Stage 3: Mood Diary App

4/1/2025

Edited 4/24/2025

Team049-WeDeLiver

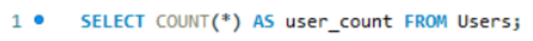
Megan Tang, Sydney Yu, Yu-Liang Lin, Bo-Syuan Hou

Data Definition Language(DDL)

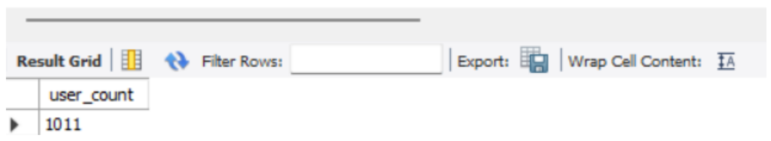
```
CREATE TABLE User (
    ID INT AUTO_INCREMENT PRIMARY KEY,
    Name VARCHAR(255) NOT NULL,
    Consecutivedays INT NOT NULL,
    LastLoginDate DATE,
);
CREATE TABLE MoodHealth (
    UserID INT NOT NULL,
    Date DATE,
    StressLevel INT NOT NULL,
    AnxietyLevel INT NOT NULL,
    SleepHours FLOAT NOT NULL,
    MoodScore INT NOT NULL,
    PRIMARY KEY (UserID, Date),
    FOREIGN KEY (UserID) REFERENCES User(ID) ON DELETE CASCADE
);
CREATE TABLE Playlist (
    PlaylistID INT PRIMARY KEY AUTO_INCREMENT,
    UserID INT NOT NULL,
    Date DATE NOT NULL,
    FOREIGN KEY (UserID) REFERENCES User(ID) ON DELETE CASCADE
);
CREATE TABLE Song (
    SongID INT PRIMARY KEY,
    Artists VARCHAR(255) NOT NULL,
    AlbumName VARCHAR(255) NOT NULL,
    TrackName VARCHAR(255) NOT NULL,
    Energy DECIMAL(5,2) NOT NULL,
    Loudness DECIMAL(5,2) NOT NULL,
    I DECIMAL(5,2) NOT NULL,
    Acousticness DECIMAL(5,2) NOT NULL,
    Liveness DECIMAL(5,2) NOT NULL,
    Valence DECIMAL(5,2) NOT NULL
);
CREATE TABLE Playlist2Song (
    PlaylistID INT NOT NULL,
    SongID INT NOT NULL,
    PRIMARY KEY (PlaylistID, SongID),
    FOREIGN KEY (PlaylistID) REFERENCES Playlist(PlaylistID) ON DELETE CASCADE,
    FOREIGN KEY (SongID) REFERENCES Song(SongID) ON DELETE CASCADE
);
```

Row count

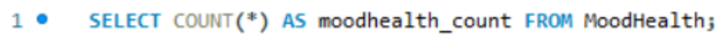
We have 5 tables with over 1000 rows: Users, MoodHealth, Playlist, Song, PlayList2Song. This was already true for the first submission, but I ran the count queries again for clarity. They are now more rows because of additional entries from running our application.



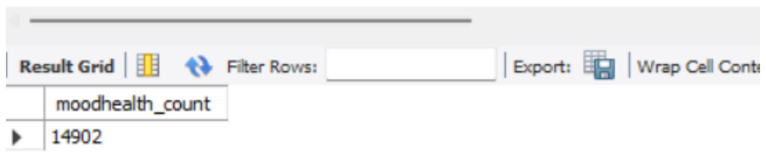
```
SELECT COUNT(*) AS user_count FROM Users;
```



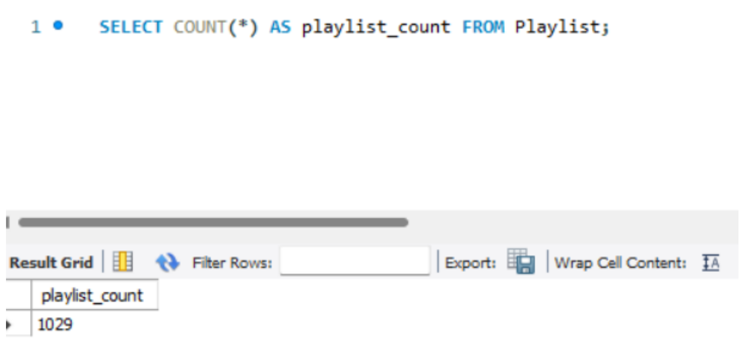
```
user_count
1011
```



```
SELECT COUNT(*) AS moodhealth_count FROM MoodHealth;
```



