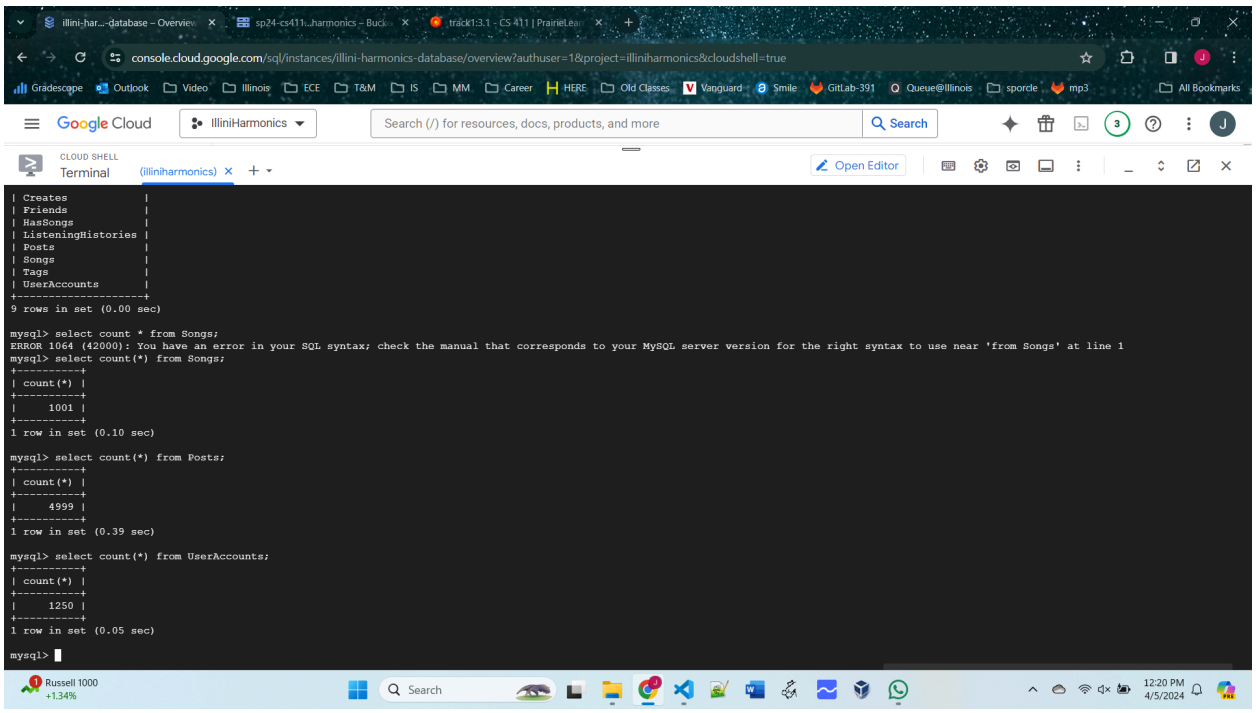
Project Track 1.3: IlliniHarmonics

Part 1: Database Design

Implementation on GCP:



DDL Commands:

For DDL commands, please see ‘code/sql-setup/tables.sql’. This file contains all of the tables used in our database, and to create them, we simply included this file in our database instance on GCP.

Table Row Counts:

Songs: 1001 entries
Posts: 4999 entries
UserAccounts: 1250 entries

```
mysql> select count(*) from Songs;
+----------+
| count(*) |
+----------+
|     1001 |
+----------+
1 row in set (0.10 sec)

mysql> select count(*) from Posts;
+----------+
| count(*) |
+----------+
|     4999 |
+----------+
1 row in set (0.39 sec)

mysql> select count(*) from UserAccounts;
+----------+
| count(*) |
+----------+
|     1250 |
+----------+
1 row in set (0.05 sec)

mysql>
```

Part 2: Advanced Queries

The advanced queries themselves can be found at ‘code/sql-queries/adv_queries.sql’. This table contains the functional versions of the queries that were run. Below is a description of the purpose of each query along with the output when run on the database.

Query 1: Popular Songs by Genre

This query selects the most popular songs of a given genre, in this case, ‘Hip Hop’. Our database will eventually use this query to return a recommendation list of popular songs to a user based on their desired genre.

```
mysql> SELECT DISTINCT songid, songname, Artists.genre
    -> FROM Songs NATURAL JOIN HasSongs NATURAL JOIN Artists
    -> GROUP BY Artists.genre, songid, songname
    -> HAVING genre LIKE '%Hip Hop%'
    -> ORDER BY Artists.genre
    -> LIMIT 15
    -> ;
+----------------------+-----------------------------------+----------+
| songid               | songname                          | genre    |
+----------------------+-----------------------------------+----------+
 |00FROhC5g4iJdax5US8jRr | Satisfy You                       | Hip Hop
 |00i2HU7TEzzftShjRrDSEF | Changes                           | Hip Hop
 |01oPNCtTniFT3YM4K3ksTf | Opposite of Adults                | Hip Hop
 |024Q1MI4uaYHF6j2Q2ZYgF | Overnight Celebrity               | Hip Hop
 |02kDW379Yfd5PzW5A6vuGt | Lemonade                          | Hip Hop
 |03ERnTvSV5fy8WAyvA4uNP | Big Gangsta                       | Hip Hop
 |03tqyYWC9Um2ZqU0ZN849H | No Hands (feat. Roscoe Dash & Wale) | Hip Hop
 |05cA5ezq5ChgS4DBbQ3jyk | Tatted Up                         | Hip Hop
 |07rmSXN6vNoquX1AsWd9pP | Lalala                            | Hip Hop
 |08uGhvS5MfBk7crUCpnjva | Blame It                          | Hip Hop
 |09EwNbGvUyu7BDEYG0cJro | Can't C Me                        | Hip Hop
 |0aJ2hFC6e9Z00Lx9RIRNcT | SWISH                             | Hip Hop
 |0AOvNRgl0SMfOibWA5bP8o | Money Trees                       | Hip Hop
 |0B2ZuTLZnWQ6gz8RlNwBu1 | On Fire                           | Hip Hop
 |0b8qcIPuFq77U6ifMUAVUP | All Gold Everything - Remix       | Hip Hop
+----------------------+-----------------------------------+----------+
15 rows in set (0.01 sec)
```

