# Lane Barton
# CS 586
# Grad Project Final Submission
# Fantasy Football Database

## Table of Contents

## ER Diagram

The only change I made to my diagram since my second submission was to add a regular_season table, which is highlighted in Red in Figure 1 (a full diagram that is a bit small is in Figure 2)

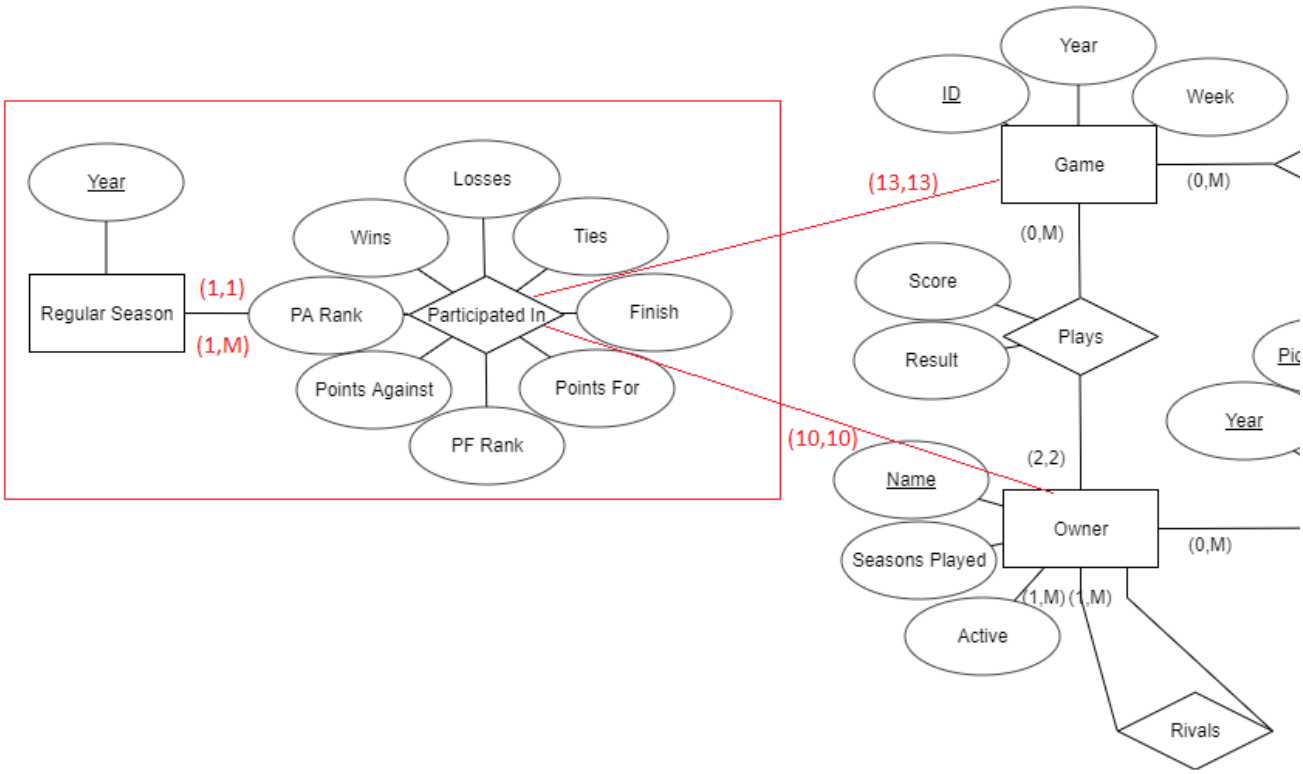


*Figure 1: Final ER Diagram, Regular Season entity added*

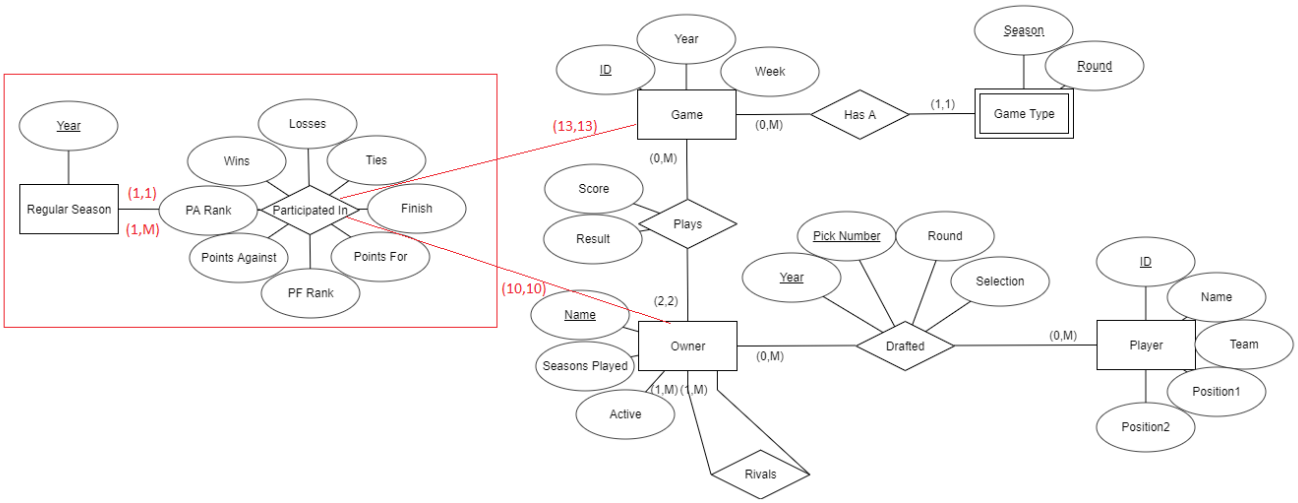


*Figure 2: Full ER Diagram (small text because of size)*

I wasn't exactly sure how to mix this in, since the table is really an aggregation of games and the "regular season" entity is really just relative to the year. But I included all the attributes I used in the actual table in the relationship and used the correct relationship numbering (a season has exactly 13

games and 10 owners, a owner has at least one and potentially many regular seasons, and a game has exactly one regular season).