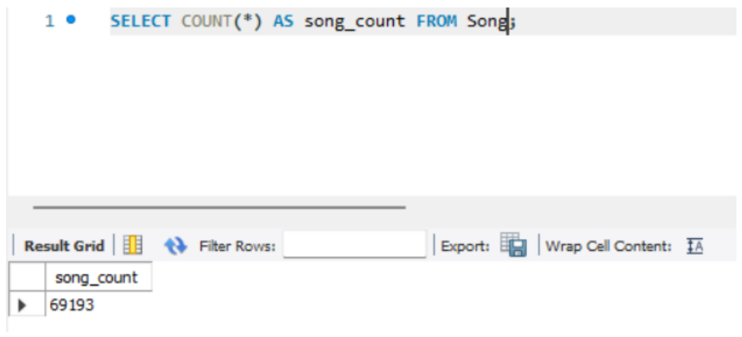
```
moodhealth_count
14902
```



```
SELECT COUNT(*) AS playlist_count FROM Playlist;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

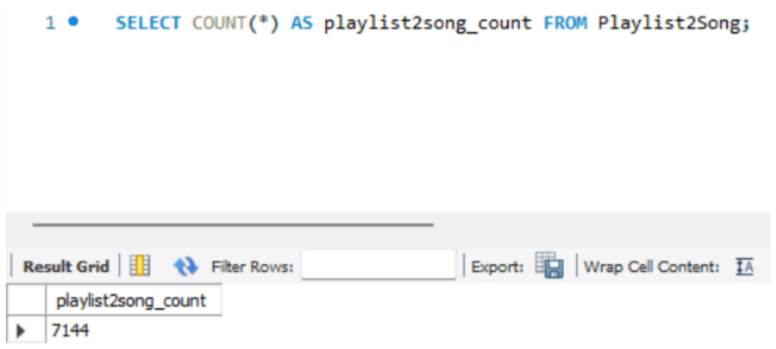
```
playlist_count
1029
```



```
SELECT COUNT(*) AS song_count FROM Song;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

```
song_count
69193
```



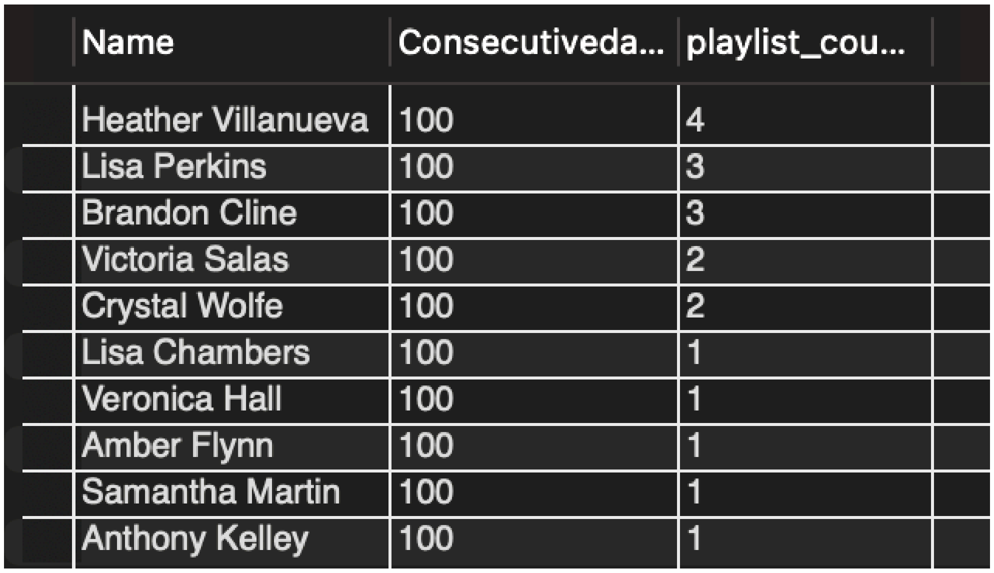
```
SELECT COUNT(*) AS playlist2song_count FROM Playlist2Song;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

```
playlist2song_count
7144
```

Advanced Query 1

Purpose:
We have a Leaderboard feature. This query select the top ten users that have the most consecutive login days, and generate the most playlists.
Query:
SELECT u.Name, u.Consecutivedays, COUNT(p.PlaylistID) AS playlist_count
FROM CS411.Users u
LEFT JOIN CS411.Playlist p ON u.ID = p.UserID
GROUP BY u.ID, u.Name, u.Consecutivedays
ORDER BY u.Consecutivedays DESC, playlist_count DESC
LIMIT 10;
Output:



Name	Consecutiveda...	playlist_cou...
Heather Villanueva	100	4
Lisa Perkins	100	3
Brandon Cline	100	3
Victoria Salas	100	2
Crystal Wolfe	100	2
Lisa Chambers	100	1
Veronica Hall	100	1
Amber Flynn	100	1
Samantha Martin	100	1
Anthony Kelley	100	1

Indexing:

Original

