CSDS 341 Project Proposal: NFL Database

Jack Zhang, Aman Bhriguvansh, Rishik Hombal, Lauren Sharkey December 6, 2020

Background:

There is a large amount of data needed to evaluate teams and players in any sport. The NFL stands for the National Football League, with 32 professional American football teams. There are many positions on each team's active roster that are filled by 53 players. Each position has its own statistics, and the types of statistics collected on each position. For example, running backs have statistics including receiving yards and rushing yards. These same statistics wouldn't make sense to keep track of for a kicker. Instead, we will keep track of their field goals made and field goals missed.

We have created a database that holds five primary positions within a team. The database created was based on the positions that are listed in a standard fantasy football team. We have combined the defense and special teams into one entity, treating them as one unit. This means that each team can have multiple quarterbacks, wide receivers, running backs and kickers. Each team only has one defense. Each of the five positions will hold information on the name of the player who plays that position and the name of the team that it is associated with (with the exception of the defense which will only be identified by the team).

The database we have created will enable users to search for the statistics of any given player on any given team, for a given position. Each position will store its own unique statistics which are relevant to that position, so a user can use aggregate functions to find the best player in a specific position based on an individual statistic or the best team based on an individual statistic. Similar to fantasy football, users should be able to pull information about different players and teams to compare them. Aggregating the data in a database will allow the users to get all of the information they want in one place

Data Description:

Entities:

Team: (<u>teamId</u> INT,
teamName CHAR(100),
city CHAR(100),
stadium CHAR(100),
superBowls INT,
division CHAR(100),
headCoach CHAR(100),
manager CHAR(100),
owner CHAR(100),
record CHAR(100),
PRIMARY KEY(teamId),
FOREIGN KEY(defenseId REFERENCING Defense)
)

This table holds information about each team in the NFL. Record will refer to the teams win/loss tie over the regular season. A win adds 3 points to the record, a tie adds 1, and a loss subtracts 1. Each team also has a unique name.

Stadium: (<u>stadiumId</u> INT,
stadiumName CHAR(100),
city CHAR(100),
state CHAR(100),

capacity INT,
homeTeam CHAR(100),
PRIMARY KEY(stadiumId),
)

This table holds information about a stadium that an NFL team plays at. Each stadium Has a team that is located in the same city, a “home team”.

Defense: (teamId INT,
defenseId INT,
interceptions INT,
sacks INT,
rshYdsAllowed INT,
passYdsAllowed INT,
PRIMARY KEY(defenseId),
FOREIGN KEY(teamId REFERENCING Team)
)

This table holds information about the defense of each team. Each team has one defense, and each defense can only be a part of one team.

WideReceiver: (wId INT,
teamId INT,
name CHAR(100)
recYds INT,
catches INT,
tds INT,
fumbles INT,
drops INT,
PRIMARY KEY(wId)
FOREIGN KEY(teamId REFERENCING Team)
)

This table holds positional information about each player designated as a wide receiver. Each team can have multiple wide receivers, but each wide receiver can only play for one team.

RunningBack: (rId INT,
teamId INT,
name CHAR(100),
recYds INT,
rshYds INT,
catches INT,
tds INT,
fumbles INT,
drops INT,
PRIMARY KEY(rId)
FOREIGN KEY(teamId REFERENCING Team)
)

This table holds positional information about each player designated as a running back. Each team can have multiple running backs, but each runningback can only be on one team.

Quarterback: (qId INT,

teamId INT,
name CHAR(100),
passingYards INT,
rushingYards INT,
completions INT,
tds INT,
PRIMARY KEY(qId)
FOREIGN KEY(teamId REFERENCING Team)
)

This table holds positional information about each player designated as a quarterback. Each team can have multiple quarterbacks, but each quarterback can only be on one team.

Kicker: (kId INT,
teamId INT,
name CHAR(100),
fgMade INT,
fgMissed INT,
PRIMARY KEY(kId)
FOREIGN KEY(teamId REFERENCING Team)
)

This table holds positional information about each player designated as a kicker. Each team can Have multiple kickers, but each kicker can only be on one team.