As an overview of what I had previously, there are owners that have a name, number of seasons played, and a status as active or not. Each owner may have some rivals, at least one and potentially many more. Each owner has participated in a draft where they have selected a player. A player has a name, team, and up to two positions, while each player drafted by an owner has an associated year, pick number, selection, and round where the player was picked. Additionally, there is a game entity which has a year and week associated with it. Each game has a game_type as a weak entity, where each game type has a season type (regular or post season) and round (regular season, bottom 4, wildcard, consolation, 3rd place, or championship). Finally, each owner is involved in many games with an associated score and result (W/L/T), while each game has exactly two owners.

## Application Overview

The application I created uses a Node JS environment, a library called Knex JS which serves as a SQL query builder without having to write SQL code over and over again, a PapaParse library to parse CSV files for data insertion, and a library called Express JS which helps create and manage a server so I could query my database once it was setup.

### Knex JS - Creating Tables

Knex JS is something I had used before and was very excited to play with for this project. After installing the library, you can configure a connection to the database by setting a connection object (see the file “knex/config.js” for a specific example). Knex can use that connection to query the database and interact with it. See the Knex.js homepage for some additional context into what I used (*Knex.js*). As a reminder, I created a Postgres database on my local computer and had pgAdmin3 to help me view it, so attempting to run the application with the code I submit might not be viable. Plus, there are a few inconsistent things I did, so the code that will reference is more for explanatory and demonstrative purposes as to what I did.

More specifically, Knex has functionality for schema building. I created a migration file which uses Knex functions to create a table using much easier to understand syntax (Figure 3 on the next page).

```
exports.up = async function (knex, Promise) {
    return Promise.all([
        knex.schema.createTable('owner', table => {
            table.increments('owner_id').primary();
            table.string('name').unique().notNullable();
            table.integer('seasons_played').notNullable();
            table.boolean('active').notNullable();
          }),

        knex.schema.createTable('rival', table => {
            table.integer('owner_id');
            table.integer('rival_id');
        }),

        knex.schema.createTable('game', table => {
            table.increments('game_id').primary();
            table.integer('year').notNullable();
            table.integer('week').notNullable();
            table.integer('game_type').notNullable();
        }),
```

*Figure 3: A sampling of table creation from my "knex/migrations/V1__init.js" migration file*

For example, in the above figure I have three tables (owner, rival, and game) and for owner I created a column called “owner_id” that increments and is a primary key, a column called “name” which is a unique and not nullable string, and more. There are some tradeoffs, however, such as specific CHECK constraints being difficult to create with the native schema building. Luckily, Knex provides the option to do a raw query (“knex.raw(<Some SQL>)”) that can write any SQL statement, allowing for other table creation aspects to be included.

The migrations functionality runs whenever a server starts and will add any new migrations that have not yet been done, meaning you can alter tables, create new tables, or do anything else in the future without having to completely rewrite the tables in the db. Finally, it’s worth noting that I did a lot of my work before deciding to add a “regular_season” table to store aggregate seasonal results, and for the sake of time efficiency I wrote a direct SQL statement in pgAdmin to create that table, meaning that my current code is not sufficient to create the entire schema.

**PapaParse – Inserting Data**

The other nice thing about Knex is that it can very easily insert, delete, or select rows from a db with clear functionality. But to do so, I had to produce values for the Knex functions to insert. For some of the simpler tables, like “owner” or “game_type”where there were only a few rows that weren’t likely to change significantly, I created arrays of values to insert in with Knex’s functionality (see “processSmallTables.js”). But for some dynamic, large data this was not very viable. Enter PapaParse, a simple streaming library which can read a local CSV file and parse the data into objects which can be inserted into the db via Knex. After scraping data from my site, and formatting it in a way that I could process, I would use PapaParse to read the file and then use it’s “chunk” callback to receive all data as a single array of row objects and process it as I desired (see Figure 4 on the next page)

```
1  const config = require('./knex/config.js');
2  const knex = require('knex')(config.knex);
3  const fs = require('fs');
4  const papa = require('papaparse');
5
6  //function main() {
7      const file = fs.createReadStream('regSeasonData.csv');
8
9      const papaConfig = {
10         header: true,
11         dynamicTyping: true,
12         skipEmptyLines: true,
13         //preview: 10,
14         chunk: parseData
15     }
16     papa.parse(file, papaConfig);
17
18 //}
19
20 function parseData(results, parser) {
21     return new Promise(async (resolve, reject) => {
22         const data = [];
23
24         const maxCount = await knex('regular_season')
25             .count()
26             .catch(err => {
27                 console.log(err);
28                 reject('Cannot fetch regular_season values')
29             });
30
31         let max = maxCount[0].max || 0;
32
33         knex.transaction(trx => {
34             knex('regular_season')
35             .insert(results.data.slice(max))
36             .transacting(trx)
37             .then(trx.commit)
38             .catch(trx.rollback);
39         })
40         .then(() => {
41             if (results.data.length > 0) {
42                 console.log((results.data.length - max) + ' new games processed and added to database.')
43             }
44             resolve();
45         })
46         .catch(error => {
47             console.error(error);
48             reject(error)
49         });
50     });
51 }
```

*Figure 4: Contents of the updateSeasons.js file, used to process seasonal data and insert new data*

For example, in the above figure, PapaParse is reading in the ‘regSeasonData.csv’ file at Line 16, and upon it’s “chunk” callback is executing the “parseData” function. This function would check to see how many seasonal rows have been added to the “regular_season” table already by querying the database through Knex (Lines 24-29), then start inserting any rows in the CSV beyond that max value (Line 35). This would add new values only to prevent duplicates – more importantly, it was wrapped in Knex’s transaction syntax so either every new value is inserted or nothing is (Lines 33-38). This process was done similarly for other tables like “game”, “game_score”, or “draft” in the files updateGames.js and updateDrafts.js, respectively to insert all data.

