0000  |
| Peace                                                           |               608.0000  |
| John Elliott                                                    |               578.0000  |
| Ebo Taylor;Pat Thomas;Henrik Schwarz                            |               564.0000  |
| Fanga                                                           |               550.5714  |
| Akoya Afrobeat Ensemble                                         |               540.0000  |
| Hillsong UNITED;TAYA                                            |               536.0000  |
| Ebo Taylor;Uhuru-Yenzu                                          |               500.0000  |
| Foli Grió Orquestra;Carlos Malta;Dofono de Omulu;Ekedi Nicinha   |               495.0000  |
| Kaleta & Zozo Afrobeat                                          |               492.5000  |
| Euforquestra                                                    |               491.8889  |
| Cultura Profética                                               |               486.2500  |
| Nomo                                                            |               483.0000  |
+-----------------------------------------------------------------+-------------------------+
15 rows in set (0.01 sec)

mysql>
```

Query 2

```
mysql> SELECT DISTINCT s.genre
    ->         FROM Song s JOIN Favorites f ON(s.spotifyTrackID = f.spotifyTrackID)
    ->         WHERE (SELECT COUNT(favoriteID)
    ->                FROM Song s1 JOIN Favorites f1 ON(s1.spotifyTrackID = f1.spotifyTrackID)
    ->                WHERE f1.userID LIKE "jbgauer" AND s1.genre = s.genre
    ->                GROUP BY s1.genre)
    ->               =
    ->               (SELECT MAX(genreCount)
    ->                FROM (SELECT COUNT(f2.favoriteID) AS genreCount
    ->                FROM Song s2 JOIN Favorites f2 ON(s2.spotifyTrackID = f2.spotifyTrackID)
    ->                WHERE f2.userID LIKE "jbgauer" GROUP BY s2.genre) AS temp)
    ->         LIMIT 1;
+-----------+
| genre     |
+-----------+
| alt-rock  |
+-----------+
1 row in set (0.01 sec)
```

This only returns the users favorite genre, which is found through parsing a users favorite song list and finding the genre that is most prevalent.

Query 3

```
mysql> SELECT s.songName, a.name as AristName
    -> FROM Song s JOIN Artist a ON s.artistID = a.artistID
    -> WHERE a.name = (SELECT a1.name
    -> FROM Favorites f1 JOIN Song s1 ON f1.spotifyTrackID = s1.spotifyTrackID JOIN Artist a1 ON s1.artistID = a1.artistID
    -> WHERE f1.userID = "jbgauer" AND f1.spotifyTrackID != s.spotifyTrackID
    -> ORDER BY RAND()
    -> LIMIT 1)
    -> LIMIT 1;
+----------+-----------+
| songName | AristName |
+----------+-----------+
| Stronger | The Score |
+----------+-----------+
1 row in set (0.69 sec)

mysql>
```

This query grabs a single song recommendation for the user. This is another song from an artist that was found in the users list of favorite songs that were not already in their favorites.

Query 4

```
mysql> SELECT
    ->     A.name AS ArtistName,
    ->     S.songName AS MostPopularSong,
    ->     S.popularity AS Popularity
    -> FROM
    ->     Artist A
    -> JOIN
    ->     Song S ON A.artistID = S.artistID
    -> WHERE
    ->     S.popularity = (
    ->         SELECT
    ->             MAX(S2.popularity)
    ->         FROM
    ->             Song S2
    ->         WHERE
    ->             S2.artistID = A.artistID
    ->     )
    -> LIMIT 15;
+-------------------------------------------------------
```

ArtistName	MostPopularSong	Popularity
Jason Mraz	I'm Yours	80
Eddie Vedder	Society	68
Toto Sorioso	Saving Forever For You	28
Niña Dioz;Shigeto	Nubes (feat. Shigeto)	19
Geraldo Pino;The Heartbeats	Black Woman Experience - Mixed	17
Tony Allen;Ty	Woman To Man	17
El Rego;Kill Emil	E Nan Mian Nuku	17
BaianaSystem;Buguinha Dub;Antonio Carlos & Jocafi;Orquestra Afrosinfônica	Água (Adubada)	18
Nomo	Discontinued	16
A Mose	Faraway	16
Assagai	Telephone Girl	17
Voilaaa;Rama Traore	Tu Mens Devant Moi	17
Voilaaa;Lass;JKriv	Ku La Foon - JKriv Remix	16
Lulu Panganiban	I'll Never Get Over You Getting Over Me	30
Ebo Taylor;Pat Thomas;Henrik Schwarz	Eye Nyam Nam 'A' Mensuro - Henrik Schwarz Blend	49

Find the most popular song by each artist
Join multiple relations: The query joins the Artist table with the Song table on the artistID field to fetch information about the artist and their songs.

Subqueries that cannot be easily replaced by a join: The query includes a subquery to find the maximum popularity for each artist's songs, which is used in the WHERE clause to filter the most popular song for each artist.

Indexing

Query 1

No Indexing Added

