Stage 6 - Project Report

- *Please list out changes in directions of your project if the final project is different from your original proposal (based on your stage 1 proposal submission).*
    - No changes were made comparing our original proposal goals to the final project. We executed a trivia application with a database of academic quiz bowl questions.

- *Discuss what you think your application achieved or failed to achieve regarding its usefulness.*
    - The application successfully achieved our goals from the original proposal. We were able to create a trivia application that allowed users to create quizzes, practice/test their knowledge, join teams, as well as display statistics/data pertaining to their knowledge of certain subjects in trivia.

- *Discuss if you changed the schema or source of the data for your application*
    - We did not need to change the schema or source of the information - we had enough data to make a solid trivia app with a variety of questions. The question structure lent itself well to the schema we had designed for it. The relationship tables (teamMembership, quizQuestion, response) best captured the idea of users joining teams, questions belonging to a quiz, and users submitting a response to questions. This also allowed us to reuse questions as prairielearn does.

- *Discuss what you change to your ER diagram and/or your table implementations. What are some differences between the original design and the final design? Why? What do you think is a more suitable design?*
    - The only change we made was in the response table, we had responseID as a primary key, but we felt this was unnecessary as we would not want a user to submit multiple responses to a question within the same quiz, which they would be able to do if we added responseID as a primary key. Instead we made quizID, userID and questionID the primary key for the table. This allowed the same to answer the question multiple times in different quizzes and different users to answer the same question in the same quiz, but would not let a user retry a question in a quiz. The latter is more suitable because it used the inherent SQL schema to enforce constraints on what the user could submit as an answer.

- *Discuss what functionalities you added or removed. Why?*
    - Some functionalities we removed were creating a login page for users, showing a trend in how well a user did in a category, as well as getting the final score/report for each quiz taken. This is because the frontend features for these would be difficult to implement, and the core functionality of the project didn't rely on them. Also showing data over time would have required session affinity which is a difficult backend feature to implement efficiently and would have increased the scope of the project to something infeasible for a semester's work.
    - Some functionalities we added were creating a stored procedure to show how each user compares to others in their team at their best category respectively. This could help in creating a team for competitive trivia, or just for fun to see statistics. We also added a trigger to delete a team when the team had no more members. This helped keep the team table compact thereby improving performance and efficiency. A backend feature we implemented was containerizing the application and deploying it to GCP Cloud Run and Cloud Build - this is a useful feature for a web application because GCP provides load balancers and autoscalers, so if many people end up using the application at once, the app will scale up the number of instances to complement the load.

- *Explain how you think your advanced database programs complement your application.*
    - We created three advanced queries, a stored procedure, and a trigger.
    - **Advanced Query 1:**
        - This gets the team(s) the user is on and the other members of that team. This complements our application because users are able to get information about which teams they're on, as well as search them up in our database to get information about which categories they excel in.
    - **Advanced Query 2:**
        - This query gets the average percent correct of total questions for members of a team. With this query, users are able to see how well their team members are performing on trivia questions and check in the future if they're improving over time. It can also let users see how they compare against teammates.
    - **Advanced Query 3:**
        - We can get the average percent correct per category for a user through the query. This allows users to see how they're currently performing in each category, and check their team member's

statistics too. This leads into how TriviAttack aids in providing an outlet for users to answer trivia questions, check their current knowledge, and improve their ability.

- **Stored Procedure:**
  - Our stored procedure returns all users in a team that have a better average score in their best category compared to the user who requested said data. This complements our application as users are able to check which members are able to perform best in their respective categories (in case the users wants to create a team for a competitive setting), and which members are achieving better results than the user requesting it.
- **Trigger:**
  - The trigger we implemented deletes a team when the last team member leaves the group. This helps the database itself as we can get rid of unnecessary information that can both clutter up resources and increase the runtime/inefficiency of the database when searching for values.

- *Each team member should describe one technical challenge that the team encountered. This should be sufficiently detailed such that another future team could use this as helpful advice if they were to start a similar project or where to maintain your project.*
  - One technical challenge that the team encountered was trying to call the database backend through a TCP socket for the current running instance of GCP requested by a client. Unfortunately that didn't work in docker, but using a UNIX socket instead was able to solve this issue. We believe the reasoning behind this is because TCP requires a validation of the IP-address being used from the requesting machine to the server, but docker always has a randomized IP-address. As we don't know the address prior to running the instance, GCP isn't able to verify the connection for security reasons. UNIX sockets on the other hand can make a proper connection directly to the client as they were on the same VPC in google cloud. UNIX doesn't need to verify the IP-address; rather only checks if the client and server are on the same network and then allows access/connection to the database backend.
  - Also containerizing the application and deploying both the front end and backend via one CI/CD pipeline was difficult - The GCP build system had different behavior than our local development system, so we had to make sure we started with the correct container and used the right installation

process to copy files from the source repository to the container and run the build.
    - Another technical challenge that we encountered was the auto-rerendering of components in Node.js and react. What would occur was that when we tried to render and use a certain function to get the question text for a question ID, it would constantly rerender our application's front end. This would cause massive issues as the question text and other components would be randomly rendered in an unpredictable manner. We tried many things to try to solve this problem: saving the question text into an array, saving the question text as a component of the "state", calling question text after a button click was triggered. In the end, to solve this problem, we chose to not automatically get the question text and made it render based on the user input of the question ID.
    - One technical challenge that we encountered was getting the technical components of our application to successfully show up in the google cloud shell terminal. For example, when trying to create the trigger and stored procedure, we would consistently encounter an issue where the cloud shell terminal would see a delimiter and halt, not allowing us to type the entire SQL code. To solve this issue, we used a python script as documented here to directly communicate with our cloud SQL application and inject the code there . We used triple quotes in conjunction with this python script to send the entire query to our cloud SQL application. The triple quotes in python ignore delimiters so this solution worked for us.

- *Are there other things that changed comparing the final application with the original proposal?*
    - Though not much changed in the backend of the project, regarding the database schema or what functionality the app would provide, we did change a few things in the front end. We had initially intended on displaying the questions one word at a time and giving the user a timer to answer questions. In addition to being a rather difficult front end feature to implement, we felt that this goes against the goal of the application which is to help people casually get better at trivia. We felt giving the users more control over which questions they attempted and being able to move back and forth between questions like in a prairielearn assessment was better. We also did not feel that an overall leaderboard for the application was useful because users would care more about their team's scores, so we provided more interfaces for a user to get their team's performance (team leaderboard, getting the best categories for an arbitrary user). We also felt a

login was not a necessary feature so that a user could look up their teammates and their associated stats without needing to develop a full social networking application.

- *Describe future work that you think, other than the interface, that the application can improve on.*
    - One potential improvement would be for our app to use a graph database instead of a relational database like we currently have for handling the teamMembership table and connecting users to each other. A graph database would provide a huge speedup because the complex networking of graph databases would handle interactions between users and groups more efficiently.

- *Describe the final division of labor and how well you managed teamwork.*
    - We all were interested in all parts of the application, so the best way we knew to get everyone involved was by doing most of the implementation and design work in our group meetings, and executing minor fixes and changes alone. Everyone worked on all aspects of the project equally and in a timely manner; no issues/problems had occurred with regards to teamwork and division of labor. Whenever someone was stuck on a certain aspect of a project, we made sure to work together and solve the problem.

Final Video Link:
https://drive.google.com/file/d/1bxR5373Px19HYAg4dQybigqIZBaeOclX/view?usp=share_link