**Express JS – A server for the application to function**

With all the tables created and data inserted, the last aspect was the build an application that can interact with it. I opted for Express JS, something I had never used before but had heard about. I followed the general formatting from a tutorial online (Tod) to create a basic HTML page, modifying it to use a text box for an SQL command and a submit button to run the query. Using the tutorial (Tod), I was able to setup an event listener for the submit button (see “public/app.js”) which in turn sends a

request with the SQL statement to a Express JS server. This server, located in “public/index.js” and once again designed with help from the tutorial (Tod), processes the request sent in and runs a function called “getQueryResults”. This was a function created exclusively by myself (see “rawquery.js” or

```
1  const knex = require('knex')(require('./knex/config.js').knex);
2
3  const getQueryResults = (query) => {
4      return new Promise((resolve, reject) => {
5          let results;
6          if (query === null || query === "") {
7              reject('Error - please enter a valid query before submitting');
8          }
9
10         const create = query.toLowerCase().includes('create table');
11         const alter = query.toLowerCase().includes('alter table');
12         const drop = query.toLowerCase().includes('drop table');
13         const insert = query.toLowerCase().includes('insert into');
14         const del = query.toLowerCase().includes('delete from');
15
16
17         if(create || alter || drop || insert || del) {
18             reject('Error - database altering queries are not allowed');
19         }
20
21         console.log(query)
22         knex.raw(query)
23             .then(result => {
24                 console.log(result);
25                 if (result.rows === null || result.rows.length === 0) {
26                     reject('Empty Response')
27                 }
28                 console.log('In query')
29                 let output = '<table><tr>';
30                 //Result should have fields attributes that holds array with column names
31                 const columns = Object.keys(result.rows[0]);
32                 for (let i=0; i < columns.length; i++) {
33                     output += `<th>${columns[i]}</th>`;
34                 }
35
36                 output += '</tr>';
37                 let row  = 0;
38                 while(row < result.rows.length) {
39                     output += '<tr>';
40                     for(let i=0; i < columns.length; i++) {
41                         output += `<td>${result.rows[row][columns[i]]}</td>`;
42                     }
43                     output += '</tr>';
44                     row++;
45                 }
46                 output += `</table><h3>Result returned ${row} rows</h3>`;
47                 resolve(output);
48             })
49             .catch(err => {
50                 reject(`${err}`);
51             });
52     });
53 }
```

*Figure 5: Code for rawquery.js, which takes an SQL statement from the application, queries the database, then turns the results into an HTML table*

Figure 5 above) to take the input SQL statement and run it directly on the database with the knex.raw function (Line 22), but only if it is not a  CREATE, ALTER, DROP, INSERT INTO, DELETE FROM statement to prevent outside manipulation. The results of the query, interpreted as an array of rows, are then processed to create an HTML table with the headers and values for each row. This HTML code is sent back to the event listener in “app.js”, which sets the innerHTML of a div tag to the table HTML (see Figure 6 on the next page). The last step would just be to open up a web browser to localhost:8080 and start using the application.

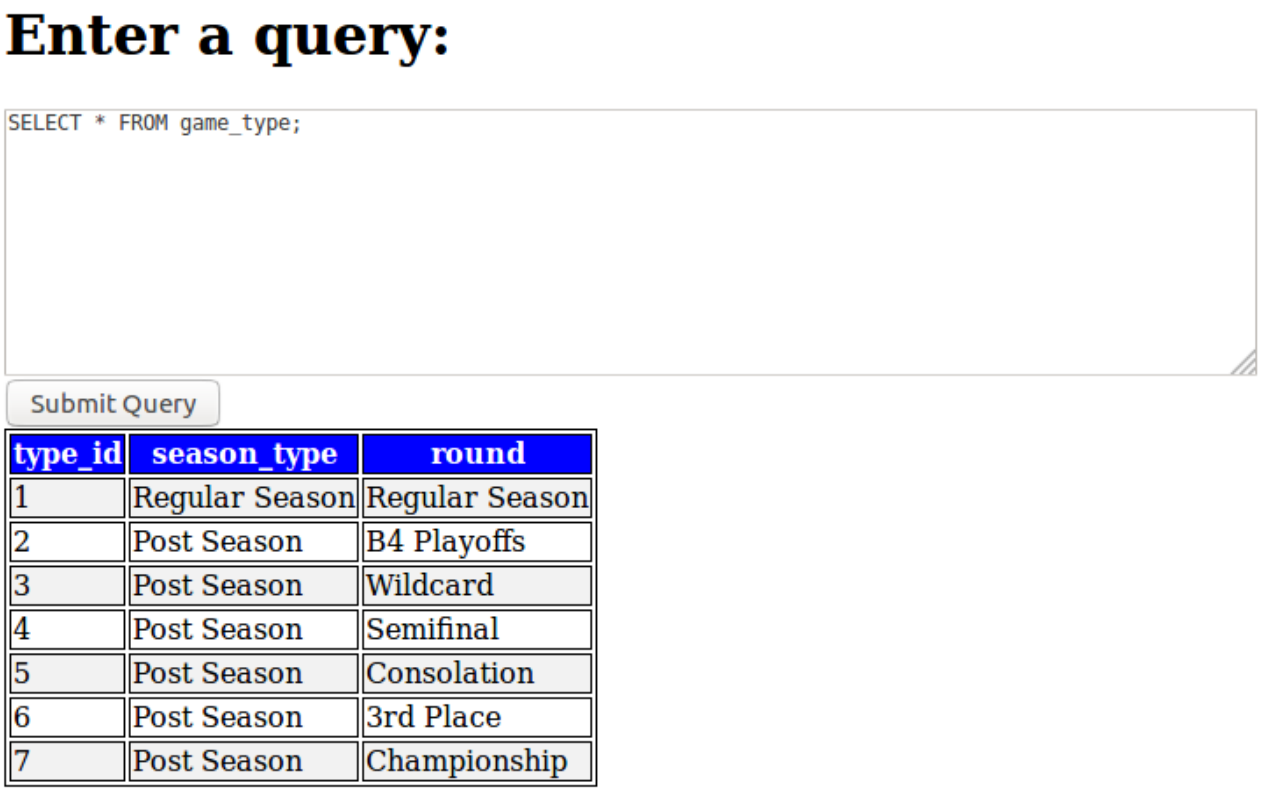


*Figure 5: An demonstration of the application querying and receiving results as a table*