Query 2: Songs with Most Listens

This query sums the amount of times each song appears in the Listening Histories table, and returns the songs which were listened to most.

```
mysql> SELECT DISTINCT Songs.songid, songname, artistid, artistname, COUNT(ListeningHistories.songid) AS listens
    -> FROM (SELECT DISTINCT songname FROM Songs) AS SongNames NATURAL JOIN Songs NATURAL JOIN HasSongs NATURAL JOIN Artists NATURAL JOIN ListeningHistories
    -> GROUP BY songid, artistid
    -> ORDER BY listens DESC
    -> LIMIT 15;
+----------------------+----------------------+----------------------+---------------------+---------+
| songid               | songname             | artistid             | artistname          | listens |
+----------------------+----------------------+----------------------+---------------------+---------+
| 7nYvUtkQMx1v80S2FH2s9J | Regulate             | 1Oa0bMld0A3u5OTYfMzp5h | Nate Dogg           |      57 |
| 5k38wzpLb15YgncyWdTZE4 | Him & I (with Halsey) | 26VFTg2z8YR0cCuwLzESi2 | Halsey              |      57 |
| 5k38wzpLb15YgncyWdTZE4 | Him & I (with Halsey) | 02kJSzxNuaWGqwubyUba0Z | G-Eazy              |      57 |
| 7nYvUtkQMx1v80S2FH2s9J | Regulate             | 2B4ZHz4QDWJTXPFPgO5peE | Warren G            |      57 |
| 3kW5Rq9AILOQQuYTSKNkQw | Erbody But Me        | 3SEJQy90Vgg1aSnZ9b4AU9 | Krizz Kaliko        |      55 |
| 3kW5Rq9AILOQQuYTSKNkQw | Erbody But Me        | 12ceFY8dHtc9XsIknO45Pa | Bizzy               |      55 |
| 3kW5Rq9AILOQQuYTSKNkQw | Erbody But Me        | 6UBA15slIuadJ8h2lPRPos | Tech N9ne           |      55 |
| 5PsP6R5YDRh5IaqAey04AA | Jigglin              | 44PA0rCQXikgOWbfY7Fq7m | Ying Yang Twins     |      55 |
| 6M47gaKejso9772SKTa3yH | Face Off             | 6UBA15slIuadJ8h2lPRPos | Tech N9ne           |      53 |
| 6M47gaKejso9772SKTa3yH | Face Off             | 1T8S48bD91THNKBByWBDyn | Dwayne Johnson      |      53 |
| 6M47gaKejso9772SKTa3yH | Face Off             | 1A8NEJYo3VW9aIZ6l55Dgu | Joey Cool           |      53 |
| 6ihObRBTB8xdSH2mlERtOX | Slow Motion          | 0rG0AZBscc8S8q1ahIsasI | JUVENILE            |      53 |
| 3Ti0GdlrotgwsAVBBugv0I | Can I Kick It?       | 09hVIj6vWgoCDtT03h8ZCa | A Tribe Called Quest |      53 |
| 6ihObRBTB8xdSH2mlERtOX | Slow Motion          | 6tBnRI3ubWYclKSbGvGtKd | Soulja Slim         |      53 |
| 6M47gaKejso9772SKTa3yH | Face Off             | 7vTyZmNA6LC0WRDg7V6yle | King Iso            |      53 |
+----------------------+----------------------+----------------------+---------------------+---------+
15 rows in set (0.15 sec)
```

Query 3: Songs with Highest Rating

This query computes the average rating of each song across all posts, and returns the top 15. This is useful to analyze the posts from our users and determine which songs are most popular, and therefore should be recommended to other users.

```
mysql> SELECT DISTINCT songid, songname, AVG(rating) as AverageRating
    -> FROM Posts NATURAL JOIN Songs NATURAL JOIN Artists
    -> GROUP BY genre, songid
    -> ORDER BY AVG(rating) DESC
    -> LIMIT 15;

\
```

songid	songname	AverageRating
1LnEW2Dx4pFZ4yZ4dXFY4f	Batter Up	5.0000
20on25jryn53hWghthWWW3	Do It To It	5.0000
37BZB0z9T8Xu7U3e65qxFy	Save Your Tears (with Ariana Grande) (Remix)	5.0000
3FfzdhMWVcjtyKbyuV0L7u	El Chapo	5.0000
3uqinR4FCjLv28bkrTdNX5	A Milli	5.0000
3XabgBQYC7H80agKcAq83Y	Shake That Monkey (feat. Lil' Jon & The EastSide Boyz)	5.0000
4QEoAZj6fVeSHj9ti2JpN8	Naggin'	5.0000
4tIPgKJa5DVpJqu7GvH4gU	Screw Dat	5.0000
5lFDtgWsjRJu8fPOAyJIAK	Back To Back	5.0000
4sWuPniClkInYAnzXAYUF	Camelot	4.7500
1pNN9kkePpNF19LYeYZ5af	Tell Tell Tell (Stop Snitchin') (feat. Mr. Bigg, Lyfe Jennings & Young Jeezy) - Explicit Album Version	4.6667
6Iuh3vNjpWhSeHM3RZgzQL	Bam	4.6667
0sf12qNH5qcw8qpgymFOqD	Blinding Lights	4.5000
1ZAyjvIk9YiD76yYyOTEG6	Plug Walk	4.5000
3YOjgTpUVNWvpYeb2mtTF7	Grillz	4.5000

```
15 rows in set (32.00 sec)
```

