Derek Watson
CSPB 3287: Fall 2023

Soccer Club Explorer

An API that allows user to explore 15 years of soccer match data for 383 European football clubs, covering nearly 68,000 matches.

Project Overview

The original idea behind my project was to gather a large set of data on soccer matches from somewhere on the web, compile it into a database, and make the data from that database accessible from a web application. Users would be able to search through the data by many different attributes and view different statistics about their favorite teams and leagues from the output of different pre-built complex SQL queries. Users would be able to choose the target of a query or add some filters via a user interface.

Unfortunately, as I'm sure happened to others, I had to scale down the fairly ambitious application portion of the project and focus on the database side more. The web application became an API, and the full set of features became a robust proof of concept. The API is currently set up with the ability for users to get *some* of the tables and search for individual records by *some* attributes. There are also five "reports" which are the complex queries involving user inputs mentioned above. While this is scaled back significantly from the original plan, the foundation is laid out for a larger application to be built.

The remainder of this paper will provide a detailed outline of the data and database in the first section, and the API and queries in the second section. The following links give further details about the project and contain information that may be referenced later in the paper:

- GitHub Repo: https://github.com/derek-watson14/3287-Semester-Project
- Live API: https://h390yqm7gg.execute-api.us-west-2.amazonaws.com/prod
    - Public API Key: 1MIFi6m5t679tGKWaBOUs92tijcojQPi3WMhdYFw
    - Use request header X-API-KEY to pass the API key and use the service.
- Video: https://cuboulder.zoom.us/rec/share/PdHFSB6EaUJiLFFb-RGOW4DFgaidXztAh9s-R_xBnx_ZmXIcdXryx22N0eNYrSA3.Jrcp4lzPF55Y6pkm?startTime=1702620906000
    - Passcode: @a#tup1C

Data Source and Database Design

My data was sourced from the website https://www.football-data.co.uk/ which has soccer match data from many European countries over many years. I chose to include only the largest, most well-known leagues in my dataset, selecting data from the top two leagues in England, France, Germany, Italy and Spain and the top leagues in the Netherlands and Portugal. Originally, I wanted to do 20 years of data, but after downloading the ~250 CSVs comprising that data, I found that the data quality dropped

off after 15 years, so I am including data for all seasons 2008-09 to 2022-23 for the 12 aforementioned leagues.

To collect and process the data I wrote a series of python scripts. One to download the 200+ CSVs from the website, one to process each and combine them, one to analyze the combined CSV to check data quality and one to create massive JSON files organized by database entity. Those scripts can be found in my GitHub repo in the folder /data-gathering/scripts. Initially I was going to use an extremely normalized data setup, where the match statistics for each of the two teams would be stored in a separate table than the common match statistics. Figure 1 shows that schema.