```
-> Limit: 10 row(s)  (actual time=10.1..10.1 rows=10 loops=1)
    -> Sort: CS411.u.Consecutivedays DESC, playlist_count DESC, limit input to 10 row(s)
per chunk  (actual time=10.1..10.1 rows=10 loops=1)
        -> Table scan on <temporary>  (actual time=9.55..9.79 rows=1000 loops=1)
            -> Aggregate using temporary table  (actual time=9.55..9.55 rows=1000 loops=1)
                -> Nested loop left join  (cost=517 rows=1658) (actual time=0.0924..6.9
rows=1364 loops=1)
                    -> Table scan on u  (cost=101 rows=1000) (actual time=0.0739..1.21
rows=1000 loops=1)
                    -> Covering index lookup on p using UserID (UserID=CS411.u.ID)  (cost=0.25
rows=1.66) (actual time=0.00344..0.00533 rows=1 loops=1000)
```

Added CREATE INDEX idx_user_consecutivedays ON User(Consecutivedays);

```
-> Limit: 10 row(s)  (actual time=8.72..8.72 rows=10 loops=1)
    -> Sort: CS411.u.Consecutivedays DESC, playlist_count DESC, limit input to 10 row(s)
per chunk  (actual time=8.72..8.72 rows=10 loops=1)
        -> Table scan on <temporary>  (actual time=8.21..8.42 rows=1000 loops=1)
            -> Aggregate using temporary table  (actual time=8.21..8.21 rows=1000 loops=1)
                -> Nested loop left join  (cost=517 rows=1658) (actual time=0.0919..5.64
rows=1364 loops=1)
                    -> Table scan on u  (cost=101 rows=1000) (actual time=0.0741..0.863
rows=1000 loops=1)
                    -> Covering index lookup on p using UserID (UserID=CS411.u.ID)  (cost=0.25
rows=1.66) (actual time=0.00386..0.00448 rows=1 loops=1000)
```

Added CREATE INDEX idx_user_consecutivedays_id ON User(Consecutivedays, ID);

```
-> Limit: 10 row(s)  (actual time=11.1..11.1 rows=10 loops=1)
    -> Sort: CS411.u.Consecutivedays DESC, playlist_count DESC, limit input to 10 row(s) per chunk  (actual time=11.1..11.1 rows=10 loops=1)
        -> Table scan on <temporary>  (actual time=10.6..10.8 rows=1000 loops=1)
            -> Aggregate using temporary table  (actual time=10.6..10.6 rows=1000 loops=1)
                -> Nested loop left join  (cost=517 rows=1658) (actual time=0.0921..5.17 rows=1364 loops=1)
                    -> Table scan on u  (cost=101 rows=1000) (actual time=0.0752..0.723 rows=1000 loops=1)
                    -> Covering index lookup on p using UserID (UserID=CS411.u.ID)  (cost=0.25 rows=1.66) (actual time=0.00337..0.00407 rows=1 loops=1000)
```

Added CREATE INDEX idx_playlist_user_include_pid ON Playlist(UserID) INCLUDE (PlaylistID);

```
-> Limit: 10 row(s)  (actual time=9.8..9.81 rows=10 loops=1)
    -> Sort: CS411.u.Consecutivedays DESC, playlist_count DESC, limit input to 10 row(s) per chunk  (actual time=9.8..9.8 rows=10 loops=1)
        -> Table scan on <temporary>  (actual time=9.07..9.32 rows=1000 loops=1)
            -> Aggregate using temporary table  (actual time=9.07..9.07 rows=1000 loops=1)
                -> Nested loop left join  (cost=517 rows=1658) (actual time=0.686..5.31 rows=1364 loops=1)
                    -> Table scan on u  (cost=101 rows=1000) (actual time=0.604..1.33 rows=1000 loops=1)
                    -> Covering index lookup on p using UserID (UserID=CS411.u.ID)  (cost=0.25 rows=1.66) (actual time=0.00304..0.00368 rows=1 loops=1000)
```

Decision:
The cost values are identical for all the different indexes. This may be because the query creates a temporary table for aggregation, becoming the dominant cost factor. The optimizer may also be using fixed estimates for the number of users (1000 rows), and are not updated with new indexes. However, logically, the best index choice is CREATE INDEX idx_playlist_user_include_pid ON Playlist(UserID) INCLUDE (PlaylistID); because it is the bottleneck join between User and Playlist, eliminating the need to access the base Playlist table. This approach is more effective than indexes on User table columns because it optimizes the query's dominant operation, and its benefits would be more significant with larger datasets.

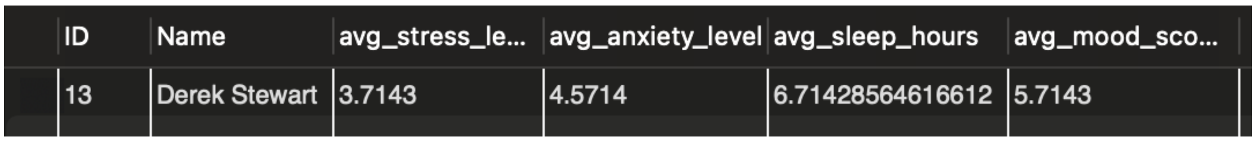
Advanced Query 2

Purpose: Given a user ID, select their name and the latest 7 mood health average score.

Query:

```
SELECT u.ID, u.Name,
       AVG(latest_mood.StressLevel) AS avg_stress_level,
       AVG(latest_mood.AnxietyLevel) AS avg_anxiety_level,
       AVG(latest_mood.SleepHours) AS avg_sleep_hours,
       AVG(latest_mood.MoodScore) AS avg_mood_score
FROM (
    SELECT mh.UserID, mh.StressLevel, mh.AnxietyLevel, mh.SleepHours, mh.MoodScore
    FROM CS411.MoodHealth mh
    WHERE mh.UserID = 13
    ORDER BY mh.Date DESC
    LIMIT 7
) AS latest_mood
JOIN CS411.Users u ON u.ID = latest_mood.UserID
GROUP BY u.ID, u.Name;
```

Output:



ID	Name	avg_stress_le...	avg_anxiety_level	avg_sleep_hours	avg_mood_sco...
13	Derek Stewart	3.7143	4.5714	6.71428564616612	5.7143

Indexing:

Original

```
-> Table scan on <temporary>  (actual time=1.6..1.6 rows=1 loops=1)
-> Aggregate using temporary table  (actual time=1.6..1.6 rows=1 loops=1)
-> Nested loop inner join  (cost=5 rows=4) (actual time=0.807..0.81 rows=4 loops=1)
-> Table scan on latest_mood  (cost=1.69..3.6 rows=4) (actual time=0.778..0.779 rows=4
loops=1)
-> Materialize  (cost=1.05..1.05 rows=4) (actual time=0.776..0.776 rows=4 loops=1)
-> Limit: 7 row(s)  (cost=0.652 rows=4) (actual time=0.0863..0.678 rows=4 loops=1)
-> Index lookup on mh using PRIMARY (UserID=13) (reverse)  (cost=0.652 rows=4)
(actual time=0.0851..0.676 rows=4 loops=1)
-> Single-row index lookup on u using PRIMARY (ID=latest_mood.UserID)  (cost=0.275
rows=1) (actual time=0.00685..0.0069 rows=1 loops=4)
```

Added CREATE INDEX idx_mh_covering ON MoodHealth(Date DESC, StressLevel, AnxietyLevel, SleepHours, MoodScore

```
-> Table scan on <temporary>  (actual time=0.0822..0.0825 rows=1 loops=1)     ->
Aggregate using temporary table  (actual time=0.0816..0.0816 rows=1 loops=1)         ->
Nested loop inner join  (cost=7.7 rows=7) (actual time=0.0547..0.0583 rows=7 loops=1)
-> Table scan on latest_mood  (cost=3.03..5.25 rows=7) (actual time=0.0449..0.0459
rows=7 loops=1)                 -> Materialize  (cost=2.66..2.66 rows=7) (actual
time=0.0434..0.0434 rows=7 loops=1)                    -> Limit: 7 row(s)  (cost=1.96 rows=7)
(actual time=0.0274..0.0302 rows=7 loops=1)                       -> Index lookup on mh using
PRIMARY (UserID=13) (reverse)  (cost=1.96 rows=17) (actual time=0.0265..0.0288
rows=7 loops=1)             -> Single-row index lookup on u using PRIMARY
(ID=latest_mood.UserID)  (cost=0.264 rows=1) (actual time=0.00132..0.00135 rows=1
loops=7)
```

Added CREATE INDEX idx_mh_date ON CS411.MoodHealth(Date DESC);