Query 4: Popular Artists Among Friends

This query iterates through each set of friends and determines which artists have been listened to most. This would allow a user to get artist recommendations based off of the listening histories of their friends.

```
mysql> SELECT DISTINCT artistid, artistname, COUNT(*) as ArtistCount
    -> FROM UserAccounts NATURAL JOIN ListeningHistories NATURAL JOIN Songs
    ->      NATURAL JOIN HasSongs NATURAL JOIN Artists
    -> WHERE UserAccounts.userid IN (SELECT userid2
    ->                               FROM Friends
    ->                               WHERE userid1 = UserAccounts.userid)
    -> GROUP BY artistid
    -> ORDER BY ArtistCount DESC
    -> LIMIT 15;
```

artistid	artistname	ArtistCount
137W8MRPWKqSmrBGDBFSop	Wiz Khalifa	4
5spEJXLwD1sKUdC2bnOHPg	Bone Thugs-N-Harmony	3
2gBjLmx6zQnFGQJCAQpRgw	Nelly	2
5me0Irg2ANcsgc93uaYrpb	The Notorious B.I.G.	2
3crnzLy8R4lVwaigKEOz7V	E-40	2
5K4W6rqBFWDnAN6FQUkS6x	Kanye West	2
3TVXtAsR1Inumwj472S9r4	Drake	2
5LHRHt1k9lMyONurDHEdrp	Tyga	2
7bXgB6jMjp9ATFy66eO08Z	Chris Brown	2
7hJcb9fa4alzcOq3EaNPoG	Snoop Dogg	2
67nwj3Y5sZQLl72VNUHEYE	Wale	1
6f4XkbvYlXMH0QgVRzW0sM	Waka Flocka Flame	1
7LnaAXbDVIL75IVPnndf7w	Jamie Foxx	1
6vXTefBL93Dj5IqAWq6OTv	French Montana	1
50co4Is1HCEo8bhOyUWKpn	Young Thug	1

```
15 rows in set (0.24 sec)
```

Part 3: Indexing

Strategy Overview:

In general, we aim to decrease the time and cost of each of our queries as much as possible. We will do this by creating indices to help organize the database and reduce data retrieval time and the number of I/O operations on disk.

However, our analysis found that no combination of indices were especially successful for the majority of queries, aside from query 2. The reason for this is that the majority of our queries involve expensive and time consuming join operations. These operations are generally unavoidable, and consume the majority of resources for each query. Naturally, we attempted to index based on the elements being joined in these queries. Unfortunately, this is not possible. Each natural join present in all four queries either joins on a primary key or a foreign key, both of which are already considered indices. Therefore, it is impossible to decrease the time and cost further than the default indexing by a significant margin without redesigning the database completely. While we did attempt to index based on attributes in WHERE or GROUP-BY clauses, the cost saved was miniscule in comparison to the cost of the joins. This explanation applies to queries 1, 3, and 4, and therefore our explanations of the final strategy selected will be more brief.

Query 1:

DEFAULT INDEXING:

```
| -> Limit: 15 row(s)  (actual time=14.919..14.923 rows=15 loops=1)
    -> Sort: Artists.genre, Songs.songid, Songs.songname  (actual time=14.918..14.920 rows=15 loops=1)
        -> Filter: (Artists.genre like '%Hip Hop%')  (actual time=13.495..14.202 rows=660 loops=1)
            -> Table scan on <temporary>  (cost=1245.27..1270.22 rows=1798) (actual time=13.483..13.785 rows=1292 loops=1)
                -> Temporary table with deduplication  (cost=1245.26..1245.26 rows=1798) (actual time=13.479..13.479 rows=1292 loops=1)
                    -> Nested loop inner join  (cost=1065.49 rows=1798) (actual time=0.116..10.963 rows=1795 loops=1)
                        -> Nested loop inner join  (cost=436.31 rows=1798) (actual time=0.105..6.085 rows=1795 loops=1)
                            -> Table scan on Artists  (cost=68.85 rows=676) (actual time=0.077..0.474 rows=676 loops=1)
                            -> Covering index lookup on HasSongs using PRIMARY (artistid=Artists.artistid)  (cost=0.28 rows=3) (actual time=0.005..0.008 rows=3 loops=676
                        -> Single-row index lookup on Songs using PRIMARY (songid=HasSongs.songid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
|
```

INDEX 1: Genre

```
| -> Limit: 15 row(s)  (actual time=8.509..8.511 rows=15 loops=1)
    -> Sort: Artists.genre, Songs.songid, Songs.songname  (actual time=8.508..8.509 rows=15 loops=1)
        -> Filter: (Artists.genre like '%Hip Hop%')  (actual time=7.636..8.073 rows=660 loops=1)
            -> Table scan on <temporary>  (cost=1245.27..1270.22 rows=1798) (actual time=7.631..7.828 rows=1292 loops=1)
                -> Temporary table with deduplication  (cost=1245.26..1245.26 rows=1798) (actual time=7.627..7.627 rows=1292 loops=1)
                    -> Nested loop inner join  (cost=1065.49 rows=1798) (actual time=0.108..6.171 rows=1795 loops=1)
                        -> Nested loop inner join  (cost=436.31 rows=1798) (actual time=0.100..3.313 rows=1795 loops=1)
                            -> Covering index scan on Artists using genre_idx  (cost=68.85 rows=676) (actual time=0.069..0.258 rows=676 loops=1)
                            -> Covering index lookup on HasSongs using PRIMARY (artistid=Artists.artistid)  (cost=0.28 rows=3) (actual time=0.003..0.004 rows=3 loops=676
                        -> Single-row index lookup on Songs using PRIMARY (songid=HasSongs.songid)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1795)
|
```