```
| -> Limit: 15 row(s)  (actual time=34.743..34.746 rows=15 loops=1)
    -> Sort: AVG(Song.songDuration) DESC, limit input to 15 row(s) per chunk  (actual time=34.741..34.743 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=33.797..34.162 rows=1193 loops=1)
            -> Aggregate using temporary table  (actual time=33.792..33.792 rows=1193 loops=1)
                -> Nested loop inner join  (cost=1814.08 rows=4838) (actual time=1.307..23.782 rows=4999 loops=1)
                    -> Table scan on Artist  (cost=120.80 rows=1193) (actual time=0.419..1.679 rows=1193 loops=1)
                    -> Index lookup on Song using artistID (artistID=Artist.artistID)  (cost=1.01 rows=4) (actual time=0.011..0.018 rows=4 loops=1193)
 |
```

The initial analysis resulted in a cost of 1814.08 for the nested inner join, a cost of 120.80 for the scan on the Artist table, and a cost of 1.01 for the index lookup on the Song table using artistID.

Index Created for Song.artistID

```
mysql> CREATE INDEX song_a ON Song(artistID);
Query OK, 0 rows affected (0.58 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (actual time=28.198..28.200 rows=15 loops=1)
    -> Sort: AVG(Song.songDuration) DESC, limit input to 15 row(s) per chunk  (actual time=28.197..28.199 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=27.478..27.691 rows=1193 loops=1)
            -> Aggregate using temporary table  (actual time=27.475..27.475 rows=1193 loops=1)
                -> Nested loop inner join  (cost=1815.50 rows=4842) (actual time=2.448..17.060 rows=4999 loops=1)
                    -> Table scan on Artist  (cost=120.80 rows=1193) (actual time=0.345..2.509 rows=1193 loops=1)
                    -> Index lookup on Song using song_a (artistID=Artist.artistID)  (cost=1.02 rows=4) (actual time=0.008..0.012 rows=4 loops=1193)
 |
```

The first index I tested was for artistID in the Song table. This resulted in a cost of 1815.5 for the nested inner join, a cost of 120.80 for the scan on the Artist table, and a cost of 1.02 for the

index lookup on the Song table using artistID. The result is that the cost went up by 1.42 for the inner join and 0.01 for the index lookup on the Song table compared to the initial run.

Index Created for Song.songDuration

```
mysql> CREATE INDEX songDuration_idx ON Song(songDuration);
Query OK, 0 rows affected (0.20 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (actual time=25.634..25.637 rows=15 loops=1)
    -> Sort: AVG(Song.songDuration) DESC, limit input to 15 row(s) per chunk  (actual time=25.634..25.635 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=25.003..25.204 rows=1193 loops=1)
            -> Aggregate using temporary table  (actual time=25.002..25.002 rows=1193 loops=1)
                -> Nested loop inner join  (cost=1858.90 rows=4966) (actual time=0.377..16.766 rows=4999 loops=1)
                    -> Table scan on Artist  (cost=120.80 rows=1193) (actual time=0.140..0.857 rows=1193 loops=1)
                    -> Index lookup on Song using Song_ibfk_1 (artistID=Artist.artistID)  (cost=1.04 rows=4) (actual time=0.009..0.013 rows=4 loops=1193)
|
```

The next index I tested was for songDuration in the Song table. This resulted in a cost of 1858.9 for the nested inner join, a cost of 120.80 for the scan on the Artist table, and a cost of 1.04 for the index lookup on the Song table using artistID. The result is that the cost went up by 44.82 for the inner join and 0.02 for the index lookup on the Song table compared to the initial run.

Index Created for Artist.name

```
mysql> CREATE INDEX artistName_idx ON Artist(name);
Query OK, 0 rows affected (0.14 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