Relations:

PlaysFor: (qId INT,
rId INT,
kId INT,
wId INT,
teamId INT,
PRIMARY KEY(qId, rId, kId, wId, teamId),
FOREIGN KEY(qId REFERENCING Quarterback),
FOREIGN KEY(rId REFERENCING RunningBack),
FOREIGN KEY(kId REFERENCING Kicker),
FOREIGN KEY(wId REFERENCING WideReceiver),
FOREIGN KEY(teamId REFERENCING Team)
)

This relation will be one to many from team to position, and many to one for those positions to a team. It connects the different positions, excluding defense, with each team. One team will have many players (and by extension many positions), however a player can only play for one team.

PlaysAt: (stadiumId INT,
teamId INT,
PRIMARY KEY(stadiumId, teamId),
FOREIGN KEY(teamId REFERENCING Team),
FOREIGN KEY(stadiumId REFERENCING Stadium)
)

This relation describes a one to one relationship between teams and stadiums. Each team has a home stadium that they play at the most, and a stadium can only have 1 home team.

Contains: (teamId INT,
defenseId INT,
PRIMARY KEY(teamId, defenseId),
FOREIGN KEY(teamId REFERENCING Team),
FOREIGN KEY(defenseId REFERENCING Defense)
)

This relation describes a one to one relationship between defense and teams. We have to relate defense to teams separately from the rest of the positions because each team will only have one defense. Each team has one defense, and each defense belongs to one team.

Functional Dependencies

Team

We can use the teamId to find the remaining attributes. Since each team has a unique teamName, we can also find all the remaining attributes. Divisions are organized by geographic location so the city can determine the division. The city of the team is based on where the stadium is located.

- teamId → {teamName, city, stadium, superBowls, division, headCoach, manager, owner, record}
- teamName → {teamId, city, stadium, superBowls, division, headCoach, manager, owner, record}
- city → {division}
- stadium → {teamName, city}
    - Therefore using the transitivity axiom its closure
    - stadium → {teamName, teamId, city, stadium, superBowls, division, headCoach, manager, owner, record}
- headCoach → {teamName}
    - Therefore using the transitivity axiom its closure is all the attributes in Team:
    - headCoach → {teamId, teamName, city, stadium, superBowls, division, manager, owner, record}

Stadium

- stadiumId → {stadiumName, city, state, capacity, homeTeam}
- city → {state}

Defense

Each defense has its own teamId and defenseId, but both are needed to determine the unique defense.

- teamId, defenseId → {interceptions, sacks, rshYdsAllowed, passYdsAllowed}

This is in 3NF because teamId, defenseId is a superkey

Quarterback

The qId represents the unique player with the position of quarterback, and when it is combined with the teamId, it represents each unique player on a specific team. Since the qId represents each unique player, we can also determine the player's name. Since qId and teamId are both primary keys, we need both to imply all the other attributes.

- qId, teamId → {name, passingYards, rushingYards, completions, tds}
- qId → {name}

Step 1: Make RHS of each FD into single attributes

- qId, teamId → {name}
- qId, teamId → {passingYards}
- qId, teamId → {rushingYards}
- qId, teamId → {completions}
- qId, teamId → {tds}
- qId → {name}

Step 2: Eliminate redundant attributes from LHS

- qId → {name}
- qId, teamId → {passingYards}
- qId, teamId → {rushingYards}
- qId, teamId → {completions}
- qId, teamId → {tds}

Step 3: Delete redundant FDs from Quarterback

- qId → {name}
- qId, teamId → {passingYards}
- qId, teamId → {rushingYards}
- qId, teamId → {completions}
- qId, teamId → {tds}

Merged (Minimal Cover)

- qId, teamId → {passingYards, rushingYards, completions, tds}
- qId → {name}

QB1 (qId, name)

- qId → {name}

QB2 (qId, teamId, passingYards, rushingYards, completions, tds)

- qId, teamId → {passingYards, rushingYards, completions, tds}

QB1 and QB2 are now in 3NF

WideReceiver

Each wide receiver has its own wId and teamId to be unique. Similar to Quarterback, the player's name can be found with the wId.

