Carson Youman
Eddy Caballero
CS 327e

Final Report: Primary G's

We knew that in order to succeed with these labs, we would need to pick a solid team name. A strong team needs a strong team name at its core, and we would not settle for mediocrity. After tossing around several pun and alliteration heavy ideas, we decided on Primary G's. Crossing the database term "primary key" with the abbreviation for gangster, "G", we had struck gold. When thinking about what sorts of datasets we were interested in, Rugby was a natural choice. As members of the University of Texas' Men's Rugby Team and avid rugby fans, we were thrilled at the opportunity to take a data driven look at the sport we love. Finding good data proved to be quite the challenge. We scoured the Internet, but were struggling to find data that would allow us to gain useful or interesting insights. Most of the data sets we encountered only contained a few fields or would be extremely time consuming to save and organize. Our dream of working with rugby data seemed to be coming to an end before even beginning and we shifted our focus to finding a data set that would give us a lot to work with. We found a few good governmental data sets, but were not nearly as excited about healthcare or causes of death.

As we were about to press on unenthusiastically, an idea came to us. We decided to reach out to an old rugby coach. He had always broken down the statistics of our matches so that we see empirical evidence of what we did well and what we needed to improve on. We were in luck; he actually had several rugby related datasets (in csv form

nonetheless) on his Github account! The most robust of the bunch contained possession data from the 2014 IRB 7s circuit, a series of tournaments held across the globe every year in places such as Dubai, London, and Hong Kong. This was an especially exciting dataset for us to work on, as we had attended the 2014 tournament in Las Vegas!

Each record the table corresponds to a single possession, holding data like the team, opponent, source of possession, and the result. While we knew we could identify some interesting trends with this data alone, we knew it would become much more powerful if we combined it with the results of each game. We had encountered this data in our previous search, and where before it was essentially useless it was now quite the opposite. Organizing this data was a challenge in and of itself, as we had to manually copy each the data from each tournament separately into an excel file. Furthermore, we had to alter it so that we could more easily traverse the data and later tie the records together with our queries. For example, the score of each game was kept in one field. We had to take a field that looked like "21-19", and turn it into two separate fields "21" and "19". We were able to do this easily using standard excel functionalities. We also wrote a custom macro that took each team and the scores as a parameter and returned the winner. We decided that it would be easier to store the winner as a separate field rather than include the logic in our SQL queries.

Once we had our data cleaned and organized, we began to model our database. At first, junction tables were a bit confusing for one of our team members. Carson had had experience working with databases before, his most recent and most extensive work

being in MIS 333k. In that course, he used Entity Framework within an MVC ASP.NET project. In Entity Framework, relationships were stored within each model. If the relationship could have many records associated with it, the property that acted as the foreign key would hold a list of objects it was related to. The junction tables were created on the back end and therefore did not have to be explicitly created. It took him a while to adjust to this as we were creating our logical model.

Next we set out to create our tables in MySQL. This was relatively new territory for both of us, as we had no prior experience with MySQL. Interestingly enough, the most difficult part of this step was setting up the environment itself. We never quite figured out how to access each other's local connections and therefore decided to use Eddy's computer. After setting up the environment, we were able to create our tables with relative ease using tutorialspoint.com as a quick and easy reference for syntax. Crucial in enabling our success in this step was taking a few minutes to critically think about the order in which we would create the tables. This allowed us to avoid foreign key dependency issues down the line.

In the same way that our team name created a solid foundation for our project as a whole, our success lab 1 enabled us to jump right into lab 2 without having to make any major changes. In order to streamline our use of pymysql we created a series of functions for commonly used commands in a file called db_connect.py. Some examples of these functions were to create and destroy connections or run statements. The main benefit of

these functions was exception handling, running try-except statements to print out the corresponding error.

When inserting our data into the database, we had to ensure we entered it in the correct order. At first we had forgotten that this was the case, but once we figured it out, it was easy to follow the order we established when we created our tables. We were quite comfortable reading data from a csv file. One of our team members, Carson, had done a lot of work with csv readers during his internship at Visa. Most of our import scripts were relatively straightforward. After using a simple counter to ignore the first line of data (the field titles), we were able to add our necessary data by accessing the required field by its index.

Another challenge we faced was in regards to adding foreign key values. We needed to find an easy way to retrieve the primary key values of teams based on their name. We knew that we could do this with a query, but wanted to keep things as simple and efficient as possible. Our solution was to create a dictionary with the keys being the team names, and the values being their primary key values. This makes sense in the scope of this project because we had a relatively short list of team names and knew their values.