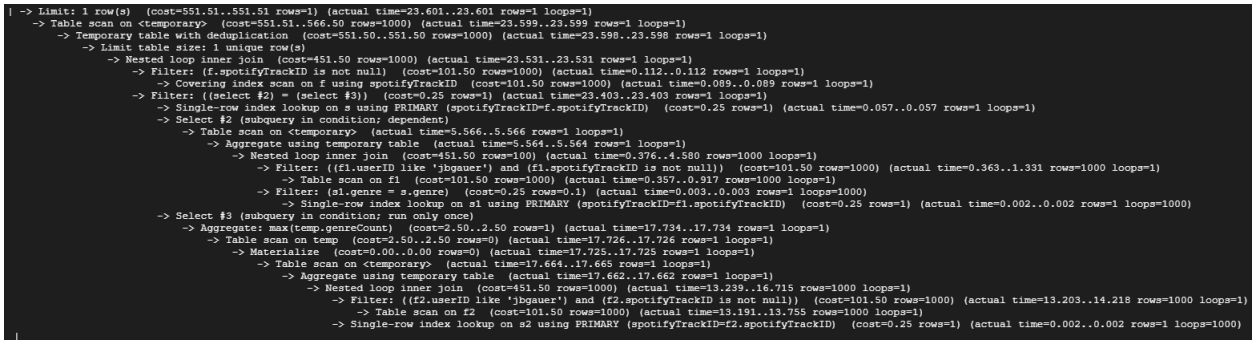
```
| -> Limit: 15 row(s)  (actual time=13.461..13.463 rows=15 loops=1)
    -> Sort: AVG(Song.songDuration) DESC, limit input to 15 row(s) per chunk  (actual time=13.461..13.462 rows=15 loops=1)
        -> Stream results  (cost=2355.50 rows=4966) (actual time=0.171..13.146 rows=1193 loops=1)
            -> Group aggregate: avg(Song.songDuration)  (cost=2355.50 rows=4966) (actual time=0.139..12.536 rows=1193 loops=1)
                -> Nested loop inner join  (cost=1858.90 rows=4966) (actual time=0.122..11.100 rows=4999 loops=1)
                    -> Covering index scan on Artist using artistName_idx  (cost=120.80 rows=1193) (actual time=0.045..0.400 rows=1193 loops=1)
                    -> Index lookup on Song using Song_ibfk_1 (artistID=Artist.artistID)  (cost=1.04 rows=4) (actual time=0.005..0.009 rows=4 loops=1193)
|
```

The last index I tested was for name in the Artist table. This resulted in a cost of 1858.9 for the nested inner join, a cost of 120.80 for the scan on the Artist table, and a cost of 1.04 for the index lookup on the Song table using artistID. The result is that the cost went up by 44.82 for the inner join and 0.02 for the index lookup on the Song table compared to the initial run.

Overall, it seems that the base analysis with no index created resulted in the lowest cost for the query. Adding an index for the artistID did increase the costs very slightly, where as creating an index for the songDuration and artist name increased the cost by a lot when it came to the join part of the query.

Query 2

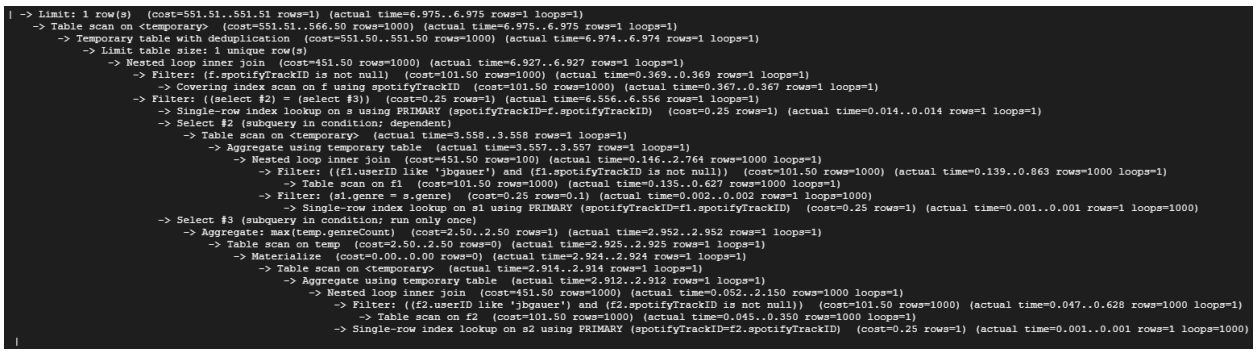
NO INDEXING ADDED (Cost 551.51)