- wId, teamId → {name, recYds, catches, tds, fumbles, drops}
- wId → {name}

Step 1: Make RHS of each FD into single attributes

- wId, teamId → {name}
- wId, teamId → {recYds}
- wId, teamId → {catches}

- wId, teamId → {tds}
- wId, teamId → {fumbles}
- wId, teamId → {drops}
- wId → {name}

Step 2: Eliminate redundant attributes from LHS
- wId → {name}
- wId, teamId → {recYds}
- wId, teamId → {catches}
- wId, teamId → {tds}
- wId, teamId → {fumbles}
- wId, teamId → {drops}

Step 3: Delete redundant FDs from WideReceiver
- wId → {name}
- wId, teamId → {recYds}
- wId, teamId → {catches}
- wId, teamId → {tds}
- wId, teamId → {fumbles}
- wId, teamId → {drops}

Merged (Minimal Cover)
- wId, teamId → {recYds, catches, tds, fumbles, drops}
- wId → {name}

WR1 (wId, name)
- wId → {name}

WR2 (wId, teamId, recYds, catches, tds, fumbles, drops)
- wId, teamId → {recYds, catches, tds, fumbles, drops}

WR1 and WR2 are now in 3NF

RunningBack
Similar to wide receiver, each running back must have its own unique rId and teamId to be unique. Similar to WideReceiver, the player's name can be found with the position id.
- rId, teamId → {name, recYds, rshYds, catches, tds, fumbles, drops}
- rId → {name}

Step 1: Make RHS of each FD into single attributes
- rId, teamId → {name}
- rId, teamId → {recYds}
- rId, teamId → {rshYds}
- rId, teamId → {catches}

- rId, teamId → {tds}
- rId, teamId → {fumbles}
- rId, teamId → {drops}
- rId → {name}

Step 2: Eliminate redundant attributes from LHS
- rId → {name}
- rId, teamId → {recYds}
- rId, teamId → {rshYds}
- rId, teamId → {catches}
- rId, teamId → {tds}
- rId, teamId → {fumbles}
- rId, teamId → {drops}

Step 3: Delete redundant FDs from RunningBack
- rId → {name}
- rId, teamId → {recYds}
- rId, teamId → {rshYds}
- rId, teamId → {catches}
- rId, teamId → {tds}
- rId, teamId → {fumbles}
- rId, teamId → {drops}

Merged (Minimal Cover)
- rId, teamId → {recYds, rshYds, catches, tds, fumbles, drops}
- rId → {name}

RB1 (rId, name)
- rId → {name}

RB2 (rId, teamId, recYds, rshYds, catches, tds, fumbles, drops)
- rId, teamId → {recYds, rshYds, catches, tds, fumbles, drops}

RB1 and RB2 are now in 3NF

Kicker
Similar to wide receiver, each kicker must have its own unique kId and teamId to be unique. Similar to WideReceiver, the player's name can be found with the position id.
- rkd, teamId → {name, fgMade, fgMissed}
- kId → {name}

Step 1: Make RHS of each FD into single attributes
- kId, teamId → {name}
- kId, teamId → {fgMade}

- kId, teamId → {fgMissed}
- kId → {name}

Step 2: Eliminate redundant attributes from LHS
- kId → {name}
- kId, teamId → {fgMade}
- kId, teamId → {fgMissed}

Step 3: Delete redundant FDs from Kicker
- kId → {name}
- kId, teamId → {fgMade}
- kId, teamId → {fgMissed}

Merged (Minimal Cover)
- kId, teamId → {fgMade, fgMissed}
- kId → {name}

K1 (kId, name)
- rId → {name}

K2 (kId, teamId, fgMade, fgMissed)
- kId, teamId → {fgMade, fgMissed}

K1 and K2 are now in 3NF

PlaysFor, PlaysAt, Contains
These relationships are made up of the primary keys of the entities that they connect. For PlaysFor, each team can have multiple running backs, wide recievers, kickers, and quarterbacks. For PlaysAt, each team has one home stadium. For Contains, each time contains on defensive line. Since no single attribute is a key, none of the three relationshopw would have an non-trivial functional dependencies.

Assumptions:

Each team contains at least one one of each position. Every position plays for a team. Every defense plays for a team. Every stadium has a home team. Each team only has one defense. No stadium shares home teams, each home team plays at a unique stadium. We store positions instead of players. This means that a player plays only one position. There is no instance of someone playing multiple positions. All the teams participate in the regular season (A record of 0 does not necessarily mean they did not participate in the season).