**Node JS – Run everything**

Finally, an aspect that I alluded to but skipped over is the usage of Node JS, which is a JavaScript environment to run my code. It uses dependencies in a “package.json” file to import libraries and can run the scripts created to execute the aspects of the application. For example, to start the server/run migrations from a terminal, I could simply input “node .” in the directory of my project to start the server and also start any migrations in my “knex/migrations” folder. When I wanted to insert/update data, I could run “node ./updateGames.js” (filename is an example – could be whatever script I wanted to run) in my terminal window to insert any new rows that have not been inserted. This is the best aspect of this project – it allows for maintainability and continuous insertion of data by repeated calls to these scripts after updating the associated CSV files with data from new years. Thus, with a combination of Node JS, Knex JS, PapaParse, and Express JS, I could create my tables, populate them, and start a server to allow my application to query the database, producing styled tables as a result.

## Application Demos

Now that an explanation of how my code/application works, it is worth taking the time to give some example inputs/outputs with the application interface to see what happens. Included in this section are four images showing the application with input in the text box and output below, plus a brief description of what is going on.

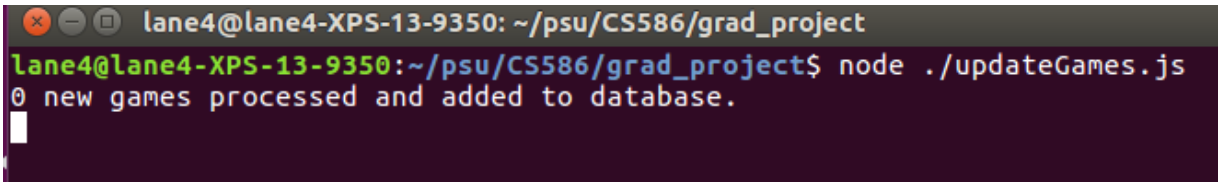


*Figure 6: Demo of running script to update game table*

First, in Figure 6 a quick little example of running a script in a terminal. After navigating to my project directory, I run a node command to run the project. Then I get a little output telling me nothing was added (I have no new games to insert).

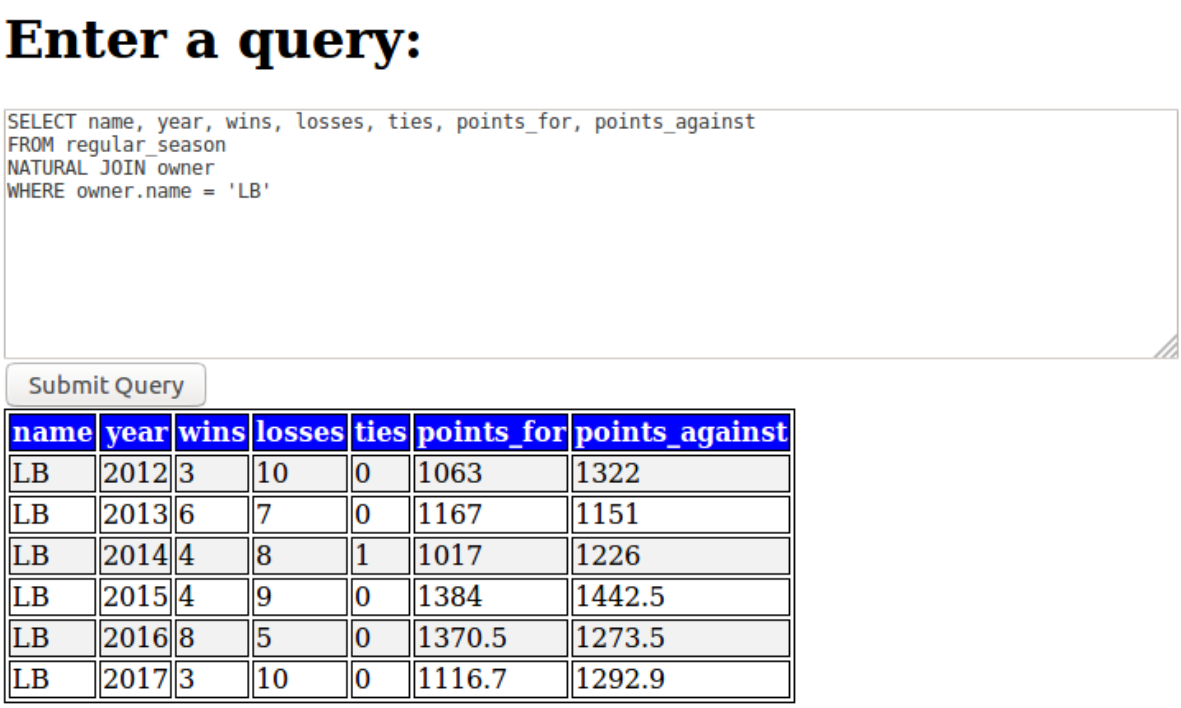


*Figure 7: Demo of a successful application query*

In the above Figure we go to the application to see a successful query. The input SQL statement is processed and turned into an HTML table which is output below. Finally, there is also a statement listing the number of rows returned in case that is relevant as well.