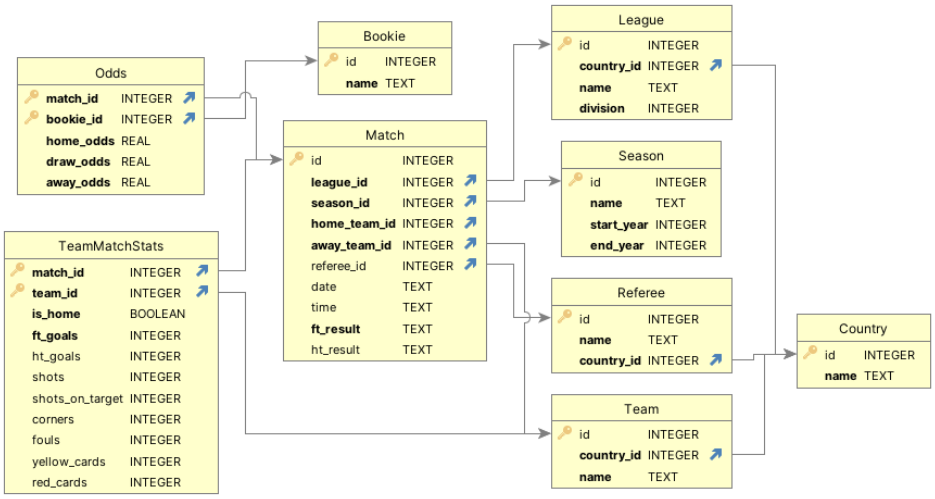


Figure 1: Database ERD Version 1.0

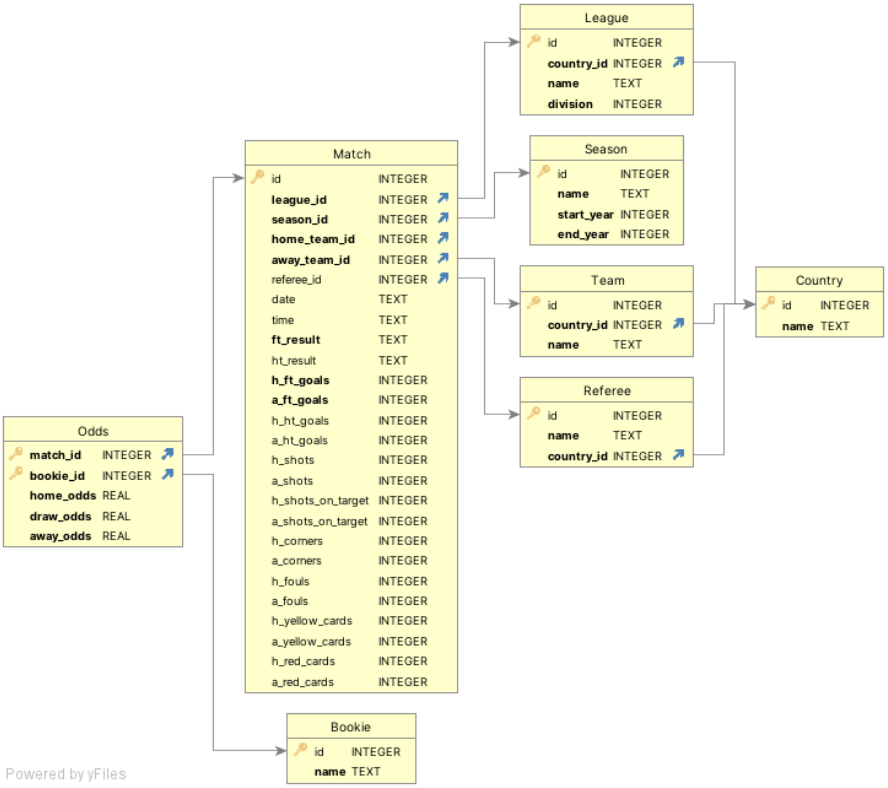


Figure 2: Final database ERD

I eventually found, however, that this approach made queries extremely unwieldy because getting any information about a match would require 3+ joins which would become costly, so I eventually decided to just put all match data in a single table, with the home and away team statistics differentiated by a `h_` or `a_` prefix. This resulted in a minimum of two fewer joins per query, and while some fields in the Match table are repeated twice for “Home” and “Away”, the information is still fully normalized. My final diagram can be seen above in Figure 2.

As you can see in Figure 2, my final database contains **eight tables**, with Match being the largest and most important and data-rich table. All of the attributes with an upward pointing blue arrow represent **foreign keys** and **relationships** between tables are denoted by the lines between different entities. Leagues have many matches, matches have many odds, odds have exactly one bookie, etc.

Because I originally wanted to avoid needing to pay for/find a free server, and because the end product (an API) would be read-only without any real users, I chose to use a SQLite database to hold my data. The .db file would be hosted in an AWS S3 storage bucket and accessed via an API Gateway API backed by a lambda function. More on that in the next section. Below is my **table creation** statement for