```
-> Table scan on <temporary>  (actual time=0.166..0.166 rows=1 loops=1)
    -> Aggregate using temporary table  (actual time=0.165..0.165 rows=1 loops=1)
        -> Nested loop inner join  (cost=7.7 rows=7) (actual time=0.127..0.131 rows=7 loops=1)
            -> Table scan on latest_mood  (cost=3.03..5.25 rows=7) (actual time=0.0577..0.0592
rows=7 loops=1)
                -> Materialize  (cost=2.66..2.66 rows=7) (actual time=0.0561..0.0561 rows=7
loops=1)
                    -> Limit: 7 row(s)  (cost=1.96 rows=7) (actual time=0.036..0.0388 rows=7 loops=1)
                        -> Index lookup on mh using PRIMARY (UserID=13) (reverse)  (cost=1.96
rows=17) (actual time=0.0352..0.0375 rows=7 loops=1)
            -> Single-row index lookup on u using PRIMARY (ID=latest_mood.UserID)  (cost=0.264
rows=1) (actual time=0.00983..0.00987 rows=1 loops=7)
```

```
CREATE INDEX idx_mh_recent_user ON CS411.MoodHealth(Date DESC)
WHERE Date > CURRENT_DATE - INTERVAL '30 days';
```

```
-> Table scan on <temporary>  (actual time=0.0819..0.0821 rows=1 loops=1)     ->
Aggregate using temporary table  (actual time=0.0812..0.0812 rows=1 loops=1)         ->
Nested loop inner join  (cost=7.7 rows=7) (actual time=0.0532..0.0566 rows=7 loops=1)
-> Table scan on latest_mood  (cost=3.03..5.25 rows=7) (actual time=0.0436..0.0446
rows=7 loops=1)                 -> Materialize  (cost=2.66..2.66 rows=7) (actual
time=0.0421..0.0421 rows=7 loops=1)                    -> Limit: 7 row(s)  (cost=1.96 rows=7)
(actual time=0.0265..0.0294 rows=7 loops=1)                       -> Index lookup on mh using
PRIMARY (UserID=13) (reverse)  (cost=1.96 rows=17) (actual time=0.0256..0.0279
rows=7 loops=1)             -> Single-row index lookup on u using PRIMARY
(ID=latest_mood.UserID)  (cost=0.264 rows=1) (actual time=0.0013..0.00134 rows=1
loops=7)
```

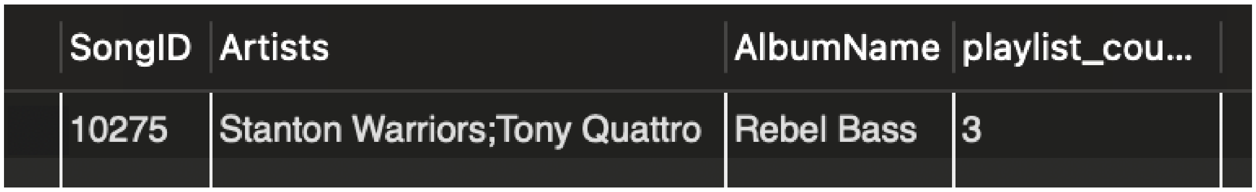
Decision:
The optimal indexing strategy is using CREATE INDEX idx_mh_covering ON MoodHealth(Date DESC, StressLevel, AnxietyLevel, SleepHours, MoodScore because it contains all the columns

needed by the subquery, filtering date and aggregation metrics. It enables efficient filtering of the dates, maintains order using DESC.
Despite identical cost metrics, this design provides important benefits and also has the fastest execution time (0.082ms vs 1.6ms baseline)

<u>Advanced Query 3</u>:

Purpose: Select the song that appears in users' playlists the most often.
Query:
```
SELECT s.SongID, s.Artists, s.AlbumName, COUNT(*) AS playlist_count
FROM Song s JOIN Playlist2Song ps ON s.SongID = ps.SongID
JOIN Playlist p ON ps.PlaylistID = p.PlaylistID
GROUP BY s.SongID, s.Artists, s.AlbumName
ORDER BY playlist_count DESC LIMIT 1;
```

Output:



	SongID	Artists	AlbumName	playlist_cou...
	10275	Stanton Warriors;Tony Quattro	Rebel Bass	3

Indexing:

Original

```
-> Limit: 1 row(s)  (actual time=1808..1808 rows=1 loops=1)
    -> Sort: playlist_count DESC, limit input to 1 row(s) per chunk  (actual time=1808..1808
rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=1801..1805 rows=9440 loops=1)
            -> Aggregate using temporary table  (actual time=1801..1801 rows=9440 loops=1)
                -> Nested loop inner join  (cost=4999 rows=10318) (actual time=0.0970..1471
rows=9980 loops=1)
                    -> Nested loop inner join  (cost=1387 rows=10318) (actual time=0.0841..1128
rows=9980 loops=1)
                        -> Covering index scan on p using UserID  (cost=101 rows=1000) (actual
time=0.0564..0.827 rows=1000 loops=1)
                        -> Covering index lookup on ps using PRIMARY (PlaylistID=p.PlaylistID)
(cost=0.256 rows=10.3) (actual time=1.11..1.12 rows=9.98 loops=1000)
                    -> Single-row index lookup on s using PRIMARY (SongID=ps.SongID)  (cost=0.25
rows=1) (actual time=0.0341..0.0341 rows=1 loops=9980)
```