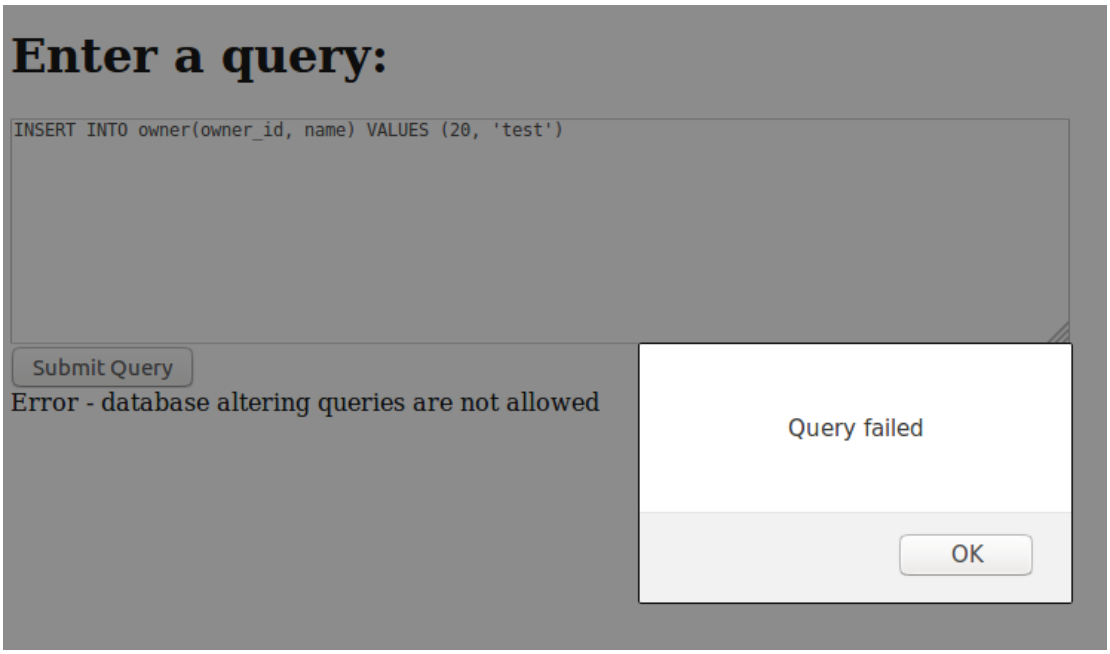

*Figure 8: Demonstrating that creation/deletion queries are (mostly) prevented*

In the above example, an attempt to insert data into the database from the application window is made. I added some minor filters to prevent any overt attempts to modify data to be prevented, responding with a popup and a little message saying “database altering queries are not allowed”. There is also likely a way to make this impossible by creating the Knex connection with a user that does not have INSERT/DELETE/CREATE roles, but I didn’t have time to fully explore that.

*Figure 9: Demo that empty reponse are still possible*

Finally, this last demo in Figure 9 shows that a query that returns nothing will produce an empty response with a litle popup again to indicate a unique result.

## CREATE TABLE Statements

As a note, my use of Knex JS to build my tables made the actual CREATE TABLE statements a little hard to extract, except for the “regular_season” table which I inserted manually with a CREATE TABLE statement. That being said, using pgAdmin I was able to view my tables and a statement does exist for each, even the one’s I made in Knex. Thus, I am opting to copy those values as the exact statement that the database interpreted as opposed to Knex schema functions. To see the Knex function, view the file “knex/migrations/V1__init.js”

Finally, it’s also worth noting that the use of Knex defaults to the “public” schema in the database and the use of migration creates two additional tables, “knex_migrations” and “knex_migrations_lock”. I am choosing to omit the creation statements for those two tables since it isn’t technically a part of my ER diagram/schema design and left the schema as “public” because I was running into some issues when trying to rename to something other than public.

**DRAFT**

```
CREATE TABLE public.draft
(
  year integer NOT NULL,
  selection_id integer NOT NULL,
  round integer NOT NULL,
  pick integer NOT NULL,
  owner_id integer,
  player_name character varying(100) NOT NULL,
  player_team character varying(5) NOT NULL,
  position_1 character varying(5) NOT NULL,
  position_2 character varying(5),
  CONSTRAINT draft_pkey PRIMARY KEY (year, selection_id),
  CONSTRAINT draft_owner_id_foreign FOREIGN KEY (owner_id)
      REFERENCES public.owner (owner_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT valid_pick CHECK (pick >= 1 AND pick <= 10),
  CONSTRAINT valid_position_1 CHECK (position_1::text = ANY (ARRAY['QB'::character varying,
'RB'::character varying, 'WR'::character varying, 'TE'::character varying, 'D/ST'::character varying,
'K'::character varying]::text[])),
  CONSTRAINT valid_position_2 CHECK (position_2::text = ANY (ARRAY['QB'::character varying,
'RB'::character varying, 'WR'::character varying, 'TE'::character varying, 'D/ST'::character varying,
'K'::character varying]::text[])),
  CONSTRAINT valid_position_order CHECK (position_1 IS NOT NULL OR position_2 IS NULL),
  CONSTRAINT valid_round CHECK (round >= 1 AND round <= 18),
  CONSTRAINT valid_selection CHECK (selection_id >= 1 AND selection_id <= 180),
  CONSTRAINT valid_year CHECK (year >= 2012)
)
WITH (
  OIDS=FALSE
);
```

**GAME**

```
CREATE TABLE public.game
```

```
(
  game_id integer NOT NULL DEFAULT nextval('game_game_id_seq'::regclass),
  year integer NOT NULL,
  week integer NOT NULL,
  game_type integer,
  CONSTRAINT game_pkey PRIMARY KEY (game_id),
  CONSTRAINT game_game_type_foreign FOREIGN KEY (game_type)
      REFERENCES public.game_type (type_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT valid_week CHECK (week >= 1 AND week <= 17),
  CONSTRAINT valid_year CHECK (year >= 2012)
)
WITH (
  OIDS=FALSE
);
```

**GAME_SCORE**

```
CREATE TABLE public.game_score
(
  game_id integer NOT NULL,
  owner_id integer NOT NULL,
  score real NOT NULL,
  result character varying(1) NOT NULL,
  CONSTRAINT game_score_pkey PRIMARY KEY (game_id, owner_id),
  CONSTRAINT game_score_game_id_foreign FOREIGN KEY (game_id)
      REFERENCES public.game (game_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT game_score_owner_id_foreign FOREIGN KEY (owner_id)
      REFERENCES public.owner (owner_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
```