INDEX CREATED

```
mysql> CREATE INDEX test_song_genre ON Song(genre);
Query OK, 0 rows affected (0.31 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Cost 551.51 (SAME)

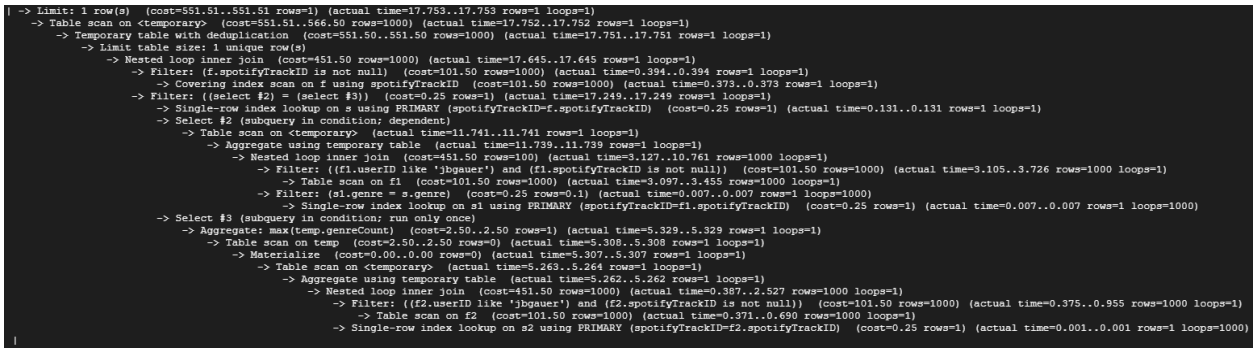


I think this didn't change because it only checked each one to see whether the genre was the same, instead of looking for something of a certain genre or otherwise. I would have expected this to knock some of the cost, but I guess according to the explanation, the cost was already only 0.25 for that.

INDEX CREATED

```
mysql> CREATE INDEX test_favorites_userID ON Favorites(userID);
Query OK, 0 rows affected (0.50 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

Cost 551.51 (SAME)

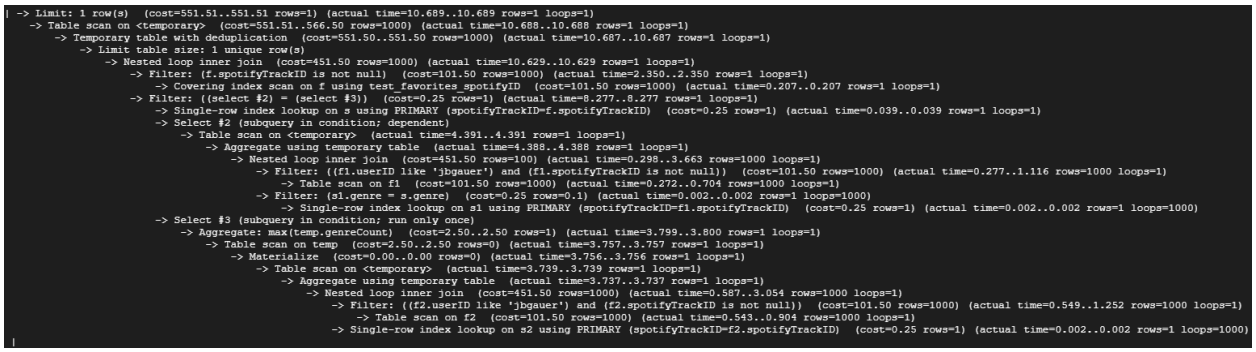


Since this is a foreign key it probably shouldn't affect it too much, since it can access that information quickly, so it makes sense why the cost didn't change.