```
ADD CREATE INDEX idx_playlist2song_songid ON Playlist2Song(SongID);
CREATE INDEX idx_playlist2song_playlistid ON Playlist2Song(PlaylistID);
```

```
-> Limit: 1 row(s)  (actual time=1507..1507 rows=1 loops=1)
    -> Sort: playlist_count DESC, limit input to 1 row(s) per chunk  (actual time=1507..1507
rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=1501..1504 rows=9440 loops=1)
            -> Aggregate using temporary table  (actual time=1501..1501 rows=9440 loops=1)
                -> Nested loop inner join  (cost=4999 rows=10318) (actual time=0.0808..1226
rows=9980 loops=1)
                    -> Nested loop inner join  (cost=1387 rows=10318) (actual time=0.0701..940
rows=9980 loops=1)
                        -> Covering index scan on p using UserID  (cost=101 rows=1000) (actual
time=0.047..0.689 rows=1000 loops=1)
                        -> Covering index lookup on ps using PRIMARY (PlaylistID=p.PlaylistID)
(cost=0.256 rows=10.3) (actual time=0.927..0.933 rows=9.98 loops=1000)
                    -> Single-row index lookup on s using PRIMARY (SongID=ps.SongID)  (cost=0.25
rows=1) (actual time=0.0284..0.0284 rows=1 loops=9980)
```

ADD CREATE INDEX idx_ps_covering ON Playlist2Song(SongID, PlaylistID);
CREATE INDEX idx_playlist_covering ON Playlist(PlaylistID) INCLUDE (UserID, Date);

```
-> Limit: 1 row(s)  (actual time=1356..1356 rows=1 loops=1)
    -> Sort: playlist_count DESC, limit input to 1 row(s) per chunk  (actual time=1356..1356
rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=1351..1354 rows=9440 loops=1)
            -> Aggregate using temporary table  (actual time=1351..1351 rows=9440 loops=1)
                -> Nested loop inner join  (cost=4999 rows=10318) (actual time=0.0727..1103
rows=9980 loops=1)
                    -> Nested loop inner join  (cost=1387 rows=10318) (actual time=0.0631..846
rows=9980 loops=1)
                        -> Covering index scan on p using UserID  (cost=101 rows=1000) (actual
time=0.0423..0.620 rows=1000 loops=1)
                        -> Covering index lookup on ps using PRIMARY (PlaylistID=p.PlaylistID)
(cost=0.256 rows=10.3) (actual time=0.834..0.840 rows=9.98 loops=1000)
                    -> Single-row index lookup on s using PRIMARY (SongID=ps.SongID)  (cost=0.25
rows=1) (actual time=0.0256..0.0256 rows=1 loops=9980)
```

ADD CREATE INDEX idx_ps_songid_count ON Playlist2Song(SongID);
CREATE INDEX idx_song_covering ON Song(SongID, Artists, AlbumName);

```
-> Limit: 1 row(s)  (actual time=1425..1425 rows=1 loops=1)
    -> Sort: playlist_count DESC, limit input to 1 row(s) per chunk  (actual time=1425..1425
rows=1 loops=1)
        -> Table scan on <temporary>  (actual time=1412..1418 rows=9440 loops=1)
```

```
            -> Aggregate using temporary table  (actual time=1412..1412 rows=9440 loops=1)
                -> Nested loop inner join  (cost=4999 rows=10318) (actual time=0.085..1158
rows=9980 loops=1)
                    -> Nested loop inner join  (cost=1387 rows=10318) (actual time=0.072..872
rows=9980 loops=1)
                        -> Covering index scan on p using UserID  (cost=101 rows=1000) (actual
time=0.051..0.732 rows=1000 loops=1)
                        -> Covering index lookup on ps using PRIMARY (PlaylistID=p.PlaylistID)
(cost=0.256 rows=10.3) (actual time=0.863..0.871 rows=9.98 loops=1000)
                    -> Single-row index lookup on s using PRIMARY (SongID=ps.SongID)  (cost=0.25
rows=1) (actual time=0.028..0.028 rows=1 loops=9980)
```

Decision:
The best logical choice is to use CREATE INDEX idx_ps_covering ON Playlist2Song(SongID, PlaylistID) because it optimizes both joins in the query of Playlist2Song, Song, and Playlist tables. While the costs show identical values, the index has a lower execution time. The composite structure outperforms single-column indexing because it satisfies both SongID and PlaylistID lookups, avoiding the storage overhead of including unnecessary columns.

Advanced Query 4:

Purpose: Insert the top 10 songs closest to the calculated mood score into a playlist. UserID = 111 is used temporarily to run the query in mySQL, but for our final application it would use the actual userID

Query:
```
SET @in_userId = 111;
  -- Step 1: Create a new playlist
  INSERT INTO Playlist (UserID, Date)
  VALUES (@in_userId, CURDATE());
  SET @newPlaylistId = LAST_INSERT_ID();