**GAME_TYPE**

```
CREATE TABLE public.game_type
(
  type_id integer NOT NULL DEFAULT nextval('game_type_type_id_seq'::regclass),
  season_type character varying(255) NOT NULL,
  round character varying(255) NOT NULL,
  CONSTRAINT game_type_pkey PRIMARY KEY (type_id)
)
WITH (
  OIDS=FALSE
);
```

**OWNER**

```
CREATE TABLE public.owner
```

```
(
  owner_id integer NOT NULL DEFAULT nextval('owner_owner_id_seq'::regclass),
  name character varying(255) NOT NULL,
  seasons_played integer NOT NULL,
  active boolean NOT NULL,
  CONSTRAINT owner_pkey PRIMARY KEY (owner_id),
  CONSTRAINT owner_name_unique UNIQUE (name),
  CONSTRAINT non_negative_seasons CHECK (seasons_played >= 0)
)
WITH (
  OIDS=FALSE
);
```

**REGULAR_SEASON**

```
CREATE TABLE public.regular_season
(
  year integer NOT NULL,
  owner_id integer NOT NULL,
  wins integer NOT NULL,
  losses integer NOT NULL,
  ties integer,
  finish integer NOT NULL,
  points_for real NOT NULL,
  pf_rank integer NOT NULL,
  points_against real NOT NULL,
  pa_rank integer NOT NULL,
  CONSTRAINT regular_season_pkey PRIMARY KEY (year, owner_id),
  CONSTRAINT regular_season_owner_id_foreign FOREIGN KEY (owner_id)
      REFERENCES public.owner (owner_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
  CONSTRAINT valid_losses CHECK (losses >= 0 AND losses <= 13),
  CONSTRAINT valid_pa_rank CHECK (pa_rank >= 0 AND pa_rank <= 10),
  CONSTRAINT valid_pf_rank CHECK (pf_rank >= 0 AND pf_rank <= 10),
  CONSTRAINT valid_wins CHECK (wins >= 0 AND wins <= 13),
  CONSTRAINT valid_year CHECK (year >= 2012)
)
WITH (
  OIDS=FALSE
);
```

**RIVAL**

```
CREATE TABLE public.rival
(
  owner_id integer NOT NULL,
  rival_id integer NOT NULL,
  CONSTRAINT rival_pkey PRIMARY KEY (owner_id, rival_id),
  CONSTRAINT rival_owner_id_foreign FOREIGN KEY (owner_id)
      REFERENCES public.owner (owner_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION,
```

```
  CONSTRAINT rival_rival_id_foreign FOREIGN KEY (rival_id)
      REFERENCES public.owner (owner_id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITH (
  OIDS=FALSE
);
```

## Questions to SQL

The following include my original list of 20 questions, their corresponding SQL statements, and a table for the output. It should be noted that many questions ask something like “for a given __ “ or “in a specific ___”, in which case I chose a single value for the field by filtering in the WHERE clause. It is very easy to tweak this to show results for all values by simply removing the filter. Also, a few questions changed, mostly because I did not end up producing a table for aggregate values corresponding the post-season. For those questions I either omitted the question or replaced it with something similar that was viable to determine by knowing that postseason games are all Week 14 or later in our league.

**1. Who has participated in this fantasy football league? Are they currently participating in the league?**

```
SELECT name, active FROM owner;
```

| name | active |
|---|---|
| LB | true |
| DG | true |
| OB | true |
| CD | true |
| WF | true |
| CN | true |
| NN | true |
| ML | true |
| DK | true |
| PD | false |
| RW | true |

**2. What is the record for a given owner in a specific year?**

```
SELECT rs.year, o.name, rs.wins, rs.losses, rs.ties
FROM owner AS o NATURAL JOIN regular_season AS rs
WHERE o.name = 'LB'
AND rs.year = '2013';
```

| year | name | wins | losses | ties |
|---|---|---|---|---|
| 2013 | LB | 6 | 7 | 0 |

**3. What is ranking for a given owner in a specific year after the regular season? ~~After the postseason?~~**

NOTE: I omitted the second part of the data given that I was having issues creating a “post_season” table to hold general values.

```
SELECT o.name, rs.finish
FROM regular_season AS rs
NATURAL JOIN owner AS o
WHERE o.name = 'LB'
AND rs.year = 2016
```

| name | finish |
|---|---|
| LB | 2 |

**4. Who are the rivals of a given team owner?**

```
SELECT own.name AS "owner", riv.name AS "rival"
FROM rival AS r
JOIN owner AS own ON (own.owner_id = r.owner_id)
JOIN owner AS riv ON (riv.owner_id = r.rival_id)
WHERE own.name = 'LB';
```

| owner | rival |
|---|---|
| LB | DG |
| LB | DK |

**5. What is an owner’s record all-time against their rivals?**

```
SELECT o.name, rm.result, COUNT(rm.result)
FROM (SELECT os.owner_id AS "owner_id", rs.owner_id AS "rival_id", os.result FROM
game_score AS os JOIN
game_score AS rs ON (os.game_id = rs.game_id AND os.owner_id !=rs.owner_id)
JOIN rival AS r ON (r.owner_id = os.owner_id AND r.rival_id = rs.owner_id)) AS rm
NATURAL JOIN owner AS o
GROUP BY o.name, rm.result
ORDER BY o.name, rm.result DESC;
```

| name | result | count |
|---|---|---|
| CD | W | 11 |
| CD | L | 4 |
| CN | W | 14 |
| CN | L | 13 |
| DG | W | 11 |
| DG | L | 2 |
| DK | W | 12 |
| DK | L | 14 |
| LB | W | 6 |
| LB | L | 17 |
| ML | W | 10 |

| | | |
|---|---|---|
| ML | L | 6 |
| NN | W | 12 |
| NN | L | 12 |
| OB | W | 4 |
| OB | L | 11 |
| PD | W | 1 |
| PD | L | 1 |
| RW | W | 4 |
| RW | L | 7 |
| WF | W | 7 |
| WF | L | 5 |