INDEX CREATED

```
mysql> CREATE INDEX test_favorites_spotifyID ON Favorites(spotifyTrackID);
Query OK, 0 rows affected (0.22 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

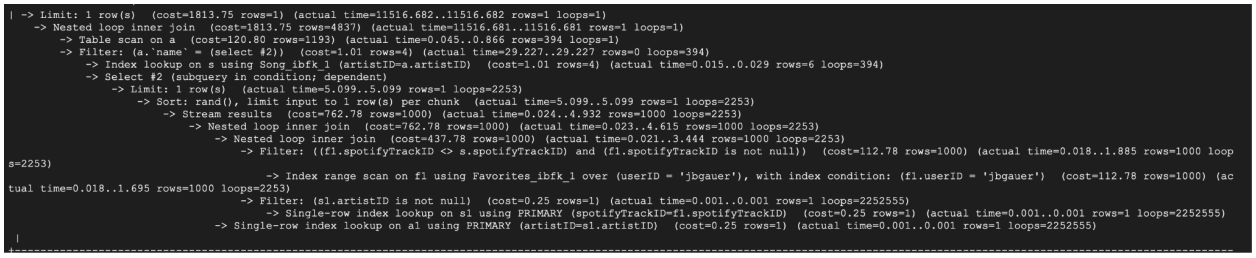
Cost 551.51 (SAME)



Since this is a foreign key it probably shouldn't affect it too much, since it can access that information quickly, so it makes sense why the cost didn't change. The reason I used foreign keys for these tests is because there were no other attributes other than genre that aren't keys that I could use to index.

Query 3

NO INDEXING ADDED (Cost 1813.75)

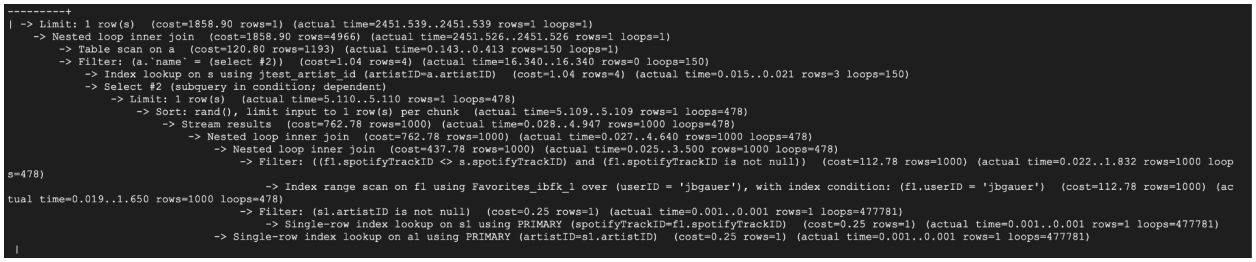


INDEX CREATED (ARTIST_ID)

```
mysql> CREATE INDEX jtest_artist_id on Song(artistID);
Query OK, 0 rows affected (0.24 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

This index was created because the query references artistID on the JOIN function.

COST (1858.90)

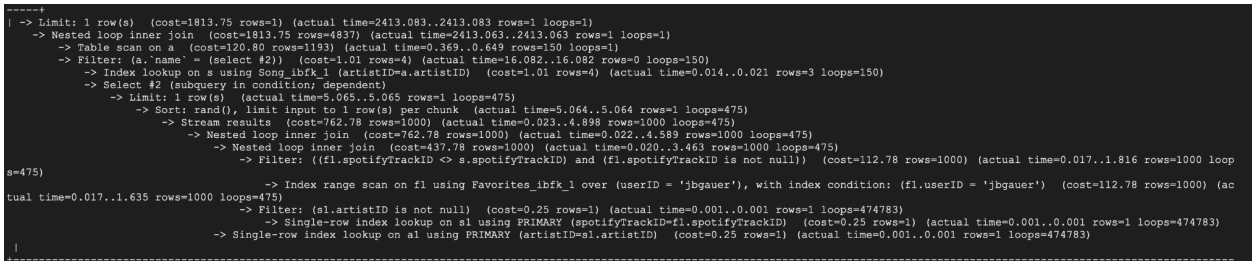


This index ended up making the cost of the query larger. You can see that the cost to filter through names went up where the index is used. This did not help the query as the original cost was 1813.75 and the cost with this index was 1858.90 so it will be removed.