INDEX 2: Genre, Songname

```
| -> Limit: 15 row(s)  (actual time=8.519..8.521 rows=15 loops=1)
    -> Sort: Artists.genre, Songs.songid, Songs.songname  (actual time=8.518..8.519 rows=15 loops=1)
        -> Filter: (Artists.genre like '%Hip Hop%')  (actual time=7.676..8.091 rows=660 loops=1)
            -> Table scan on <temporary>  (cost=1245.27..1270.22 rows=1798) (actual time=7.670..7.859 rows=1292 loops=1)
                -> Temporary table with deduplication  (cost=1245.26..1245.26 rows=1798) (actual time=7.666..7.666 rows=1292 loops=1)
                    -> Nested loop inner join  (cost=1065.49 rows=1798) (actual time=0.147..6.194 rows=1795 loops=1)
                        -> Nested loop inner join  (cost=436.31 rows=1798) (actual time=0.138..3.328 rows=1795 loops=1)
                            -> Covering index scan on Artists using genre_idx  (cost=68.85 rows=676) (actual time=0.099..0.293 rows=676 loops=1)
                            -> Covering index lookup on HasSongs using PRIMARY (artistid=Artists.artistid)  (cost=0.28 rows=3) (actual time=0.003..0.004 rows=3 loops=676)
                        -> Single-row index lookup on Songs using PRIMARY (songid=HasSongs.songid)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1795)
```

INDEX 3: Songname

```
| -> Limit: 15 row(s)  (actual time=8.576..8.578 rows=15 loops=1)
    -> Sort: Artists.genre, Songs.songid, Songs.songname  (actual time=8.575..8.576 rows=15 loops=1)
        -> Filter: (Artists.genre like '%Hip Hop%')  (actual time=7.682..8.143 rows=660 loops=1)
            -> Table scan on <temporary>  (cost=1245.27..1270.22 rows=1798) (actual time=7.674..7.910 rows=1292 loops=1)
                -> Temporary table with deduplication  (cost=1245.26..1245.26 rows=1798) (actual time=7.669..7.669 rows=1292 loops=1)
                    -> Nested loop inner join  (cost=1065.49 rows=1798) (actual time=0.062..6.220 rows=1795 loops=1)
                        -> Nested loop inner join  (cost=436.31 rows=1798) (actual time=0.055..3.414 rows=1795 loops=1)
                            -> Table scan on Artists  (cost=68.85 rows=676) (actual time=0.042..0.275 rows=676 loops=1)
                            -> Covering index lookup on HasSongs using PRIMARY (artistid=Artists.artistid)  (cost=0.28 rows=3) (actual time=0.003..0.004 rows=3 loops=676)
                        -> Single-row index lookup on Songs using PRIMARY (songid=HasSongs.songid)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1795)
```

FINAL STRATEGY: INDEX 1: Genre

Adding the genre index can be shown to marginally decrease the query run time from a default of 0.004 seconds to a time of 0.002 seconds. However, this index does not reduce the scan time nor the scan cost on the artist table when compared to the default table. Additionally, as seen by the second and third indices, despite the songname being used in the SELECT field of the query, adding an index on it does not change query execution at all, and it is simply ignored. Overall, the possible decreases in cost and time are significantly limited by the joins occurring on primary keys as described in the overview.

Query 2:

DEFAULT INDEXING:

```
| -> Limit: 15 row(s)  (actual time=152.980..152.982 rows=15 loops=1)
    -> Sort: listens DESC, limit input to 15 row(s) per chunk  (actual time=152.979..152.980 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=151.985..152.689 rows=1795 loops=1)
            -> Aggregate using temporary table  (actual time=151.982..151.982 rows=1795 loops=1)
                -> Nested loop inner join  (cost=881670.02 rows=6396553) (actual time=1.632..62.352 rows=66002 loops=1)
                    -> Nested loop inner join  (cost=162512.76 rows=179859) (actual time=1.595..11.369 rows=1795 loops=1)
                        -> Nested loop inner join  (cost=144481.92 rows=179859) (actual time=1.587..7.631 rows=1795 loops=1)
                            -> Filter: (Songs.songname = SongNames.songname)  (cost=100540.76 rows=100200) (actual time=1.570..3.218 rows=1001 loops=1)
                                -> Inner hash join (<hash>(Songs.songname)=<hash>(SongNames.songname))  (cost=100540.76 rows=100200) (actual time=1.568..2.747 rows=1001 loops=1)
                                    -> Table scan on Songs  (cost=0.02 rows=1001) (actual time=0.038..0.604 rows=1001 loops=1)
                                    -> Hash
                                        -> Table scan on SongNames  (cost=317.08..332.08 rows=1001) (actual time=1.049..1.188 rows=978 loops=1)
                                            -> Materialize  (cost=317.06..317.06 rows=1001) (actual time=1.049..1.049 rows=978 loops=1)
                                                -> Table scan on <temporary>  (cost=201.96..216.96 rows=1001) (actual time=0.728..0.848 rows=978 loops=1)
                                                    -> Temporary table with deduplication  (cost=201.95..201.95 rows=1001) (actual time=0.727..0.727 rows=978 loops=1)
                                                        -> Table scan on Songs  (cost=101.85 rows=1001) (actual time=0.060..0.388 rows=1001 loops=1)
                            -> Covering index lookup on HasSongs using FK_songid_HS (songid=Songs.songid)  (cost=0.26 rows=2) (actual time=0.003..0.004 rows=2 loops=1001)
                        -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.00 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
                    -> Covering index lookup on ListeningHistories using FK_songid_LH (songid=Songs.songid)  (cost=0.44 rows=36) (actual time=0.019..0.026 rows=37 loops=1795)
```

