**NBA Regular Season vs. Playoffs Stats Comparison**

**Question**:

How do NBA teams perform in the playoffs vs the regular season? Is there a big difference between games played in the playoffs and regular season?

* Do teams score more or less points?
* Do teams push the pace more and take more shots? Or does the game slow down as teams approach each game more strategically?
* Do teams settle for jump shots instead of getting easy baskets?
* How does the physicality of the games change?
* Are there any significant differences in how each possession is played in terms of time?
* Is carelessness a point of emphasis in the playoffs?
* Does the game type affect playing as a team and moving the ball around?
* Is there a difference in fighting for extra possessions to give each team a better shot at winning?
* Being a huge fan of basketball and watching the sport from a very young age, I would predict that teams generally perform differently in the playoffs versus the regular season. One difference I’ve noticed between the two game types is that the game definitely gets more physical in the playoffs. I’ve also seen that teams typically commit much more careless turnovers in regular season games than they do in the playoffs. Overall, once teams realize that they actually have a chance of chasing a championship by earning themselves a spot in the playoffs, they mentally and physically lock in which can be seen by differences in their game.

**Data Collection**

I wanted to find a complete, official NBA dataset. After some research and observations of various datasets on different sites including Kaggle, the most complete one was found on Kaggle by WYATT WALSH:

Source 1

<https://www.kaggle.com/datasets/wyattowalsh/basketball>

* Includes games, teams, and players data dating all the way back to the first NBA season in 1946
* Sourced from nba’s official stats website via nba\_api
* Data stored in form of SQLite database within zip file. So, to read the file on Mac, I had to do some research. Came across *DB Browser for SQLite* database manager tool which works with such databases and downloaded the program. Observed data upon opening and confirmed records, tables, columns, etc. with dataset details on its Kaggle page
  + - Total Tables: 16
    - Largest Table: *Game, Game\_Inactive\_Players, Game\_Officials*
    - Latest data: 2021 NBA regular season
* As complete as this dataset was, I quickly noticed it didn’t contain game data on ANY NBA playoffs. Playoffs typically start mid-end April and extend into June or July, but such data was not available. So, I had to research datasets that contained playoffs game data. Found dataset on Kaggle once again by NATHAN LAUGA:

Source 2

<https://www.kaggle.com/datasets/nathanlauga/nba-games>

* Includes all games data (preseason, regular season, & playoffs) from 2003 to 2022
* Also sourced from nba’s official stats website via nba\_api
* Stored in the form of 5 CSV files
  + Total Tables: 5
  + Records By Table: games\_details (645953 records)

games (25796 records)

**MySQL**

**Importing Data into MySQL**

Source 1

DB Browser provides several options to export the data: 1. Export entire database 2. Export only tables into CSV or JSON files. Since this data will go into a new database in MySQL, I found it convenient to export the entire database to a SQL file which will then be imported into the server. Having looked through the data, for the purposes of the project, I realized I don’t need all of it. The file is actually quite large so I chose to exclude those tables that were irrelevant for my analysis (although still extremely useful for future projects & insights).

8 out of 16 Tables used

Before exporting, I chose to not keep column names in INSERT INTO statements from the DB Browser export window. Also, I chose to overwrite any old schemas if they already existed in the database.

Once the export was completed, I opened & observed the file in Workbench. Some things I noticed were double quotation marks around every table name and column name. Since this syntax is not used in MySQL, I used the Replace tool in Workbench to Replace every instance of these with a blank. However, the app kept crashing whenever I tried this in workbench because the file was too big to search through. So, I opened the file in Text Editor instead and did the same exact procedure (took ~1s to replace 471K instances). I then saved the file from within Text Editor which automatically updated the file in the original directory.

Upon opening the updated version in MySQL, I had to change the data types of many fields before executing the entire file so that when I’m working with the data later (as in aggregations or other functions), I won’t come across as many errors.

1. Added check constraint for couple fields in *player\_attributes* table to ensure only ‘Y’ or ‘N’ values
2. Added check constraint for field in game table to ensure only ‘W’ or ‘L’ values
3. Game\_date\_day

Another thing I noticed in the *Game* table was that many fields were defined as REAL data types even

though the only digit after the decimal points was 0 for all records in such columns. Although such data (FGM, FGA) is usually written as whole numbers, I noticed that inserting values like 34.0, 54.0, 78.0 and defining the columns as REAL works just fine because MySQL will just leave off the trailing 0 upon import.

Each tables INSERT statements were enclosed in a **transaction** so that I could work on any errors/warnings upon execution without committing any data to the server until all issues were taken care of.

* Had to fix issue where empty fields were being passed to INT defined columns. Explicitly replaced empty fields with NULLs using Text Edit. Although TEXT fields spark no such errors for empty fields, replaced any empty TEXT cells with NULLs as well.
* Jerome Anderson had *Jersey* numbers 21-23 for INT defined field, changed to 21
* Jesse Arnelle had *Jersey* numbers 15-18 for INT defined field, changed to 18

. “ “

. “ “

. 31 total records had similar issues for *Jersey* field values

* Couple of records in *Jersey* like this: 11 15
* Fixed *Jersey* for Hamed Haddadi in Game\_Inactive\_players to 98

After taking care of all errors, all data was properly inserted into each table (checked several records for

data completeness & accuracy, also verified # of records for each table against data details page on Kaggle).