**6. What is an owner's record all-time against another specific owner?**

```
SELECT o.name, vs.result, COUNT(vs.result)
FROM (SELECT os.owner_id AS "owner_id", rs.owner_id AS "opp_id", os.result FROM
game_score AS os JOIN
game_score AS rs ON (os.game_id = rs.game_id AND os.owner_id !=rs.owner_id)) AS vs
JOIN owner AS o ON (o.owner_id = vs.owner_id)
JOIN owner AS r ON (r.owner_id = vs.opp_id)
WHERE o.name = 'LB'
AND r.name = 'ML'
GROUP BY o.name, vs.result
ORDER BY o.name, COUNT(vs.result);
```

| name | result | count |
|---|---|---|
| LB | W | 1 |
| LB | L | 8 |

**7. Who has the best regular season record all-time?**

NOTE: I opted for average win percentage excluding ties as the measurement...theoretically this question could have been asking for a specific W/L/T record, but I felt that win-percentage is better given that some owners have played for a different number of years

```
SELECT wp.name, wp.win_percentage
FROM (SELECT o.name, CAST(SUM(rs.wins) AS float)/CAST(SUM(rs.wins+rs.losses) AS float) AS
"win_percentage"
FROM owner AS o
NATURAL JOIN regular_season AS rs
GROUP BY o.name) AS wp
WHERE wp.win_percentage = (SELECT MAX(wp.win_percentage)
FROM (SELECT o.name, CAST(SUM(rs.wins) AS float)/CAST(SUM(rs.wins+rs.losses) AS float) AS
"win_percentage"
FROM owner AS o
NATURAL JOIN regular_season AS rs
GROUP BY o.name) AS wp)
```

| name | win_percentage |
|---|---|
| ML | 0.602564102564103 |

**8. Who has the ~~best postseason record~~ most post-season wins of all-time?**
NOTE: I modified this to “Most postseason wins” since the lack of an aggregate table for the regular season would have made it extremely difficult to do – it would require summing by W results, summing by L results, casting a win percentage, then finding the max win percentage from there, which seemed excessive given that the guy with the most wins certainly has the best record.

```
SELECT ps.name,  ps.count AS "Wins"
FROM(SELECT o.name, gs.result, COUNT(gs.result)
FROM game AS g
NATURAL JOIN owner AS o
NATURAL JOIN game_score AS gs
WHERE g.week > 13
GROUP BY o.name, gs.result) AS ps
WHERE ps.count >= ALL (SELECT COUNT(gs.result)
FROM game AS g
NATURAL JOIN owner AS o
NATURAL JOIN game_score AS gs
WHERE g.week > 13
AND gs.result = 'W'
GROUP BY o.name, gs.result)
```

| name | Wins |
|---|---|
| CD | 16 |

**9. What is the average score for a given week?**

```
SELECT g.week, AVG(gs.score)
FROM game AS g
NATURAL JOIN game_score AS gs
WHERE g.week = 1
GROUP BY g.week
ORDER BY g.week;
```

| week | avg |
|---|---|
| 1 | 101.081666564941 |

**10. What is the least points scored in a single game? The most?**

```
SELECT MIN(gs.score)
FROM game_score AS gs
```

| min |
|---|
| 26 |

```
SELECT MAX(gs.score)
FROM game_score AS gs
```

| max |
|---|
| 181.5 |

**11. What is the most points scored in a single ~~season~~ regular season?**
NOTE: The following was adjusted to the just the regular season because of some issues with post-season data (namely the 1 and 2 seeds receive a bye in Week 14 and that data is not included, so omitting the postseason provides a level field for everyone)

```
SELECT MAX(points_for)
FROM regular_season;
```

| max |
|---|
| 1529 |

**12. What is the average drafting position of a given owner (where they usually pick in the draft)?**

```
SELECT o.name, AVG(d.pick)
FROM owner AS o
NATURAL JOIN draft AS d
WHERE d.round = 1
GROUP BY o.name
ORDER BY avg;
```

| name | avg |
|---|---|
| LB | 2.8333333333333333 |
| WF | 3.1666666666666667 |
| RW | 3.7500000000000000 |
| DK | 5.6666666666666667 |
| CN | 5.8333333333333333 |
| NN | 5.8333333333333333 |
| OB | 6.0000000000000000 |
| DG | 6.3333333333333333 |
| PD | 6.5000000000000000 |
| ML | 7.0000000000000000 |
| CD | 7.6666666666666667 |

**13. What is the average drafted position of a given player (where are they usually picked in the draft)?**

```
SELECT player_name, AVG(selection_id), COUNT(player_name) AS "Years Drafted"
FROM draft
WHERE player_name = 'Antonio Brown'
OR player_name = 'Ben Roethlisberger'
OR player_name = 'Steelers D/ST'
GROUP BY player_name
ORDER BY avg;
```

| player_name | avg | Years Drafted |
|---|---|---|
| Antonio Brown | 25.5000000000000000 | 6 |
| Ben Roethlisberger | 101.3333333333333333 | 6 |
| Steelers D/ST | 134.6666666666666667 | 3 |

**14. What player position does a given owner usually pick in a given round of the draft?**

```
SELECT fr.name, fr.position_1, fr.count
FROM (SELECT o.name, d.position_1, COUNT(d.position_1 )
FROM owner AS o
NATURAL JOIN draft AS d
WHERE o.name = 'LB'
AND d.round = 1
GROUP BY o.name, d.position_1
ORDER BY o.name) AS fr
WHERE fr.count  = (SELECT MAX(fr.count)
FROM (SELECT o.name, d.position_1, COUNT(d.position_1 )
FROM owner AS o
NATURAL JOIN draft AS d
WHERE o.name = 'LB'
AND d.round = 1
GROUP BY o.name, d.position_1
ORDER BY o.name) AS fr)
```

| name | position_1 | count |
|---|---|---|
| LB | RB | 3 |
| LB | WR | 3 |

**15. When is the earliest in the draft that a kicker has been taken? Which owner made that selection?**

```
SELECT o.name, d.year, d.player_name, d.selection_id
FROM owner AS o
NATURAL JOIN draft AS d
WHERE d.position_1 = 'K'
AND d.selection_id = (
SELECT MIN(selection_id)
FROM draft
WHERE position_1 = 'K')
```

| name | year | player_name | selection_id |
|---|---|---|---|
| NN | 2012 | Stephen Gostkowski | 92 |

