Project Report

Our final project was very similar to our original proposal and was basically what we visioned from the start. The definition of a "fantasy league" changed a bit during the time, since going in different members had different ideas of what a fantasy league meant. We eventually settled on one groupmates idea to use the model of a Premier soccer fantasy league.

The application satisfied all functionalities that we wanted so it is quite useful. The one issue with usefulness is that we used fake artist names and data, so it is not so realistic. However, this is a fairly easy change to retrieve real life data. Otherwise, I would say this is an overwhelming success! As far as we can tell, there are no really big bugs or errors that we are prone to, and we have the major functionalities in place.

Originally, our plan was for the data to be from the Spotify API, but we ran into some problems with queries there and converting it into usable data. For that reason, we decided to generate realistic data based on Spotify, to show as a proof of concept.

We also kept our ER diagram and table implementations mostly the same. For the purposes of the demo however, we added a time feature since the nature of our project is that it needs to be played over the course of a long time, so to show how points are awarded over time, we added a "Game Setting" to the database to keep track of what time each game is at and so we can fast forward. The original design is more suitable for production code when we have the real clock going, but for demo purposes, the new change is fine.

We were initially considering adding in roster trading, but it seems to be a bit outside the scope of this class. It is quite complicated and we couldn't figure out how to implement it correctly. Given more time, that is one of the first features we would love to add.

Our application uses several advanced database programs including triggers, stored procedures, and transactions to maintain data integrity, enforce constraints, and automate complex operations.

We implemented multiple triggers to automate important updates:

- When a new artist statistic is inserted, the after_artist_stats_update trigger automatically updates the corresponding rosters' points by calling a stored procedure CalculateRosterPoints.

- The before_roster_insert trigger prevents a player from creating multiple rosters in the same league by checking existing entries and raising an error if necessary.

- The after_roster_delete trigger updates the league's player count when a roster is deleted, maintaining consistency between tables.

- The price_change_monitor trigger calculates the percentage change in an artist's price compared to the previous month, which can be useful for monitoring artist value trends.

For stored procedures, we created several to manage complex operations efficiently:

- AddArtistToRoster and RemoveArtistFromRoster procedures ensure that adding or removing an artist updates both the roster member table and adjusts the user's budget correctly, using transactions to guarantee atomicity.

- CalculateRosterPoints computes the total points for a roster based on artist statistics and updates the roster record accordingly.

- UpdateAllRosterPoints iterates through all rosters to refresh their points, useful after a batch update of artist stats.

- GetLeagueStandings generates a leaderboard for a given league, summing up players' points in an efficient, reusable query.

We used **transactions** in all operations that involved modifying multiple tables together, such as when deleting a player, joining or leaving a league, adding or removing artists from a roster, and advancing the game month. This ensures that the database remains in a consistent state even if an error occurs during a multi-step operation.

For advanced queries, we incorporated multiple complex SQL operations:

- Joins across multiple relations: Many endpoints, such as /api/standings, /api/leagues, and /api/rosters, involve joins between tables like Roster, League, and Player to fetch aggregated information for user display.

- Aggregation via GROUP BY: The league standings and roster point calculations use GROUP BY to summarize points across different rosters and players efficiently.

- Subqueries: For example, when fetching the latest artist stats, we used subqueries inside EXISTS clauses and ranking logic (RANK() OVER (PARTITION BY...)) to ensure we are always retrieving the most recent and relevant statistics, which cannot be easily replaced by simple joins.

These advanced database programs directly enhance the utility of the application by supporting critical features such as roster management, budget tracking, artist transactions, and real-time league updates.

One technical challenge each team member encountered:

- Shreyes: We were having an issue where the database would not allow certain IP addresses to connect. I was racking my brain, because it appeared as a connection error and everything looked right with code. I did a lot of testing to figure out what could be causing the bug, and I realized that on the GCP website, you need to add approved IP addresses. What I did was add a generic IP address which would allow the database to be essentially public. This solved the error.

- Chris: Interacting with the database caused me some frustration, from league.leagueid being dependent on roster.leagueid (it should be vice versa) to different ids not auto incrementing properly, there was a lot of unforeseen issues in setting up the database. Make sure you plan your DDL (create DB) commands well and be prepared to use commands such as describe and alter to diagnose issues with your schema and subsequently provide the related changes.

- Cyrus: This was my first time working on a full-stack project, and I was not familiar with how the frontend and backend communicate. I spent time reading documentation to better understand the connection between them, and eventually built out the overall project structure.

- David: Move local SQL to GCP. We initially wanted to store our SQL locally. However, this resulted in many issues such as different users not being in sync. We initially tried to sync it up but it didn't work well. Therefore, we decided to have one common database on GCP. For setting up a GCP database, we needed multiple scripts so that we had the necessary constraints and triggers, as well as generated random data. I think next time we will start by using GCP.

Future work: The first thing to do would be to implement the Spotify API so we can use real data. This would not be terribly difficult, but would take some time. The next step would be to implement a system where you can trade people between rosters in a league. That is a really fun aspect of things like fantasy football, and I would love to see it here. Furthermore, a UI overhaul would be nice. Our current UI isn't bad, but it is not really at production level.

Division of labor:
- Shreyes: GCP Setup, IP, Project report, demo presenter
- Chris: Implement league create/join, view/edit players, demo video script, demo presenter
- Cyrus: Setup project structure, project report, debugging the backend, create tables, implement the triggers and stored procedure
- David: Setup GCP SQL database, project report, league standings, adding artists to roster, automatically creating roster when creating/joining league, demo recorder