the Match table, showing the **constraints** I used as well as the create **index** and create **trigger** statements for the same table.

```
create_table_match = """
CREATE TABLE Match (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    league_id INTEGER NOT NULL,
    season_id INTEGER NOT NULL,
    home_team_id INTEGER NOT NULL,
    away_team_id INTEGER NOT NULL,
    referee_id INTEGER,
    date TEXT,
    time TEXT,
    ft_result TEXT NOT NULL CHECK( ft_result IN ('H', 'D', 'A') ),
    ht_result TEXT CHECK( ht_result IN ('H', 'D', 'A') ),
    h_ft_goals INTEGER NOT NULL,
    a_ft_goals INTEGER NOT NULL,
    h_ht_goals INTEGER,
    a_ht_goals INTEGER,
    h_shots INTEGER,
    a_shots INTEGER,
    h_shots_on_target INTEGER,
    a_shots_on_target INTEGER,
    h_corners INTEGER,
    a_corners INTEGER,
    h_fouls INTEGER,
    a_fouls INTEGER,
    h_yellow_cards INTEGER,
    a_yellow_cards INTEGER,
    h_red_cards INTEGER,
    a_red_cards INTEGER,
    FOREIGN KEY (league_id) REFERENCES League(id),
    FOREIGN KEY (season_id) REFERENCES Season(id),
    FOREIGN KEY (home_team_id) REFERENCES Team(id),
    FOREIGN KEY (away_team_id) REFERENCES Team(id),
    FOREIGN KEY (referee_id) REFERENCES Referee(id)
);
"""
```

Figure 3: CREATE TABLE statement for table "Match"

```
result_triggers_data = [
    ("ft", "UPDATE"),
    ("ht", "UPDATE"),
    ("ft", "INSERT"),
    ("ht", "INSERT"),
]

result_triggers = []

# Write triggers for all combinations of result and insert/update
for time, action in result_triggers_data:
    create_trigger = f"""
    CREATE TRIGGER ensure_{time}_result_in_range_{action.lower()}
    BEFORE {action} ON Match
    FOR EACH ROW
    BEGIN
        SELECT
        CASE
            WHEN NEW.h_{time}_goals > NEW.a_{time}_goals AND NEW.{time}_result != 'H' THEN
                RAISE(FAIL, 'Result must be H if home team scored more goals')
            WHEN NEW.h_{time}_goals < NEW.a_{time}_goals AND NEW.{time}_result != 'A' THEN
                RAISE(FAIL, 'Result must be A if away team scored more goals')
            WHEN NEW.h_{time}_goals = NEW.a_{time}_goals AND NEW.{time}_result != 'D' THEN
                RAISE(FAIL, 'Result must be D if both teams scored the same number of goals')
        END;
    END;
    """
    result_triggers.append(create_trigger)

# Will often be querying/joining by league_id, season_id, home_team_id, away_team_id
match_indicies = [
    "CREATE INDEX idx_match_league_id ON Match (league_id);",
    "CREATE INDEX idx_match_season_id ON Match (season_id);",
    "CREATE INDEX idx_match_home_team_id ON Match (home_team_id);",
    "CREATE INDEX idx_match_away_team_id ON Match (away_team_id);",
    "CREATE INDEX idx_match_ft_result ON Match (ft_result);",
]
```

Figure 4: Triggers and Indexes for Match table

Looking at figures 3 and 4 you can see multiple check **constraints** and BEFORE UPDATE/INSERT **triggers** to help enforce business logic. First, the result of the match should be either 'H', 'A' or 'D' and no other string. That is enforced with a SQL CHECK statement. The triggers are checking that the `ft_result` column actually reflects the score of the match, rejecting INSERTS and UPDATES where a row is inserted that states the home team "won" but simultaneously has fewer goals. Finally there are several indices created on commonly sorted and joined columns within the Match table.

In total in the database, there are seven column indexes created and four triggers. To see all of them, visit the `/db-creation/create_db_tables.py` script in my GitHub repository. Because I was using SQLite, I was able to simply use the python standard library sqlite3 package to perform all of my table creations. After creating the tables, I iterated through each item in the massive JSON files created earlier in my data acquisition process to fill the database with my converted CSV data. This was performed by the `/db-creation/batch_insert.py` script. Next I will look at the web API.