  -- Step 2: Compute mood averages
  SET @avg_stress = (
    SELECT AVG(StressLevel)
    FROM (
      SELECT StressLevel
      FROM MoodHealth
      WHERE UserID = @in_userId
      ORDER BY Date DESC
      LIMIT 7
    ) AS recent
  );
```

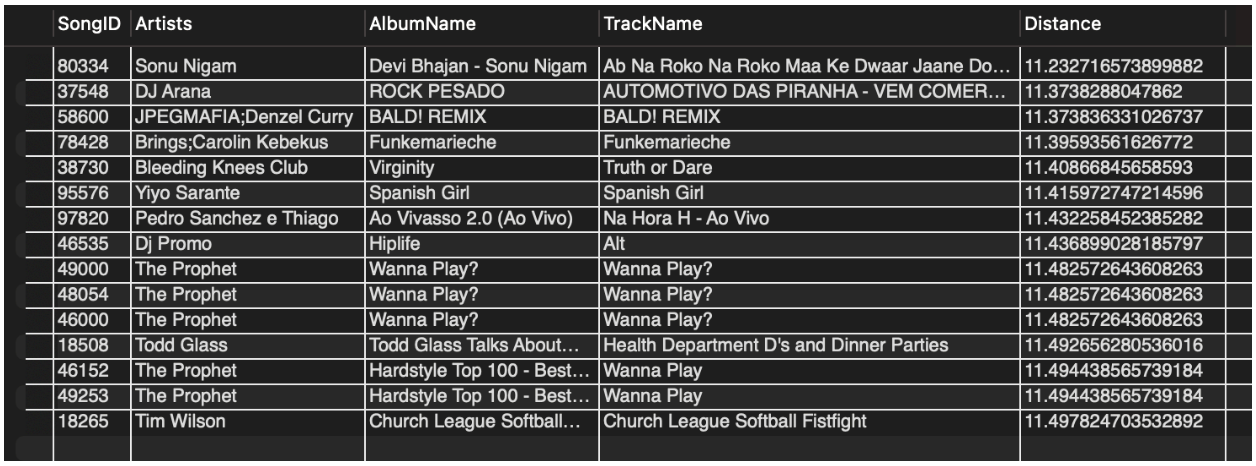
```
SET @avg_anxiety = (
   SELECT AVG(AnxietyLevel)
   FROM (
      SELECT AnxietyLevel
      FROM MoodHealth
      WHERE UserID = @in_userId
      ORDER BY Date DESC
      LIMIT 7
   ) AS recent
);

SET @avg_sleep = (
   SELECT AVG(SleepHours)
   FROM (
      SELECT SleepHours
      FROM MoodHealth
      WHERE UserID = @in_userId
      ORDER BY Date DESC
      LIMIT 7
   ) AS recent
);

SET @avg_mood = (
   SELECT AVG(MoodScore)
   FROM (
      SELECT MoodScore
      FROM MoodHealth
      WHERE UserID = @in_userId
      ORDER BY Date DESC
      LIMIT 7
   ) AS recent
);
```
(EXPLAIN ANALYZE would go here)
```
-- Step 3: Get max values
SELECT
   10, 10, MAX(SleepHours), MAX(MoodScore)
INTO
   @max_stress, @max_anxiety, @max_sleep, @max_mood
FROM MoodHealth;