**16. Which player(s) has a given owner selected the most all-time?**

```
SELECT dp.name, dp.player_name, dp.count
FROM (SELECT o.name, d.player_name, COUNT(d.player_name)
FROM draft AS d
NATURAL JOIN owner AS o
WHERE o.name = 'LB'
GROUP BY o.name, d.player_name) AS dp
WHERE dp.count >= ALL (SELECT COUNT(player_name)
FROM draft AS d
NATURAL JOIN owner AS o
WHERE o.name = 'LB'
GROUP BY d.player_name)
```

| name | player_name | count |
|---|---|---|
| LB | Lamar Miller | 2 |
| LB | James White | 2 |
| LB | Dwayne Bowe | 2 |
| LB | Tony Romo | 2 |
| LB | Texans D/ST | 2 |
| LB | Patriots D/ST | 2 |
| LB | Marshawn Lynch | 2 |
| LB | Vincent Jackson | 2 |
| LB | Jonathan Stewart | 2 |
| LB | Dan Bailey | 2 |
| LB | Jordy Nelson | 2 |

**17. Who has scored in the most points on average in the postseason?**

```
SELECT avg.name, avg.avg
FROM (SELECT o.name, AVG(gs.score)
FROM game_score AS gs
NATURAL JOIN owner AS o
NATURAL JOIN game AS g
WHERE g.week > 13
GROUP BY o.name) AS avg
WHERE avg.avg = (SELECT MAX(avg.avg)
FROM (SELECT o.name, AVG(gs.score)
FROM game_score AS gs
NATURAL JOIN owner AS o
NATURAL JOIN game AS g
WHERE g.week > 13
GROUP BY o.name) AS avg)
```

| name | avg |
|---|---|
| CD | 127.31666692098 |

**18. What is the average points scored by all teams for each season?**

```
SELECT g.year, AVG(gs.score)
FROM game AS g
NATURAL JOIN game_score AS gs
GROUP BY g.year
ORDER BY g.year;
```

| year | avg |
|---|---|
| 2012 | 94.9620253164557 |
| 2013 | 98.2088607594937 |
| 2014 | 94.620253164557 |
| 2015 | 102.193670659126 |
| 2016 | 100.255695994896 |
| 2017 | 96.4367087641849 |

**19. Which owner has made the playoffs the most times?**
As a note, the bottom four teams end up in a round-robin “playoffs” - the real playoffs is a knockout bracket. Since the first round includes two byes (and the data is not in the game or game_score table), I checked for all games in Week 15 that were not in the B4 Playoffs to guarantee a count of playoff teams.

```
SELECT pc.name, pc.count
FROM (SELECT o.name, COUNT(o.name)
FROM game AS g
NATURAL JOIN game_score AS gs
JOIN game_type AS gt ON (gt.type_id = g.game_type)
NATURAL JOIN owner AS o
WHERE g.week = 15
AND gt.round != 'B4 Playoffs'
GROUP BY o.name) AS pc
WHERE pc.count = (SELECT MAX(pc.count)
FROM (SELECT o.name, COUNT(o.name)
FROM game AS g
NATURAL JOIN game_score AS gs
JOIN game_type AS gt ON (gt.type_id = g.game_type)
NATURAL JOIN owner AS o
WHERE g.week = 15
AND gt.round != 'B4 Playoffs'
GROUP BY o.name) AS pc)
```

| name | count |
|---|---|
| ML | 6 |

**20. What is a given owner’s max, min, and average finishing position in the regular season?**

```
SELECT name, MIN(finish) AS "Best Finish", MAX(finish) AS "Worst Finish", AVG(finish)
FROM regular_season
NATURAL JOIN owner
GROUP BY name
ORDER BY avg
```

| name | Best Finish | Worst Finish | avg |
|---|---|---|---|
| ML | 1 | 6 | 3.5000000000000000 |
| DG | 1 | 7 | 4.5000000000000000 |
| CD | 3 | 7 | 4.6666666666666667 |
| DK | 3 | 10 | 4.8333333333333333 |
| OB | 1 | 10 | 5.3333333333333333 |
| WF | 1 | 9 | 5.6666666666666667 |
| NN | 2 | 9 | 5.6666666666666667 |
| RW | 1 | 10 | 5.8000000000000000 |
| CN | 2 | 10 | 6.8333333333333333 |
| LB | 2 | 9 | 7.5000000000000000 |
| PD | 10 | 10 | 10.0000000000000000 |

## Listing of All Data

Finally, I believe that I talked to you in class and was told that I did not in fact have to list the contents of all tables as part of this assignment, but it is mentioned in the third submission details. So as an intermediate step, here is a listing of the count of all tables:

```
(SELECT 'owner' AS "table", COUNT(*) FROM owner)
UNION (SELECT 'game_score' AS "table", COUNT(*) FROM game_score)
UNION (SELECT 'game_type' AS "table", COUNT(*) FROM game_type)
UNION (SELECT 'rival' AS "table", COUNT(*) FROM rival)
UNION (SELECT 'draft' AS "table", COUNT(*) FROM draft)
UNION (SELECT 'regular_season' AS "table", COUNT(*) FROM regular_season)
ORDER BY "table"
```

| table | count |
|---|---|
| draft | 1075 |
| game_score | 948 |
| game_type | 7 |
| owner | 11 |
| regular_season | 60 |
| rival | 16 |

I will also attach another PDF with a listing of all tables to my submission email just in case that is required.

**WORKS CITED**

Knex JS. Knex.js - A SQL Query Builder. 18 July 2018, knexjs.org/.

Tod, Robert. "Tutorial: Setting up Node.js with a database." Hackernoon.com, 7 May 2017, hackernoon.com/setting-up-node-js-with-a-database-part-1-3f2461bdd77f.