Had this list been longer, or the primary key value was subject to change based on the original entry of the data, we would have needed to run a SELECT statement to find the appropriate value (SELECT Team_ID FROM Team WHERE Team_Name =?). We utilized this dictionary, as well as a similar one for Tournaments, in our

import_TeamTournament junction table. Here are the dictionaries themselves: t_dictionary = {'HK7s':1, 'L7s':2, 'D7s':3, 'GC7s':4, 'T7s':5, 'NZ7s':6, 'S7s':7, 'PE7s':8, 'USA7s':9 } and team_dictionary = {'ARG':1, 'ASM':2, 'AUS':3, 'BEL':4, 'BRA':5, 'CAN':6, 'ENG':7, 'FIJ':8, 'FRA': 9, 'HKG':10, 'JPN':11, 'KEN':12, 'NZL':13, 'PNG':14, 'POR':15, 'RSA':16, 'RUS':17, 'SAM':18, 'SCO': 19, 'USA':20, 'WAL': 21, 'ZIM':22} . We ran SELECT statements for some of the more difficult or impossible to predict primary key values. In import Possession we ran the following SELECT statements to retrieve the Game_ID (cursor.execute("SELECT Game_ID FROM Game WHERE T_Game_Number = '" + line[2].strip() + "' and T_ID = '" + str(t_dictionary[str(line[1].strip())]) + "'")). It did take us some time to figure out how to retrieve the value from the cursor, but we used cursor.fetchone()[0] because the query should only return one value.

The objective of lab 3 was to create SQL queries and views and to then develop a command-line interface in python with menu options for each query. We started lab 3 by thinking of what sort of insights we could gain from possession and game data. We knew we wanted to see data about the amount of possession and the types of wins. We decided to start with a simple query to show how many of each type of possessions there were throughout all the tournaments. This turned out to be quite simple, all we had to do was create a simple select statement from the possession table and count the number of possession ids. The next few queries turned out to be a bit more complicated. In order to accomplish them we first had to create a view using multiple select and join statements. Creating this view took us awhile due to how complex it was. You can see in the picture

bellow just how many selects and joins we used.

```
CREATE VIEW Close_Games_Raw as
select Team_1 as Team1,(select distinct t.Team_Name from team t inner join game g on t.Team_ID = g.Team_1 where g.Team_1 = Team1) as
Team1Name, Score_1, Score_2, Team_2 as Team2, (select distinct t.Team_Name from team t inner join game g on t.Team_ID = g.Team_2 where
     g.Team_2 = Team2) as Team2Name
from game g inner join tournament t on g.T_ID = t.T_ID where (abs(Score_1 - Score_2) < 7)
order by t.T_Date;
```

Luckily the views used to find blowout games and user chosen point differential games were very similar to this. The only part we needed to change was the where statement in the fifth line. Then we created select statements that queried from those views to show which games were close and which were blowouts. We also created a query that allowed the user to enter a number and show which games had that point differential. Our next view showed what tournaments each team played in. Like before, we created multiple queries from this view. One of these queries asked for the user to pick a team to show how many tournaments this team played in. In order to avoid possible errors from a string input we decided to create a team dictionary so a user only had to enter an integer from a given list to choose what team they wanted to see data from. We used this method in each of the queries that asked for user input. Compared to lab 2 where we populated our tables, we found that sending commands to MySQL in this lab was much easier.

When creating the command-line interface we did not run into many problems. Most of the problems we ran into were easy to fix syntax errors. However, there was one problem that we ran into with the views that we created. When we ran a query in the interface that used a view that had already been used in a previous query an error was returned. The error was occurring because the query was trying to create the same view again. So, to fix this we just added “drop view if exists” to any query that used a view. This fixed our problem by simply recreating the view each time a query required it. We also checked the user input, when the user chose to see teams that won a user entered

range of tournaments. To make sure the query returned valid results we added two loops to check the number range that was entered. The first made sure the lower end of the range is between 1-9 and the second loop made sure that the greater end of the range is between the lower and 9. This made sure that the query would return a valid result.