-- Step 4: Insert top 10 recommended songs into Playlist2Song
INSERT INTO Playlist2Song (PlaylistID, SongID)
SELECT @newPlaylistId, ranked.SongID
FROM (
    SELECT
      SongID,
      SQRT(
        POWER((Loudness - @avg_stress / @max_stress), 2) +
        POWER((Energy - @avg_anxiety / @max_anxiety), 2) +
        POWER((Valence - @avg_anxiety / @max_anxiety), 2) +
        POWER((Acousticness - @avg_sleep / @max_sleep), 2) +
        POWER((Mode - @avg_sleep / @max_sleep), 2) +
        POWER((Liveness - @avg_mood / @max_mood), 2)
      ) AS Distance
    FROM (
      SELECT SongID, Loudness, Energy, Valence, Acousticness, Mode, Liveness
FROM Song
WHERE
 SongID >= FLOOR(RAND() * (SELECT MAX(SongID) FROM Song)) AND
 SongID NOT IN (
  SELECT SongID
  FROM Playlist2Song p2s
  JOIN Playlist p ON p2s.PlaylistID = p.PlaylistID
  WHERE p.UserID = @in_userId
 )
LIMIT 1000
    ) AS s
    ORDER BY Distance ASC
    LIMIT 10
 ) AS ranked;
```

Output:



SongID	Artists	AlbumName	TrackName	Distance
80334	Sonu Nigam	Devi Bhajan - Sonu Nigam	Ab Na Roko Na Roko Maa Ke Dwaar Jaane Do…	11.232716573899882
37548	DJ Arana	ROCK PESADO	AUTOMOTIVO DAS PIRANHA - VEM COMER…	11.3738288047862
58600	JPEGMAFIA;Denzel Curry	BALD! REMIX	BALD! REMIX	11.373836331026737
78428	Brings;Carolin Kebekus	Funkemarieche	Funkemarieche	11.39593561626772
38730	Bleeding Knees Club	Virginity	Truth or Dare	11.40866845658593
95576	Yiyo Sarante	Spanish Girl	Spanish Girl	11.415972747214596
97820	Pedro Sanchez e Thiago	Ao Vivasso 2.0 (Ao Vivo)	Na Hora H - Ao Vivo	11.432258452385282
46535	Dj Promo	Hiplife	Alt	11.436899028185797
49000	The Prophet	Wanna Play?	Wanna Play?	11.482572643608263
48054	The Prophet	Wanna Play?	Wanna Play?	11.482572643608263
46000	The Prophet	Wanna Play?	Wanna Play?	11.482572643608263
18508	Todd Glass	Todd Glass Talks About…	Health Department D's and Dinner Parties	11.492656280536016
46152	The Prophet	Hardstyle Top 100 - Best…	Wanna Play	11.494438565739184
49253	The Prophet	Hardstyle Top 100 - Best…	Wanna Play	11.494438565739184
18265	Tim Wilson	Church League Softball…	Church League Softball Fistfight	11.497824703532892

Indexing:

Original

```
-> Aggregate: max(MoodHealth.SleepHours), max(MoodHealth.MoodScore)  (cost=2984
rows=1) (actual time=6.69..6.7 rows=1 loops=1)     -> Table scan on MoodHealth
(cost=1504 rows=14797) (actual time=0.0427..4.99 rows=14902 loops=1)
```

```
CREATE INDEX idx_moodhealth_user_date ON MoodHealth(UserID, Date DESC);
```

```
-> Aggregate: max(MoodHealth.SleepHours), max(MoodHealth.MoodScore)  (cost=2984
rows=1) (actual time=6.1..6.1 rows=1 loops=1)     -> Table scan on MoodHealth
(cost=1504 rows=14797) (actual time=0.0653..4.4 rows=14902 loops=1)
```

```
CREATE INDEX idx_song_features ON Song(SongID, Loudness, Energy, Valence,
Acousticness, Mode, Liveness);
```

```
-> Aggregate: max(MoodHealth.SleepHours), max(MoodHealth.MoodScore)  (cost=2984
rows=1) (actual time=5.61..5.61 rows=1 loops=1)     -> Table scan on MoodHealth
(cost=1504 rows=14797) (actual time=0.0447..4.13 rows=14902 loops=1)
```

```
CREATE INDEX idx_moodhealth_user_recent_metrics ON MoodHealth(UserID, Date DESC,
StressLevel, AnxietyLevel, SleepHours, MoodScore)
WHERE Date > CURRENT_DATE - INTERVAL '90 days';
```

```
-> Aggregate: max(MoodHealth.SleepHours), max(MoodHealth.MoodScore)  (cost=2984
rows=1) (actual time=6.93..6.93 rows=1 loops=1)     -> Table scan on MoodHealth
(cost=1504 rows=14797) (actual time=0.0417..5.04 rows=14902 loops=1)
```

Decision:

The best strategy combines idx_moodhealth_user_date for fast UserID/Date filtering and idx_song_features to optimize distance calculations. While no index improved the MAX() scan, which always reads the full table, these two indexes speed up the most critical parts: fetching recent mood data and ranking songs. Avoid over-indexing didn't help, and the MAX() operation inherently requires a full scan.