Web API

As stated earlier in the paper, I had to scale down my original idea from a web application to an API and from a fully fledged data-exploration application to a robust proof of concept. In the end I created a serverless web API using AWS Cloud Development Kit (CDK) with the following endpoints:

Endpoint from base URL	Possible query string parameters	Description
/league	name	Get all leagues or search by name
/team	name	Get all teams or search by name

/season		Get all seasons
/reports/betting-upsets	leagueId, startSeasonId, endSeasonId	Get the biggest upsets in a league over a range of seasons
/reports/scoreline-occurrences	leagueId, startSeasonId, endSeasonId	Get the most common scorelines in a league
/reports/team-history	teamId, startSeasonId, endSeasonId	Get the season-by-season stats for a given team
/reports/historical-table	leagueId, startSeasonId, endSeasonId	Get the standings for a league over a period of time
/reports/league-compare	startSeasonId, endSeasonId	Compare aggregate statistics across leagues

The idea is that the league, team and season routes can be used to get the ids needed to access the reports. All of these are GET methods, so for the end users the data is read only which is the only real possibility using SQLite. If new seasons were to be inserted or old data updated, in theory a database administrator would perform that locally, then redeploy the application to AWS with the updated database attached. In reality we would probably use a more robust solution in a production application.

Now I will go over one of the more complex **queries** and show how **joins, grouping** and **aggregate functions** are used to get complex statistical information from the data. All of the queries can be found in the `/application/router/Controller.py` file in my GitHub repository. Figure 5 shows the query used for the "betting-upsets" report. It probably uses the most SQL features of any of the queries I created for this project. It includes: column aliasing, joins, subqueries, aggregate functions, grouping and a limit statement, among other features. It returns all calculated columns that are not in the original database and can be altered by user input that is passed into the query as parameters, making them safe from SQL injection, since they are coming from the internet. The user can choose the league that is examined as well as a range of seasons by forming a query like:

https://h390yqm7gg.execute-api.us-west-2.amazonaws.com/prod/reports/betting-upsets**?leagueId=1&startSeasonId=12&endSeasonId=15**`