INDEX 1: Songname

```
-> Limit: 15 row(s)  (actual time=167.450..167.452 rows=15 loops=1)
  -> Sort: listens DESC, limit input to 15 row(s) per chunk  (actual time=167.449..167.451 rows=15 loops=1)
      -> Table scan on <temporary>  (actual time=166.523..167.179 rows=1795 loops=1)
          -> Aggregate using temporary table  (actual time=166.521..166.521 rows=1795 loops=1)
              -> Nested loop inner join  (cost=9479.58 rows=65404) (actual time=0.913..71.004 rows=66002 loops=1)
                  -> Nested loop inner join  (cost=2126.23 rows=1839) (actual time=0.887..15.150 rows=1795 loops=1)
                      -> Nested loop inner join  (cost=1482.57 rows=1839) (actual time=0.879..11.255 rows=1795 loops=1)
                          -> Nested loop inner join  (cost=1033.27 rows=1025) (actual time=0.868..6.312 rows=1001 loops=1)
                              -> Filter: (SongNames.songname is not null)  (cost=301.86..115.11 rows=1001) (actual time=0.857..1.283 rows=978 loops=1)
                                  -> Table scan on SongNames  (cost=302.06..317.06 rows=1001) (actual time=0.856..1.141 rows=978 loops=1)
                                      -> Materialize  (cost=302.05..302.05 rows=1001) (actual time=0.854..0.854 rows=978 loops=1)
                                          -> Group (no aggregates)  (cost=201.95 rows=1001) (actual time=0.050..0.638 rows=978 loops=1)
                                              -> Covering index scan on Songs using songname_idx  (cost=101.85 rows=1001) (actual time=0.045..0.317 rows=1001 loops=1)
                              -> Covering index lookup on Songs using songname_idx (songname=SongNames.songname)  (cost=0.81 rows=1) (actual time=0.004..0.005 rows=1 loops=978)
                          -> Covering index lookup on HasSongs using FK_songid_HS (songid=Songs.songid)  (cost=0.26 rows=2) (actual time=0.004..0.005 rows=2 loops=1001)
                      -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
                  -> Covering index lookup on ListeningHistories using FK_songid_LH (songid=Songs.songid)  (cost=0.44 rows=36) (actual time=0.021..0.028 rows=37 loops=1795)
```

INDEX 2: Artistname

```
| -> Limit: 15 row(s)  (actual time=816.304..816.308 rows=15 loops=1)
    -> Sort: listens DESC, limit input to 15 row(s) per chunk  (actual time=816.303..816.305 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=814.829..815.791 rows=1795 loops=1)
            -> Aggregate using temporary table  (actual time=814.824..814.824 rows=1795 loops=1)
                -> Nested loop inner join  (cost=1138116.98 rows=6390163) (actual time=187.961..672.886 rows=66002 loops=1)
                    -> Filter: (SongNames.songname = Songs.songname)  (cost=181410.46 rows=179680) (actual time=128.404..134.331 rows=1795 loops=1)
                        -> Inner hash join (<hash>(SongNames.songname)=<hash>(Songs.songname))  (cost=181410.46 rows=179680) (actual time=128.402..132.593 rows=1795 loops=1)
                            -> Table scan on SongNames  (cost=317.08..332.08 rows=1001) (actual time=1.145..1.791 rows=978 loops=1)
                                -> Materialize  (cost=317.06..317.06 rows=1001) (actual time=1.145..1.145 rows=978 loops=1)
                                    -> Table scan on <temporary>  (cost=201.96..216.96 rows=1001) (actual time=0.785..0.921 rows=978 loops=1)
                                        -> Temporary table with deduplication  (cost=201.95..201.95 rows=1001) (actual time=0.783..0.783 rows=978 loops=1)
                                            -> Table scan on Songs  (cost=101.85 rows=1001) (actual time=0.043..0.364 rows=1001 loops=1)
                            -> Hash
                                -> Nested loop inner join  (cost=1446.00 rows=1795) (actual time=81.630..125.761 rows=1795 loops=1)
                                    -> Nested loop inner join  (cost=817.75 rows=1795) (actual time=81.612..123.316 rows=1795 loops=1)
                                        -> Covering index scan on HasSongs using FK_songid_HS  (cost=189.50 rows=1795) (actual time=81.549..118.836 rows=1795 loops=1)
                                        -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
                                    -> Single-row index lookup on Songs using PRIMARY (songid=HasSongs.songid)  (cost=0.25 rows=1) (actual time=0.001..0.001 rows=1 loops=1795)
                    -> Covering index lookup on ListeningHistories using FK_songid_LH (songid=HasSongs.songid)  (cost=1.77 rows=36) (actual time=0.284..0.296 rows=37 loops=1795)
```

INDEX 2, PART 2: Artistname

```
| -> Limit: 15 row(s)  (actual time=181.459..181.462 rows=15 loops=1)
    -> Sort: listens DESC, limit input to 15 row(s) per chunk  (actual time=181.458..181.460 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=180.415..181.161 rows=1795 loops=1)
            -> Aggregate using temporary table  (actual time=180.412..180.412 rows=1795 loops=1)
                -> Nested loop inner join  (cost=881670.02 rows=6396553) (actual time=1.863..73.236 rows=66002 loops=1)
                    -> Nested loop inner join  (cost=162512.76 rows=179859) (actual time=1.824..15.122 rows=1795 loops=1)
                        -> Nested loop inner join  (cost=144481.92 rows=179859) (actual time=1.813..10.803 rows=1795 loops=1)
                            -> Filter: (Songs.songname = SongNames.songname)  (cost=100540.76 rows=100200) (actual time=1.786..3.817 rows=1001 loops=1)
                                -> Inner hash join (<hash>(Songs.songname)=<hash>(SongNames.songname))  (cost=100540.76 rows=100200) (actual time=1.784..3.296 rows=1001 loops=1
                                    -> Table scan on Songs  (cost=0.02 rows=1001) (actual time=0.109..0.831 rows=1001 loops=1)
                                    -> Hash
                                        -> Table scan on SongNames  (cost=317.08..332.08 rows=1001) (actual time=1.157..1.280 rows=978 loops=1)
                                            -> Materialize  (cost=317.06..317.06 rows=1001) (actual time=1.156..1.156 rows=978 loops=1)
                                                -> Table scan on <temporary>  (cost=201.96..216.96 rows=1001) (actual time=0.831..0.957 rows=978 loops=1)
                                                    -> Temporary table with deduplication  (cost=201.95..201.95 rows=1001) (actual time=0.830..0.830 rows=978 loops=1)
                                                        -> Table scan on Songs  (cost=101.85 rows=1001) (actual time=0.079..0.423 rows=1001 loops=1)
                            -> Covering index lookup on HasSongs using FK_songid_HS (songid=Songs.songid)  (cost=0.26 rows=2) (actual time=0.006..0.007 rows=2 loops=1001)
                        -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.00 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
                    -> Covering index lookup on ListeningHistories using FK_songid_LH (songid=Songs.songid)  (cost=0.44 rows=36) (actual time=0.023..0.029 rows=37 loops=1795)
```