INDEX CREATED

```
mysql> CREATE INDEX jtest_spot_track on Favorites(spotifyTrackID);
Query OK, 0 rows affected (0.21 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

This index was created because the query references spotifyTrackID inside of the subquery JOIN.
COST(1813.75)



This actually ended up not impacting performance at all, as the cost stayed the same at 1813.75. This is because the query was already using a index and the foreign key, and since spotifyTrackID in favorites is a foreign key it had no benefit.

INDEX CREATED

```
mysql> CREATE INDEX jtest_fav_id on Favorites(userID);
Query OK, 0 rows affected (0.24 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

This index was created because the query uses Favorites.userID in the subquery during the WHERE.
COST(1813.75)

```
-+
| -> Limit: 1 row(s)  (cost=1813.75 rows=1) (actual time=901.196..901.196 rows=1 loops=1)
    -> Nested loop inner join  (cost=1813.75 rows=4837) (actual time=901.182..901.182 rows=1 loops=1)
        -> Table scan on a  (cost=120.80 rows=1193) (actual time=0.299..0.376 rows=57 loops=1)
        -> Filter: (a.`name` = (select #2))  (cost=1.01 rows=4) (actual time=15.803..15.803 rows=0 loops=57)
            -> Index lookup on s using Song_ibfk_1 (artistID=a.artistID)  (cost=1.01 rows=4) (actual time=0.019..0.033 rows=3 loops=57)
            -> Select #2 (subquery in condition; dependent)
                -> Limit: 1 row(s)  (actual time=4.852..4.852 rows=1 loops=185)
                    -> Sort: rand(), limit input to 1 row(s) per chunk  (actual time=4.852..4.852 rows=1 loops=185)
                        -> Stream results  (cost=762.78 rows=1000) (actual time=0.014..4.698 rows=1000 loops=185)
                            -> Nested loop inner join  (cost=762.78 rows=1000) (actual time=0.014..4.394 rows=1000 loops=185)
                                -> Nested loop inner join  (cost=437.78 rows=1000) (actual time=0.011..3.286 rows=1000 loops=185)
                                    -> Filter: (f1.userID = 'jbgauer')  (cost=112.78 rows=1000) (actual time=0.008..1.630 rows=1000 loops=185)
                                        -> Index range scan on f1 using Favorites_ibfk_2 over (NULL < spotifyTrackID), with index condition: ((f1.spotifyTrackID <> s.spotifyTrackID) and (f1.
spotifyTrackID is not null))  (cost=112.78 rows=1000) (actual time=0.008..1.469 rows=1000 loops=185)
                                    -> Filter: (s1.artistID is not null)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=184965)
                                        -> Single-row index lookup on s1 using PRIMARY (spotifyTrackID=f1.spotifyTrackID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=184965)
                                -> Single-row index lookup on a1 using PRIMARY (artistID=s1.artistID)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=184965)
 |
```

This index also ended up not changing the cost from 1813.75. This is a little surprising to me, but the index ended up not being more efficient than the indexing method the query was also using so it was never grabbed.

Query 4

NO INDEXING ADDED

```
----------------------------------------------------------------------------------------------------------------------------------------------------------------+
| -> Limit: 15 row(s)  (cost=1814.08 rows=15) (actual time=5.541..13.536 rows=15 loops=1)
    -> Nested loop inner join  (cost=1814.08 rows=4838) (actual time=5.525..13.517 rows=15 loops=1)
        -> Table scan on A  (cost=120.80 rows=1193) (actual time=0.338..0.352 rows=15 loops=1)
        -> Filter: (S.popularity = (select #2))  (cost=1.01 rows=4) (actual time=0.536..0.874 rows=1 loops=15)
            -> Index lookup on S using artistID (artistID=A.artistID)  (cost=1.01 rows=4) (actual time=0.209..0.218 rows=7 loops=15)
            -> Select #2 (subquery in condition; dependent)
                -> Aggregate: max(S2.popularity)  (cost=1.82 rows=1) (actual time=0.097..0.097 rows=1 loops=99)
                    -> Index lookup on S2 using artistID (artistID=A.artistID)  (cost=1.42 rows=4) (actual time=0.010..0.093 rows=41 loops=99)
 |
+----------------------------------------------------------------------------------------------------------------------------------------------------------------
```

My initial key is 1814.08. The nested join cost of 1814.08. Scanning the first table A of Artist cost 120.8 and the aggregate cost was 1.82 for the max popularity.