Example Queries:

Get the teamid and average of values for the "recYds" stat for each team in the "NFC" division:

SELECT DISTINCT t.teamId, AVG(w.recYds) FROM Team t, WideReceiver w
WHERE t.divison = “NFC”, w.teamId = t.teamId
GROUP BY w.teamId;

$_{teamId}G_{avg(recYds)}\,(\sigma_{division = \text{“NFC”}}\,(\text{Team} \bowtie \text{WideReceiver}))$

$\{x \mid \exists\, t \in \text{Team}\ (x[teamId] = t[teamId] \wedge t[division] = \text{“NFC”})$
$\quad \wedge \exists\, w \in \text{WideReceiver}\ (w[teamId] = t[teamId] \wedge w[redYds] = 10)\}$

*Note: Since we did not learn how to use aggregate functions for TRC, it was impossible to fully express this example query. In this query we will be we will be getting the teamids of team in the NFC that have at least one wide receiver on their rosters with receiving yards that are greater than 10.

Get teamid of every team in the "NFC" division:

SELECT t.teamId FROM Team t WHERE t.division = “NFC”;

$\pi_{teamId}\,(\sigma_{division = \text{“NFC”}}\,(\text{Team}))$

$\{x\ \exists \wedge \neg \mid \in$

$\{x \mid \exists\, t \in \text{Team}\ (x[teamId] = t[teamId] \wedge t[division] = \text{“NFC”})\}$

Get the winner of the “AFC” division where the winning team has the best record:

SELECT t1.teamName FROM Team t1
WHERE t1.division = “AFC”
AND t1.record = (SELECT MAX(t.record)
FROM Team t
WHERE t.teamId = t1.teamId)

$\pi_{teamName}\,(_{teamName}G_{max(record)}\,(\sigma_{division = \text{“AFC”}}\,(\text{Team})))$

$\{x \mid \exists\, t \in \text{Team}\ (x[teamId] = t[teamId] \wedge t[division] = \text{“AFC”} \wedge t[record] = 100)\}$

*Note: Since we did not learn how to use aggregate functions for TRC, it was impossible to fully express this example query. In this query we will be we will be getting the teamids of team in the AFC that have a record of 100.

Get the name of the team that has the kicker who has the highest field goals made:

SELECT t.teamName FROM Team t, Kicker k
WHERE k.teamId = t.teamId
AND k.fgMade = (SELECT MAX(k1.fgMade)
FROM Kicker k1
WHERE k1.teamId = k.teamId);

$\pi_{teamName}\,(_{teamName}G_{max(fgMade)}\,(\text{Team} \bowtie \text{Kicker}))$

$\{x \mid \exists\, t \in \text{Team}\ (x[teamId] = t[teamId])$
$\quad \wedge \exists\, k \in \text{Kicker}\ (k[teamId] = t[teamId] \wedge k[fgMade] = 10)\}$

*Note: Since we did not learn how to use aggregate functions for TRC, it was impossible to fully express this example query. In this query we will be we will be getting the teamids of team that have at least one kicker on their rosters with more than 10 fieldgoals made.

Find the number of running backs that the Chargers have on their roster

SELECT COUNT(r.rId) FROM Team t, RunningBack r
WHERE t.teamName = “Chargers” AND r.teamId = t.teamId;