INDEX 3: Songname, Artistname (2.2)

```
| -> Limit: 15 row(s)  (actual time=189.616..189.619 rows=15 loops=1)
    -> Sort: listens DESC, limit input to 15 row(s) per chunk  (actual time=189.615..189.617 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=188.515..189.308 rows=1795 loops=1)
            -> Aggregate using temporary table  (actual time=188.512..188.512 rows=1795 loops=1)
                -> Nested loop inner join  (cost=9479.58 rows=65404) (actual time=0.991..68.194 rows=66002 loops=1)
                    -> Nested loop inner join  (cost=2126.23 rows=1839) (actual time=0.964..13.915 rows=1795 loops=1)
                        -> Nested loop inner join  (cost=1482.57 rows=1839) (actual time=0.955..9.995 rows=1795 loops=1)
                            -> Nested loop inner join  (cost=1033.27 rows=1025) (actual time=0.945..5.015 rows=1001 loops=1)
                                -> Filter: (SongNames.songname is not null)  (cost=301.86..115.11 rows=1001) (actual time=0.929..1.362 rows=978 loops=1)
                                    -> Table scan on SongNames  (cost=302.06..317.06 rows=1001) (actual time=0.928..1.206 rows=978 loops=1)
                                        -> Materialize  (cost=302.05..302.05 rows=1001) (actual time=0.926..0.926 rows=978 loops=1)
                                            -> Group (no aggregates)  (cost=201.95 rows=1001) (actual time=0.064..0.695 rows=978 loops=1)
                                                -> Covering index scan on Songs using songname_idx  (cost=101.85 rows=1001) (actual time=0.057..0.345 rows=1001 loops=1)
                                -> Covering index lookup on Songs using songname_idx (songname=SongNames.songname)  (cost=0.81 rows=1) (actual time=0.003..0.004 rows=1 loops=978)
                            -> Covering index lookup on HasSongs using FK_songid_HS (songid=Songs.songid)  (cost=0.26 rows=2) (actual time=0.004..0.005 rows=2 loops=1001)
                        -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
                    -> Covering index lookup on ListeningHistories using FK_songid_LH (songid=Songs.songid)  (cost=0.44 rows=36) (actual time=0.021..0.027 rows=37 loops=1795)
```

FINAL STRATEGY: Songname

This query loops over each song and sums how many times it appears in listening histories. Naturally, the best index for this query is songname, which drastically reduces the cost from a default of 881670 from the outer nested loop join to 9479 as well as halving the time from 0.04 seconds to 0.02 seconds. Adding an index based on artistname actually increases the cost and time; although adding in a GROUP BY artistname statement to the query allows the index on artistname to return to the default cost and time. Overall, in this case, it is better to keep the query simple and only index based on songname.

Query 3:

DEFAULT INDEXING:

```
| -> Limit: 15 row(s)  (actual time=34862.419..34862.422 rows=15 loops=1)
    -> Sort with duplicate removal: AverageRating DESC, Posts.songid, Songs.songname  (actual time=34862.418..34862.420 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=34857.014..34858.237 rows=2976 loops=1)
            -> Aggregate using temporary table  (actual time=34857.006..34857.006 rows=2976 loops=1)
                -> Nested loop inner join  (cost=345704.93 rows=1721772) (actual time=0.313..2166.593 rows=3379324 loops=1)
                    -> Inner hash join (no condition)  (cost=172254.23 rows=1721772) (actual time=0.297..373.960 rows=3379324 loops=1)
                        -> Index range scan on Posts using FK_songid_P over (NULL < songid), with index condition: (Posts.songid is not null)  (cost=0.19 rows=2547) (actual time=0.017..27.42
ws=4999 loops=1)
                        -> Hash
                            -> Covering index scan on Artists using genre_idx  (cost=68.85 rows=676) (actual time=0.046..0.212 rows=676 loops=1)
                    -> Single-row index lookup on Songs using PRIMARY (songid=Posts.songid)  (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=3379324)
```

INDEX 1: Genre

```
-> Limit: 15 row(s)  (actual time=34862.419..34862.422 rows=15 loops=1)
  -> Sort with duplicate removal: AverageRating DESC, Posts.songid, Songs.songname  (actual time=34862.418..34862.420 rows=15 loops=1)
      -> Table scan on <temporary>  (actual time=34857.014..34858.237 rows=2976 loops=1)
          -> Aggregate using temporary table  (actual time=34857.006..34857.006 rows=2976 loops=1)
              -> Nested loop inner join  (cost=345704.93 rows=1721772) (actual time=0.313..2166.593 rows=3379324 loops=1)
                  -> Inner hash join (no condition)  (cost=172254.23 rows=1721772) (actual time=0.297..373.960 rows=3379324 loops=1)
                      -> Index range scan on Posts using FK_songid_P over (NULL < songid), with index condition: (Posts.songid is not null)  (cost=0.19 rows=2547) (actual time=0.017..27.427 r
=4999 loops=1)
                      -> Hash
                          -> Covering index scan on Artists using genre_idx  (cost=68.85 rows=676) (actual time=0.046..0.212 rows=676 loops=1)
                  -> Single-row index lookup on Songs using PRIMARY (songid=Posts.songid)  (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=3379324)
```

