Title and Project Summary
Title: SQLMate

Summary:

SQLMate is a web-based API designed to make querying our structured Music database accessible to non-technical users by abstracting SQL into an intuitive, visual, and no-code query builder. Through a collection of drag-and-drop elements, checkboxes, and other interactive UI components that dynamically reveal related tables based on foreign key relationships, users can construct queries without writing SQL.

On the backend, SQLMate will use a custom query generation algorithm to convert the UI selections into an SQL query which produces the result that will be sent back to the user. After generating the results, users can download the table/dataset and save it to their account for future modifications and interactions.

Description

The main goal of this project is to make data exploration more accessible to a wider audience so they can easily pursue their research goals. Additionally, our extensive collection of data features related to Music will save users a lot of time compared to other means of data preparation. For example, if a user is looking for a single dataset containing the top one hundred songs by streaming numbers and also details of the artist, it would be easier to use our application as opposed to assembling their own dataset using multiple external sources and using another tool to join the information.

As stated in the summary, we are trying to create a layer of abstraction on top of SQL which retains the power of SQL on structured databases, but without the complexity. This can be achieved by carefully designing a smart UI that, when combined with a metadata layer that describes relationships between tables, can capture the necessary information required for the backend algorithm to generate an equivalent SQL query that reflects the user's intentions.

Creative Component

One technically challenging feature that would elevate the user experience is a preview section that displays the result of the current set of UI selections if they were to execute them then. This would allow the user to see if they are on the right track to create what they want and, if not, use the output to adjust their approach. Once the preview section accurately displays what they want, they can easily save and/or download it with the confidence that it is as they desire.

Another creative component of our project is a visual editor that supports CRUD operations on the tables that users save to their accounts. This is technically challenging because similar to the backend algorithm, it requires us to relate frontend user interaction to backend instructions to modify the existing table that the user altered.

The last creative component of our project is the backend query translation engine. This is because it is a very complex and technically challenging feature that we will design to be as generic as possible, meaning that it should be able to dynamically recognize JOINs, validate queries, etc. just based on the definition of any arbitrary schema. This is important so that our backend code remains clean and maintainable while allowing our application to remain generalizable across different database schemas, even though we will only focus on music for this project.

Usefulness

Our application is useful because it abstracts away all of the SQL queries and allows for a wider range of people to interact with databases. For many, SQL can be daunting to learn, and people who might be interested in learning about music are very broad. Creating an interactive user interface and abstracting away the SQL makes databases and data analysis more approachable.

On our frontend, we would have a selection of dropdown menus that display various options for working with tables. For example, we will have a dropdown menu that allows users to pick which tables they would like to join, a dropdown menu with the columns they want to select, and a dropdown menu with a couple of hand-curated WHERE conditions. Once they have selected all of the options that they would like to create their query with, they will press a button that will then display the table that the user curated. We may also add a feature that shows the user the actual SQL query that returned the table, which allows the user to see the query and potentially learn a little bit of SQL in the process.

As far as we are aware, there are no websites that are doing quite what we are doing. There are websites like Stats for Spotify, which tell the user their top tracks, what songs they recently played, etc. However, there isn't a website that allows users to create their own SQL query about music (or any topic) without having to write any of the SQL themselves.

Realness

We have currently selected two datasets, but could potentially add more as we move further along in the project.

The first dataset we are using is the Spotify Tracks Dataset from Kaggle. This dataset is in CSV format and stores about 114,000 rows, with each row representing a single Spotify track. There are 21 columns, with several dedicated to the description and identification of a track (such as track_id, track_name, album_name, duration_ms, etc.). There are also other columns that quantify certain features of a track, such as "danceability," "speechiness," "loudness," etc.

The second dataset we are using is the Spotify Artist Feature Collaboration Network from Kaggle. This dataset contains two CSV files, one called nodes.csv and one called edges.csv. nodes.csv stores data on a large selection of Spotify artists, and edges.csv stores data on who each artist collaborated with. We will likely not use the edges.csv file, so we will focus more on the nodes.csv file. The nodes.csv file contains 6 columns, 2 of which are a means of identifying the artist (spotify_id, name), and the other 4 columns are features of that artist, such as the number of followers or their most popular genre of song. There are over 156,000 rows, where each row represents a single artist.

Functionality

The first main functionality that we will have is a panel on the left side of the screen that displays the database schema to the user. In our database, we will store several tables related to music. For example, we will have a table or two storing data on Spotify tracks from the Spotify Tracks Dataset. We will also have another table storing the data in the nodes.csv file from the Spotify Artist Feature Collaboration Network dataset. Having this as a reference allows the user to see the database they're interacting with and make better decisions on how they can JOIN tables, SELECT the types of columns, and more.