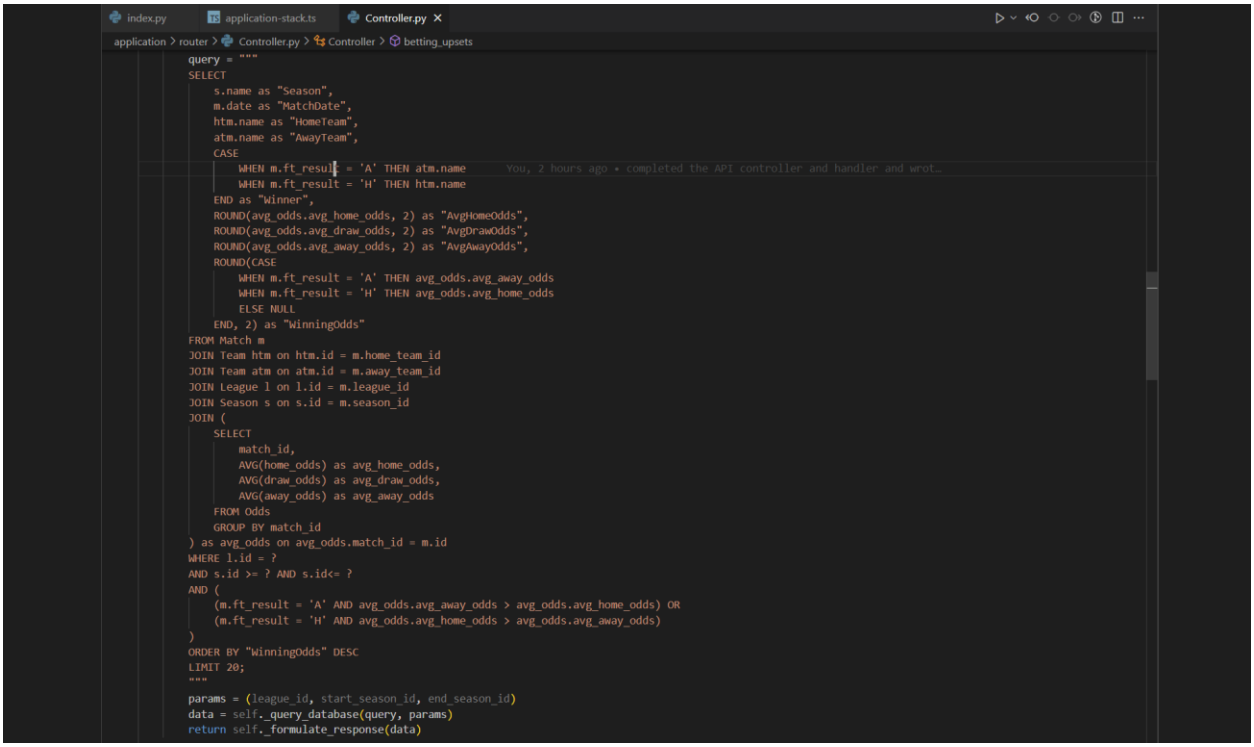
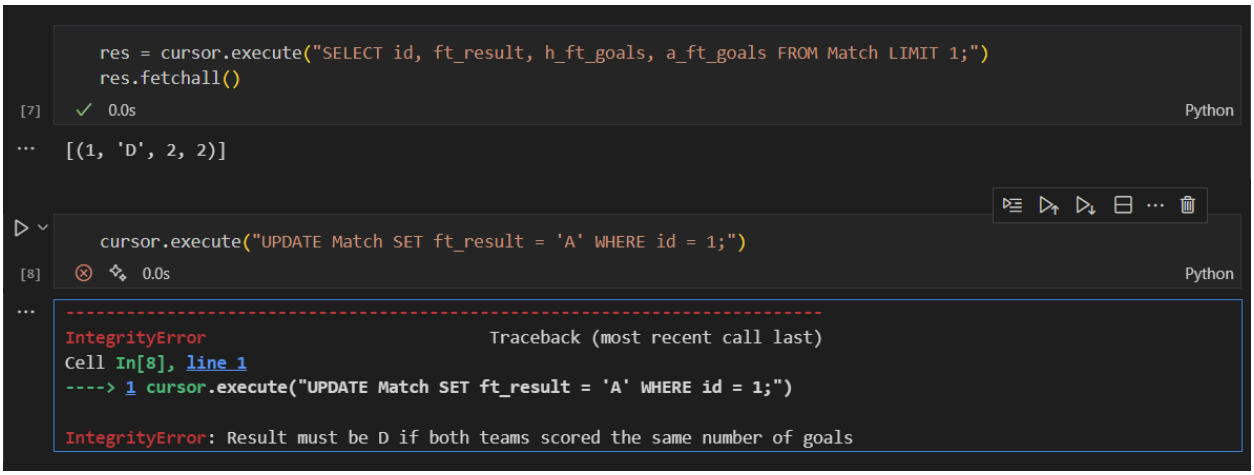


Figure 5: Query for reports/betting-upsets endpoint

The above query can be tested using a tool like Postman if you pass the public API key from above in the headers. If I had had more time to continue developing the project, I would have liked to create more "report" style queries than the five I created in this version. I also would have liked to build a frontend application that used the queries to populate tables and graphics on a website as I originally envisioned, with user input being passed from a web form into the query parameters of the different routes. Because the application/API is read only, I will have to demonstrate how my triggers effect **updates** and **deletes** in the video linked at the beginning of this paper. I will include a screenshot from a Jupyter Notebook showing the output when I attempt to take an action blocked by a trigger on the Match table:



Conclusion

Time constraints prevented me from getting as far into building the application as I had hoped, but I focused on the database portion of the project as instructed and met many of my learning goals: I improved my understanding of database design, my ability to write complex queries and got a better understanding of the benefits of indexing. Beyond that I also learned how important it can be to use integer primary keys over string-based properties within queries, even when string column has an index. On my home PC, everything ran very quickly no matter how I wrote it because it is a fairly powerful machine. My API router function on the other hand is backed by a lambda function that only has access to 256mb of RAM, about 1/64 the amount in my computer. When I used the League.name or Team.name column in my WHERE clauses, some of the super complex queries would time out of the three second execution window, behavior that persisted even after adding an index. When I used League.id or Team.id however, the execution was again almost instant. Overall I am happy with the results of this project and how much I have learned.