INDEX 2: Rating

```
| -> Limit: 15 row(s)  (actual time=33362.203..33362.206 rows=15 loops=1)
    -> Sort with duplicate removal: AverageRating DESC, Posts.songid, Songs.songname  (actual time=33362.202..33362.204 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=33336.051..33337.295 rows=2976 loops=1)
            -> Aggregate using temporary table  (actual time=33336.044..33336.044 rows=2976 loops=1)
                -> Nested loop inner join  (cost=349553.71 rows=1721772) (actual time=62.468..2768.171 rows=3379324 loops=1)
                    -> Inner hash join (no condition)  (cost=172282.51 rows=1721772) (actual time=36.721..765.216 rows=3379324 loops=1)
                        -> Index range scan on Posts using posts_songid_idx over (NULL < songid), with index condition: (Posts.songid is not null)  (cost=0.21 rows=2547) (actual time=30.041..382
.431 rows=4999 loops=1)
                        -> Hash
                            -> Table scan on Artists  (cost=72.60 rows=676) (actual time=3.272..6.481 rows=676 loops=1)
                    -> Single-row index lookup on Songs using PRIMARY (songid=Posts.songid)  (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=3379324)
```

INDEX 3: Songname, genre

```
| -> Limit: 15 row(s)  (actual time=32902.607..32902.610 rows=15 loops=1)
    -> Sort with duplicate removal: AverageRating DESC, Posts.songid, Songs.songname  (actual time=32902.606..32902.608 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=32897.410..32898.615 rows=2976 loops=1)
            -> Aggregate using temporary table  (actual time=32897.402..32897.402 rows=2976 loops=1)
                -> Nested loop inner join  (cost=345704.93 rows=1721772) (actual time=0.296..2186.175 rows=3379324 loops=1)
                    -> Inner hash join (no condition)  (cost=172254.23 rows=1721772) (actual time=0.283..375.089 rows=3379324 loops=1)
                        -> Index range scan on Posts using posts_songid_idx over (NULL < songid), with index condition: (Posts.songid is not null)  (cost=0.19 rows=2547) (actual time=0.014..28.0
64 rows=4999 loops=1)
                        -> Hash
                            -> Covering index scan on Artists using genre_idx  (cost=68.85 rows=676) (actual time=0.042..0.193 rows=676 loops=1)
                    -> Single-row index lookup on Songs using PRIMARY (songid=Posts.songid)  (cost=0.00 rows=1) (actual time=0.000..0.000 rows=1 loops=3379324)
```

FINAL STRATEGY: Default Indexing

This is another query where the bulk of the cost comes from expensive natural joins on primary keys. Additionally, this query is drastically slowed by the computation of AVG(rating) for each song. As the Posts table contains nearly 5000 entries, and there are more than 1000 songs, this is an incredibly expensive operation. Surprisingly, adding an index based on rating does not speed up the query or reduce cost; instead, it raises the overall cost, as seen in the results of INDEX 2. Since this query takes 34 seconds to run, it will probably not be usable in our final product. No method of indexing was able to reduce the time or cost at all.

Query 4:

DEFAULT INDEXING:

```
| -> Limit: 15 row(s)  (actual time=275.505..275.507 rows=15 loops=1)
    -> Sort: ArtistCount DESC, limit input to 15 row(s) per chunk  (actual time=275.504..275.505 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=275.458..275.471 rows=86 loops=1)
            -> Aggregate using temporary table  (actual time=275.452..275.452 rows=86 loops=1)
                -> Nested loop inner join  (cost=38934.31 rows=63902) (actual time=2.718..275.064 rows=99 loops=1)
                    -> Nested loop inner join  (cost=23644.20 rows=63902) (actual time=0.113..142.884 rows=66002 loops=1)
                        -> Nested loop inner join  (cost=8354.09 rows=63902) (actual time=0.107..61.350 rows=66002 loops=1)
                            -> Nested loop inner join  (cost=1169.70 rows=1797) (actual time=0.079..8.874 rows=1795 loops=1)
                                -> Nested loop inner join  (cost=540.82 rows=1797) (actual time=0.068..4.847 rows=1795 loops=1)
                                    -> Covering index scan on Songs using PRIMARY  (cost=101.85 rows=1001) (actual time=0.045..0.451 rows=1001 loops=1)
                                    -> Covering index lookup on HasSongs using FK_songid_HS (songid=Songs.songid)  (cost=0.26 rows=2) (actual time=0.003..0.004 rows=2 loops=1001)
                                -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
                            -> Covering index lookup on ListeningHistories using FK_songid_LH (songid=Songs.songid)  (cost=0.44 rows=36) (actual time=0.020..0.027 rows=37 loops=1795)
                        -> Single-row covering index lookup on UserAccounts using PRIMARY (userid=ListeningHistories.userid)  (cost=0.14 rows=1) (actual time=0.001..0.001 rows=1 loops=66002)
                    -> Single-row covering index lookup on Friends using PRIMARY (userid1=ListeningHistories.userid, userid2=ListeningHistories.userid)  (cost=0.14 rows=1) (actual time=0.002..0.
002 rows=0 loops=66002)
```