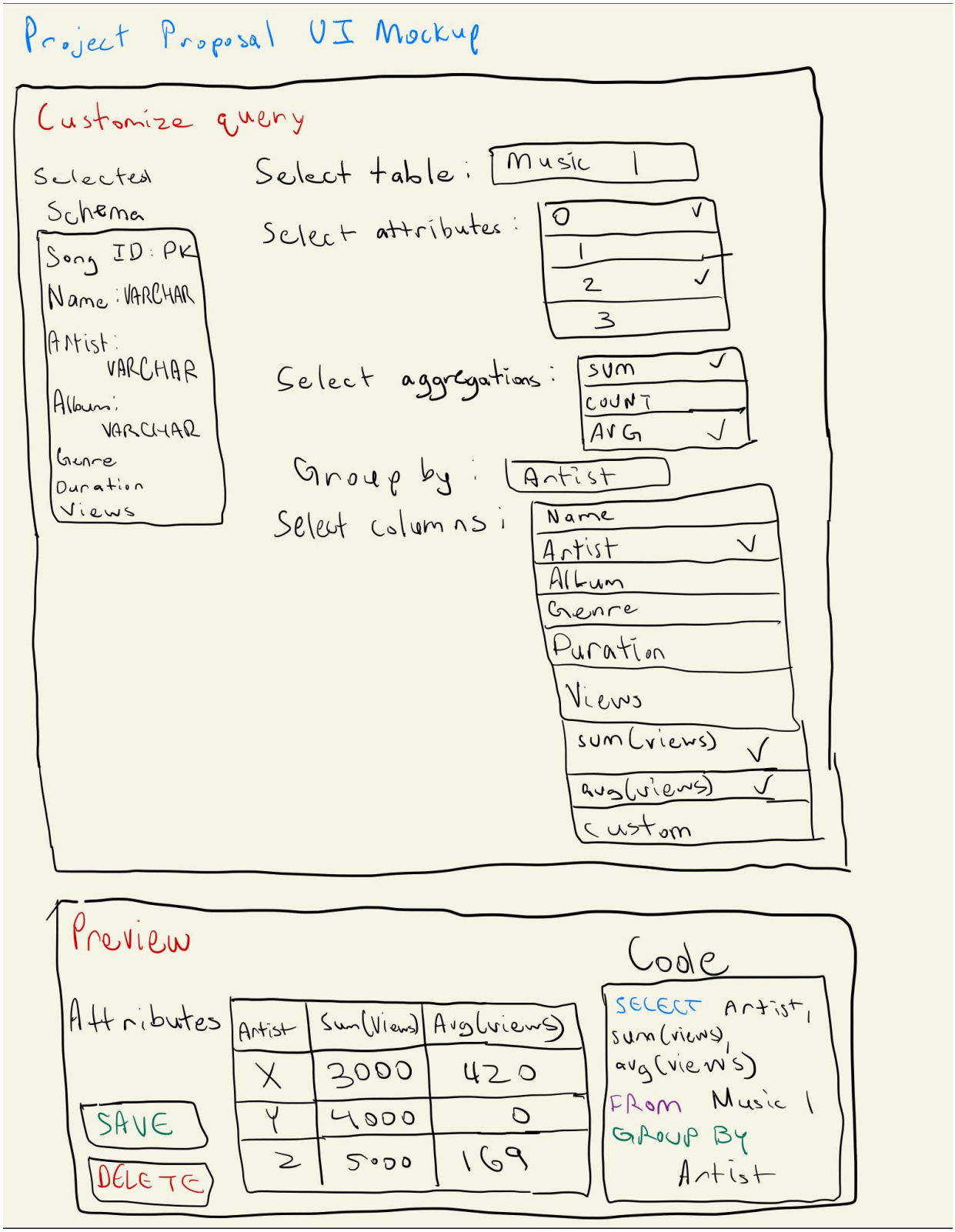
Our second main functionality is our selection of dropdown menus that allow the user to choose which tables, columns, and attributes the user wants in their SQL query. The first dropdown menu will store which tables that the user wants. The dropdown menu will show every table in our database, and we

will allow the user to choose up to two tables, with a JOIN being used to connect those two tables (if two are chosen). The next dropdown will allow the user to choose which columns they want to have in their returned table. Depending on which tables were chosen in the first dropdown, the second dropdown will contain all of the columns in the tables chosen, allowing the user to choose as many columns as they would like. This is equivalent to the SELECT instruction in SQL.
The next dropdown will have a series of curated WHERE statements. What the user gets to choose from changes depending on what columns they've selected.
We may also have dropdowns with aggregates and GROUP BY options as well.

Our third main functionality will be a panel that shows the table that was returned based on the user's selections in our dropdown menus. This is pretty self-explanatory: the app will create a visual representation of what the SQL query (that was created by the user's options in the dropdown menus) returned and display it to the user. We may also add the SQL code that generated the query in this panel.

Our fourth main functionality is to allow the user to create a table based on their returned table. After the table is returned, the user can create a table using their query, separate from the rest of the tables in our database. Should they wish, the user can update their table by adding new rows to their table. They will also be able to use our dropdown menus to select columns from their chosen table (the user will not be able to use our JOIN dropdowns or be allowed to use our curated WHERE statements; only SELECT will be available). When the user generates another table that they want to save, the user will be allowed to store their returned table in the database, but the previous table that was stored in the database will be deleted.

Low-Fidelity UI Mockup



Project Work Distribution

The work for our project can be divided into the following main components:

1. Database
    - Putting together the core datasets
    - Designing the schema by dividing information into separate, related tables
    - Managing the database
    - (Potentially) setting up data pipelines to enable up-to-date data
2. Backend
    - Writing the core query generation algorithm
    - Creating a system to perform CRUD operations on user-generated tables
3. Frontend
    - Designing the frontend UI for generating tables
    - Designing the UI that enables CRUD operations for user-generated tables
4. Integration
    - Determining and implementing the infrastructure of the application
    - Defining endpoints for components of the application to communicate
    - Set up a user authentication system

Our group expects that most of our project development time will be spent on the backend since it generates the SQL queries based on what the user selected on the frontend. Therefore, for most of the project, we will divide up the work like this:

James and Juno will be responsible for designing the various SQL queries based on the combinations the user can choose from the dropdown menus on the frontend. James and Juno will also help write and design code so that these queries are done.

Ryan and Michael will be responsible for writing and testing the backend. Using the design of the SQL queries and code that James and Juno give them, they will go write and test the code to make sure that each of the possible queries works as well as provide feedback if James and Juno either 1) forgot a possible combination of options or 2) the design needs to be changed.

Once we feel that we have made enough progress on the backend, one or two people will move over to frontend development (likely James and Juno). These individuals will be responsible for designing and creating the frontend and creating endpoints to connect with the backend.