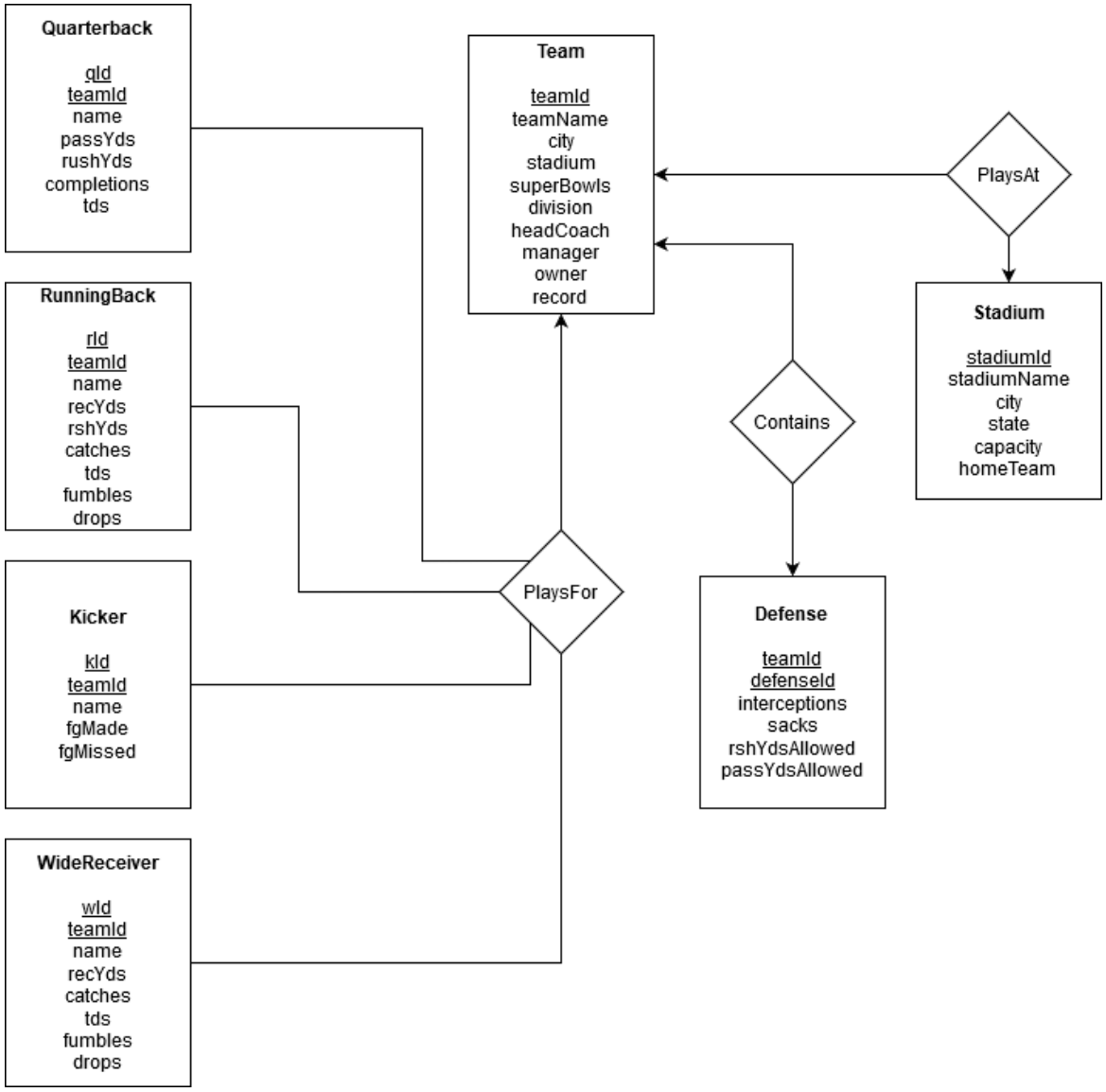
$_{teamName}G_{count(rId)}\,(\sigma_{teamName\,=\,\text{“Chargers”}}\,(\text{Team} \bowtie \text{RunningBack}))$

$\{x \mid \exists\, r \in \text{RunningBack}\ (x[rId] = r[rId])$

$\wedge\, \exists\, t \in \text{Team}\ (r[teamId] = t[teamId] \wedge t[teamName] = \text{“Chargers”})\}$

*Note: Since we did not learn how to use aggregate functions for TRC, it was impossible to fully express this example query. In this query we will be we will be getting the rids of the running backs that the Chargers have on their roster.