INDEX 1: Artistname

```
| -> Limit: 15 row(s)  (actual time=264.186..264.188 rows=15 loops=1)
    -> Sort: ArtistCount DESC, limit input to 15 row(s) per chunk  (actual time=264.185..264.186 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=264.130..264.151 rows=86 loops=1)
            -> Aggregate using temporary table  (actual time=264.125..264.125 rows=86 loops=1)
                -> Nested loop inner join  (cost=38934.31 rows=63902) (actual time=2.496..263.807 rows=99 loops=1)
                    -> Nested loop inner join  (cost=23644.20 rows=63902) (actual time=0.100..137.327 rows=66002 loops=1)
                        -> Nested loop inner join  (cost=8354.09 rows=63902) (actual time=0.094..59.073 rows=66002 loops=1)
                            -> Nested loop inner join  (cost=1169.70 rows=1797) (actual time=0.055..8.228 rows=1795 loops=1)
                                -> Nested loop inner join  (cost=540.82 rows=1797) (actual time=0.048..4.649 rows=1795 loops=1)
                                    -> Covering index scan on Songs using PRIMARY  (cost=101.85 rows=1001) (actual time=0.031..0.431 rows=1001 loops=1)
                                    -> Covering index lookup on HasSongs using FK_songid_HS (songid=Songs.songid)  (cost=0.26 rows=2) (actual time=0.003..0.004 rows=2 loops=1001)
                                -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=1795)
                            -> Covering index lookup on ListeningHistories using FK_songid_LH (songid=Songs.songid)  (cost=0.44 rows=36) (actual time=0.019..0.026 rows=37 loops=1795)
                        -> Single-row covering index lookup on UserAccounts using PRIMARY (userid=ListeningHistories.userid)  (cost=0.14 rows=1) (actual time=0.001..0.001 rows=1 loops=66002)
                    -> Single-row covering index lookup on Friends using PRIMARY (userid1=ListeningHistories.userid, userid2=ListeningHistories.userid)  (cost=0.14 rows=1) (actual time=0.002..0.
002 rows=0 loops=66002)
```

INDEX 2: Songname, Artistname

```
| -> Limit: 15 row(s)  (actual time=208.180..208.182 rows=15 loops=1)
    -> Sort: ArtistCount DESC, limit input to 15 row(s) per chunk  (actual time=208.179..208.180 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=208.129..208.144 rows=86 loops=1)
            -> Aggregate using temporary table  (actual time=208.124..208.124 rows=86 loops=1)
                -> Nested loop inner join  (cost=57479.13 rows=63838) (actual time=106.898..207.967 rows=99 loops=1)
                    -> Nested loop inner join  (cost=35135.90 rows=63838) (actual time=106.889..207.757 rows=99 loops=1)
                        -> Nested loop inner join  (cost=19539.77 rows=35564) (actual time=106.866..207.482 rows=47 loops=1)
                            -> Nested loop inner join  (cost=7092.29 rows=35564) (actual time=106.838..207.357 rows=47 loops=1)
                                -> Nested loop inner join  (cost=1506.00 rows=1250) (actual time=104.633..205.135 rows=1 loops=1)
                                    -> Covering index scan on UserAccounts using PRIMARY  (cost=131.00 rows=1250) (actual time=19.802..33.684 rows=1250 loops=1)
                                    -> Single-row covering index lookup on Friends using PRIMARY (userid1=UserAccounts.userid, userid2=UserAccounts.userid)  (cost=1.00 rows=1) (actual time=0.137
..0.137 rows=0 loops=1250)
                                -> Covering index lookup on ListeningHistories using PRIMARY (userid=UserAccounts.userid)  (cost=1.63 rows=28) (actual time=2.203..2.217 rows=47 loops=1)
                            -> Single-row covering index lookup on Songs using PRIMARY (songid=ListeningHistories.songid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=47)
                        -> Covering index lookup on HasSongs using FK_songid_HS (songid=ListeningHistories.songid)  (cost=0.26 rows=2) (actual time=0.004..0.005 rows=2 loops=47)
                    -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=99)
|
```

INDEX 3: Songname

```
| -> Limit: 15 row(s)  (actual time=208.180..208.182 rows=15 loops=1)
    -> Sort: ArtistCount DESC, limit input to 15 row(s) per chunk  (actual time=208.179..208.180 rows=15 loops=1)
        -> Table scan on <temporary>  (actual time=208.129..208.144 rows=86 loops=1)
            -> Aggregate using temporary table  (actual time=208.124..208.124 rows=86 loops=1)
                -> Nested loop inner join  (cost=57479.13 rows=63838) (actual time=106.898..207.967 rows=99 loops=1)
                    -> Nested loop inner join  (cost=35135.90 rows=63838) (actual time=106.889..207.757 rows=99 loops=1)
                        -> Nested loop inner join  (cost=19539.77 rows=35564) (actual time=106.866..207.482 rows=47 loops=1)
                            -> Nested loop inner join  (cost=7092.29 rows=35564) (actual time=106.838..207.357 rows=47 loops=1)
                                -> Nested loop inner join  (cost=1506.00 rows=1250) (actual time=104.633..205.135 rows=1 loops=1)
                                    -> Covering index scan on UserAccounts using PRIMARY  (cost=131.00 rows=1250) (actual time=19.802..33.684 rows=1250 loops=1)
                                    -> Single-row covering index lookup on Friends using PRIMARY (userid1=UserAccounts.userid, userid2=UserAccounts.userid)  (cost=1.00 rows=1) (actual time=0.137
..0.137 rows=0 loops=1250)
                                -> Covering index lookup on ListeningHistories using PRIMARY (userid=UserAccounts.userid)  (cost=1.63 rows=28) (actual time=2.203..2.217 rows=47 loops=1)
                            -> Single-row covering index lookup on Songs using PRIMARY (songid=ListeningHistories.songid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=47)
                        -> Covering index lookup on HasSongs using FK_songid_HS (songid=ListeningHistories.songid)  (cost=0.26 rows=2) (actual time=0.004..0.005 rows=2 loops=47)
                    -> Single-row index lookup on Artists using PRIMARY (artistid=HasSongs.artistid)  (cost=0.25 rows=1) (actual time=0.002..0.002 rows=1 loops=99)
|
```

FINAL STRATEGY: Default Indexing

Once again, this query is not affected by any indices. Trying different combinations of the artistname and songname index which were effective in the past yield no results and are strictly ignored by this query. Again, the bulk of the cost and time comes from joining tables based on userid, which is a primary key and already indexed. Unlike query 3, this query performs well and will be usable in our final product. Overall, for lack of a better option, we will use the default indexing on this table.