Overall, lab 3 was fairly easy since we did not run into many problems. After completing it, we realized that there are a few things that we should have done differently. We assumed that the champion of each tournament is the team who won game 45. This number works perfectly with our data, but it is possible that in the future there could be a tournament with more teams where the championship game is not game 45. We should have developed a different way to find which team is the champion of each tournament. If we were to do this project again we also should have created queries that gave us more meaningful insights, like the average number of possessions during a game and the most common place on the field that tries were scored from. We could have found both of these from our data with some work.

The objective of the final project was very similar to the objectives of lab 3. First, we created an API client in Python and searched for a topic in Twitter. We then stored those JSON results in a new table in our MySQL server. The first problem we ran into during this project was installing tweepy. It turned out that we had an older version of python running, so we first had to install the latest version. After we cleared that up we were still running into an error when trying to install. After some Google searches we figured out that we had to run some extra commands in order to get tweepy to work.

These commands did the trick: “sudo pip install requests[security]”, “pip install pyOpenSSL ndg-httpsclient pyasn1”. We then started to alter the example API client so that it worked with our own database. To do this we simply changed the sql_query in the picture below

```
def do_data_pull(api_inst):

  sql_query = "select Team_Name, Team_ID from Team order by Team_Name"
```

and also the insert_stmt in the picture below.

```
for page in twitter_cursor.pages():
  for item in page:
    json_str = json.dumps(item._json)
    print json_str
    print "found a " + team_name + " tweet"
    insert_stmt = "insert into Tweet(tweet_doc, team_ID) values(%s, %s)"
```

Now that we had tweepy installed and had altered the API client we went to Twitter to find our API key and token. However, when we entered our key and token into our API client we got an error. Apparently our key was invalid, so we reset our key and tried again. Resetting the key fixed this problem. At first, we tried searching fro tweets with the hash tag #Rugby7s and a team name. Unfortunetly, this did not return a ufficient number of results. Instead, we searched for tweets with the hash tag #Rugby and a team name. This gave us more than enough results. Now, we had a working API client and a new table in our database filled with tweets containing the hash tag #Rugby and a team name from our table Team.

The next step was to create some new queries using the new Tweet table. These queries were much easier to create than the ones we created in lab 3 since we were not required to specific types of join or aggregations. Our first query was a simple query that returned the number of tweets each team in out database had if the team had at least one

tweet. We used a join in order to retrieve the name of the team from the Team table. The next query was similar but returned all the teams and the tweet counts including those teas with zero tweets. The next query was similar to the second but instead of returning the tweet count it returned the number of users that tweeted for a team. After creating the first three queries we realized that we needed to pull more data from the JSON in tweet_doc. So, we decided to pull the number of favorites for each tweet. We did this by adding the following lines to our create table statement.

```
favorite_count varchar(32) generated always
    as (json_unquote(json_extract(tweet_doc, '$.favorite_count'))) stored,
```

This gave us enough data to create two more queries for our command-line interface. The fourth query returned the number of favorites that tweets about a certain team received. This was very similar to our second query where we returned the tweet count for each team. However, here we used a where statement in order to return only the teams that had at least one favorite on a tweet about them. For our last query we returned how many tweets a user had received for his or her tweet.

The next step was to extend our command-line interface to include our new queries. This step didn't take very long; we simply followed the same format as in lab 3. Since we did not include any queries that required views or user input we did not need to add any error checking to the functions. Now the only thing left to do was create a backup of our database. This step took us some troubleshooting. We thought that we only had to enter the command into the command line, but what we really had to do was change the directory to the location of mysqldump and then enter the command. After we did this it turned out that our backup was too large to commit to our Github repository.

Luckily someone was also having this problem so we were able to find a solution on Piazza. All we had to do was specify which tables we wanted to backup. We did this using the following command: mysqldump -u username -p dbname table1,table2,table3 > backup1.sql. By using this method, we created multiple backups each with a few tables in them. However, Our Tweet table backup was still too large to commit, even after compressing the file. So, we installed Git Large File Storage in order to deal with this issue. This allowed us to commit the backup to our repository.

After finishing the final project we think that there are a few things we could have done better. We could have done multiple topic searches so that we could have had more meaningful tweets for each team. Instead we just broadened our search to return more results. Another way we could have created more meaningful queries was to extract more data from tweet_doc. Overall, we learned a great deal throughout the duration of this course, and primarily through the practice we got during the lab assignments. We had both had slight experience in both python and SQL, but had never used them together. Being able to manipulate data through python opens up a lot of possibilities, and is a valuable skill to add to our repertoire.