ER Diagram:



Data Generation

The data for our project was a mix of real data and pseudo generated data. We obtained all the real data by web scraping. We then pseudo generated the rest of the data. For things like the different stats, we predetermined a range of realistic possible values and used python's random library to generate values in those ranges. We also generated all the names by first web scraping popular first and last names, then randomly combining a first and last name. Then once all the data was generated, we formatted everything into a Dictionary data structure which was then formatted into a json to give to our backend. We decided to generate a portion of our data because we found it difficult to:

A) Find the right data to match our ER diagram
B) Different data sources differed slightly in the way the formatted and it would have been difficult deciphering those differences
C) Convenience, we had to build out a lot of our backend and set up the ORM and actually work on the Database stuff, didn't want to bottleneck our work with data gathering

Web Application

To allow users to interact with our data, we used a web application. We used React to render different queries on a webpage. On the webpage, there are buttons with queries such as:

- Find the winner of the NFC division
- Find the names of all the teams in the NFC division
- Find the kicker with the most field goals made

If a user selects these buttons, they are displayed with an answer. To accomplish this, the frontend queries the backend with API requests. The API request is set up in the backend. Once the frontend fetches the data, the backend returns a .json with the answer. The frontend then parses the .json to text and renders it on the webpage. We use CSS to style the elements on the page so that it looks nice. In the future, we would like to present the user with more options such as choosing different divisions. We could then track the user choice and send a custom api request. This eliminates the need for numerous hardcoded api requests.

Implementation/ Back End

To deliver data to the web application, we used a Java Spring Boot backend. This backend allowed us to do a couple things:

- Parse our generated data
- Convert data into entities
- Insert entities into our database tables
- Query our database to retrieve data

- Serialize this data into a JSON format
- Transmit the JSON to the front end

To accomplish these tasks we made use of a variety of methods within Spring. Firstly, we used Google's JSON simple library to aid us in parsing the JSON file which was generated. Then, after understanding the data, we then created Java entities to represent the different tables and their data. Within these entities, we had to indicate the relationship between the different entities so our tables would have relations between entities. For example we had to indicate that the relationship between a team and a stadium was one to one, and the relationship between a team and a quarterback was one to many. To do this, we had to use annotations built into the Java Persistence library. We also had to indicate the cascading methodology to let the database know how to respond if any data was deleted or updated. After the creation of these entities, we inserted data into the entities and then into the database.

The database we chose to use was a Hikari database. Hikari is a lighter database that is very useful when prototyping applications. We chose to use this database for a number of reasons. Firstly is its integration with the Spring Boot software. As an in-memory database, Hikari gave us speed and flexibility when using Spring to both load data into the database as well as when we queried the database to return data. Additionally, there was no need with the Hikari database to have a separate database software listening on a port allowing us to save system resources when running the application. Finally, since Hikari came bundled with the Java Persistence Library, we did not need to have any additional software to use it. While this database will not scale for commercial applications, for a class project, this is adequate.

To send data to the front end, we first needed to retrieve the information from the database. Spring allows us a couple methods of doing this. We can get all entries from a table, we can get a specific entry based upon the id of that entry, or we can get entries/data corresponding to a JPQL query. We used the first and last for this project, although the first was simply for debugging purposes so we could ensure that data was loaded into the database properly and that the JSON serializer was performing appropriately. After making the queries to the database via JPQL, we sent data to a Spring REST Controller. The REST controller then listens for HTML Get requests on port 8080 on the host machine. After receiving a request on that port, the REST controller then triggers the query and delivers the queried data to the front end.

Our Code

A link to the GitHub where all of the code that was used for this project can be found here: https://github.com/blazepower/CSDS-341-Final-Project

What We Learned

Throughout the course of this project we were able to deeper explore various topics ranging from data fetching to design schema. More specifically, we learned how to fetch data using API requests on the frontend. This is how we were able to create the frontend for our database. We also learned how to parse .JSON files, on both the front and back end, and render elements on a web page. While collecting the data to be used for our database, we learned about web scraping and data formatting. Throughout this class we gained a much better understanding of how to design schemas and set up ER diagrams, we had to go through a few different iterations of our ER diagram before we settled on something that worked. This reinforced the topics that we learned in class regarding ER diagrams and relational schemas. Lastly, we learned how to set up a database with relations between the different tables.

Contributions:

Rishik Hombal:
Implementation of Database, Queries

Aman Bhriguvansh:
ER Diagram design, data constraints

Lauren Sharkey:
Data visualization, Queries

Jack Zhang:
Data generation, Schema design