INDEX CREATED for Song(artistID)

```
mysql> CREATE INDEX song_a ON Song(artistID);
Query OK, 0 rows affected (0.58 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
----------------------------------------------------------------------------------------------------------------------------------------------------------------+
 -> Limit: 15 row(s)  (cost=1815.50 rows=15) (actual time=1.163..6.676 rows=15 loops=1)
  -> Nested loop inner join  (cost=1815.50 rows=4842) (actual time=1.162..6.673 rows=15 loops=1)
      -> Table scan on A  (cost=120.80 rows=1193) (actual time=0.068..0.073 rows=15 loops=1)
      -> Filter: (S.popularity = (select #2))  (cost=1.02 rows=4) (actual time=0.099..0.439 rows=1 loops=15)
          -> Index lookup on S using song_a (artistID=A.artistID)  (cost=1.02 rows=4) (actual time=0.009..0.017 rows=7 loops=15)
          -> Select #2 (subquery in condition; dependent)
              -> Aggregate: max(S2.popularity)  (cost=1.83 rows=1) (actual time=0.063..0.063 rows=1 loops=99)
                  -> Index lookup on S2 using song_a (artistID=A.artistID)  (cost=1.42 rows=4) (actual time=0.009..0.059 rows=41 loops=99)
|
----------------------------------------------------------------------------------------------------------------------------------------------------------------
```

We created an index called song_a that was for Song artistID. We chose this because we thought adding the artistID would lower cost as it would be quicker than referencing the primary key. However, there was an increased the base cost from 1814.08 to 1815.50. With the scan of Artist Table A being the same cost but the lookup using song_a is 1.02 instead of 1.01. Additionally the aggregate increased from 1.82 to 1.83. We decided to remove that index and noticed that when dropping song_a we needed to drop the foreign key to the artist and then recreate the foreign key. This for some reason increased the cost to 1858.90 but we aren't sure why.

INDEX CREATED Song(popularity)

```
mysql> CREATE INDEX song_p ON Song(popularity);
Query OK, 0 rows affected (0.21 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (cost=1858.90 rows=15) (actual time=5.271..12.025 rows=15 loops=1)
    -> Nested loop inner join  (cost=1858.90 rows=4966) (actual time=5.257..12.010 rows=15 loops=1)
        -> Table scan on A  (cost=120.80 rows=1193) (actual time=0.055..0.061 rows=15 loops=1)
        -> Filter: (S.popularity = (select #2))  (cost=1.04 rows=4) (actual time=0.217..0.627 rows=1 loops=15)
            -> Index lookup on S using Song_ibfk_1 (artistID=A.artistID)  (cost=1.04 rows=4) (actual time=0.015..0.024 rows=7 loops=15)
            -> Select #2 (subquery in condition; dependent)
                -> Aggregate: max(S2.popularity)  (cost=1.87 rows=1) (actual time=0.090..0.090 rows=1 loops=99)
                    -> Index lookup on S2 using Song_ibfk_1 (artistID=A.artistID)  (cost=1.46 rows=4) (actual time=0.013..0.085 rows=41 loops=99)
 |
+
```

We then created another index for Song popularity called song_p and it had no effect on the query except it increased overall cost, the index lookup cost from 1.03 to 1.04 the aggregate cost from 1.83 to 1.87. So we decided to remove it
INDEX CREATED Song(songNa

```
ysql> CREATE INDEX song_n ON Song(songName);
uery OK, 0 rows affected (0.22 sec)
ecords: 0  Duplicates: 0  Warnings: 0
```

```
| -> Limit: 15 row(s)  (cost=1858.90 rows=15) (actual time=1.055..6.121 rows=15 loops=1)
    -> Nested loop inner join  (cost=1858.90 rows=4966) (actual time=1.053..6.118 rows=15 loops=1)
        -> Table scan on A  (cost=120.80 rows=1193) (actual time=0.064..0.069 rows=15 loops=1)
        -> Filter: (S.popularity = (select #2))  (cost=1.04 rows=4) (actual time=0.091..0.403 rows=1 loops=15)
            -> Index lookup on S using Song_ibfk_1 (artistID=A.artistID)  (cost=1.04 rows=4) (actual time=0.010..0.019 rows=7 loops=15)
            -> Select #2 (subquery in condition; dependent)
                -> Aggregate: max(S2.popularity)  (cost=1.87 rows=1) (actual time=0.057..0.057 rows=1 loops=99)
                    -> Index lookup on S2 using Song_ibfk_1 (artistID=A.artistID)  (cost=1.46 rows=4) (actual time=0.007..0.054 rows=41 loops=99)
 |
```

Lastly created an index called song_n that indexed to Song's songName.It also showed no change within the cost. Overall we believe that no additional indexing is needed as all indexing seemed to have a negative impact on this query.