* Ended the transaction with **COMMIT** to permanently add all data to the server.

Source 2

Out of the 5 csv files, I only needed the two that pertained to games data (*games* & *games\_details*). Since they’re both csv files, the best method to import them into MySQL was to use LOAD DATA LOCAL INFILE command. First, I made copies of both original data files onto new workbooks in case I need to edit or make any changes (suffixed ED). First, I needed to do the following before implementing the command.

1. Create tables into which data from input files will be imported
   1. Defined data types for every column
      1. In *Games\_ALL* table, used BOOLEAN for *HOME\_TEAM\_WINS* because values either 0 or 1
      2. In Game\_Player\_Stats table, added check constraint for *START\_POSITION* (for empty cells in input file, replace with ‘Bench’; will need to perform preprocessing transformations)

-TIME for *MIN\_played*

Games\_ALL Table

1. Perform preprocessing transformation on GAME\_DATE\_EST input field in gamesED input file using STR\_TO\_DATE in order to obtain proper date format in MySQL

1st Attempt:

.220s ; Records: 25796 Deleted: 0 Skipped: 0 Warnings: 1188

* + Warnings were ‘incorrect integer values’ & ‘data truncated’ for empty cells for both INT and DOUBLE defined fields. Had to go back and include more preprocessing transformations in my LOAD DATA command to account for INT and DOUBLE fields that had empty strings in input file. Used IF logic to replace all empty cells with NULLs, else keep original value for these fields (NULLIF). (Dropped previous table before running command again)

2nd Attempt: success

.315s ; Records: 25796 Deleted: 0 Skipped: 0 Warnings: 0

Games\_Player\_Stats Table (check back on FIRST\_NAME & MIN\_played for logic/completeness)

1. Perform preprocessing transformation on START\_POSITION field. In input file, if the cells were empty, that means the players did not start the game in the starting lineup and were coming off the bench. So, in order to prevent errors for this ENUM-defined field, I used IF logic once again to replace all instances of empty input cells with ‘Bench’ (CASE statement)
2. MIN\_played input field didn’t have a consistent format. From observation, format for all players who played 24MIN or more was MM:SS:(1/100s) and format for all players who played less than 24MIN was MM:SS. So, I first converted the format of this field to be consistent for every cell and easy for import into MySQL’s TIME data type (in Excel, under Format Cells > More Number Formats > Time). Then, performed preprocessing transformation in MySQL using STR\_TO\_DATE and time format components

1st Attempt

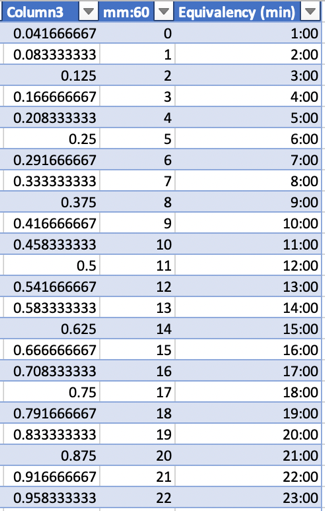
5.828s ; Records: 645953 Deleted: 0 Skipped: 0 Warnings: 2032413

* + ‘incorrect integer values’ & ‘data truncated’ warnings/messages for all empty input cells which were being mapped to INT & DOUBLE defined fields. Similar to Games\_ALL table, used NULLIF logic to replace all empty input cells with NULLs upon mapping. Noticed this was mostly occurring for all players who did not play (DNP) because no data was available for them. (Had to drop/create table before running command again)

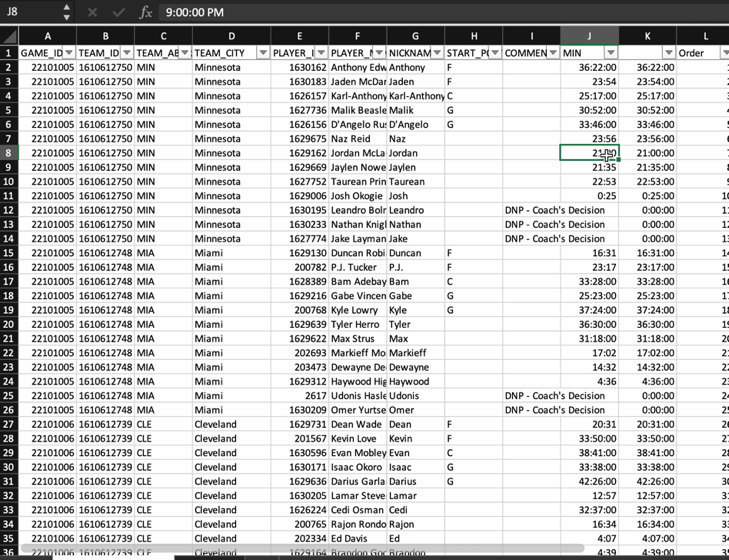
2nd Attempt:

6.894s ; Records: 645953 Deleted: 0 Skipped: 0 Warnings: 131117

* + ‘incorrect integer value’ warnings for every single empty input cell in last column only. This was because this particular file was probably created on a Windows system where two characters (\r\n) are usually used as line terminators. So, I edited the LINES TERMINATED section accordingly
  + Other errors pertained to the time field (MIN\_played). I noticed here that Step 4 actually changed the original values quite a lot which is why I was getting so many ‘incorrect datetime value’ errors upon loading. So, I had to reevaluate/compare the OG/ED sources and made appropriate changes this time around
    - In the input file, there were some cells that were left aligned (in format mm:ss) while all others were right aligned. So, I first right aligned all cells.
    - 725 cells had format mm:60s; in MySQL, highest limit for seconds time component is 59. After confirming from researching the game data for 3 random players on nba.com, these cells meant adding 1min to the players minutes played. So, in Excel, I used following formula in adjacent column: LEFT function to extract the mm and add 1 to them, then concatenated to include 00s. Then, replaced original values with these new values (went through A LOT of trouble because copy/pasting within filtered columns is a major inconvenience in Excel and not a direct feature)[ <https://techcommunity.microsoft.com/t5/excel/paste-to-visible-cells-only-in-a-filtered-cells-only/m-p/1013991>]. To maintain original order of table, created new column which numbered every row and sorted by that column ASC
    - 656 cells had decimal values, which doesn’t make sense for time. After digging a little deeper and researching the players official stats on google once again, I noticed that values like 0.041666667 actually represented 0:60 (1min), values like 0.083333333 represented 1:60 (2min), and so on. So, I created an equivalency chart to make converting these values easy. Used “Fill Left” to replace old values with new ones. Or, I could’ve just converted the column type to TIME



* + - 12 cells had negative values. 9 of those were all coming from one specific game (GAME\_ID= 10800045). The other 3 were all from a single game as well (GAME\_ID= 10800035). I figured it might be a reporting issue, so for now I just replaced all these values with 0:00:00. What I could do later on is find the avg. time each of these players played in that season and replace the default 0’s with that time
    - Converted all 0’s to 0:00:00
    - 94 0:00’s to 0:00:00
    - 91 1:00’s to 1:00:00
    - In order to standardize all other times that were in format mm:ss or m:ss, in an adjacent column, I first found the length of such times. Then, I used an IF function to capture only these time formats based on their lengths and append them with :00. Else, keep original time format. The IF function caused many other changes since it was applied to the entire dataset: There were many cells that only contained integer values (23,606) which also got changed to their TIME equivalents, but not the right ones. After confirming with game data on nba.com, I decided to correct them in this step. Some reported times were 96, 86, 78min which is odd if the game doesn’t extend into several OT’s. Other games were reporting 5 min for every player on both teams. Further research into the data unveiled many games must’ve had some reporting issues, and many seemed to be in Oct. To prevent errors upon loading, I replaced all times above 59 with 0 but made note (12 total). The rest, I sorted >=1 and appended to match the standardized format using CONCAT; 725 cells had format mm:00 in General which were converted to the standardized time using CONCAT; the IF function added 0min to all players who did not play in any games, where previously the cells were empty which I’d rather keep. Sorted data based on game comments and deleted all cells; Lastly, to convert the entire original column to the new values in this adjacent formula column, I created another adjacent column to paste only the values & number formatting of the editing column, then used “Fill Left” to replace (Had to drop/create table before running command again)
      * + See below for what this looked like



3rd Attempt (workbench kept crashing, so ran from terminal):

6.445s ; Records: 645953 Deleted: 0 Skipped: 0 Warnings: 449

* Most of the warnings were messages stating ‘incorrect datetime value’ where the value was some string ending of adjacent column to the left (ex. left eye, right foot, right ankle, etc.). After looking at the log file and finding some of the error cells in the input file, I noticed a pattern where all cells had commas in them. I instantly realized that the comma was ending the field (because that’s the delimiter which defines the FIELDS TERMINATED line) early and anything that followed the comma was passed over to the next MIN\_played field. So, I replaced all instances of commas with semicolons in Excel (441).
* A few of the warnings pertained to 'incorrect datetime value’ where the mm were >= 60. Obviously, 59 is the limit so I had to go back to the input file and see what was happening (8 cells had such values). Just to confirm that such data actually makes sense, I googled the official games stats for a couple of the players and found that the data is indeed valid because the numbers are coming from games that extended into 3OT, and star players usually play close to the full game. Since it was only 8 time values, I decided to just UPDATE the table once its created for these games & players instead of trying to capture them in preprocessing which would have been a little more difficult.
* Also noticed I had to add NULLIF condition for MIN\_played field

4th Attempt: success

6.843s ; Records: 645953 Deleted: 0 Skipped: 0 Warnings: 8

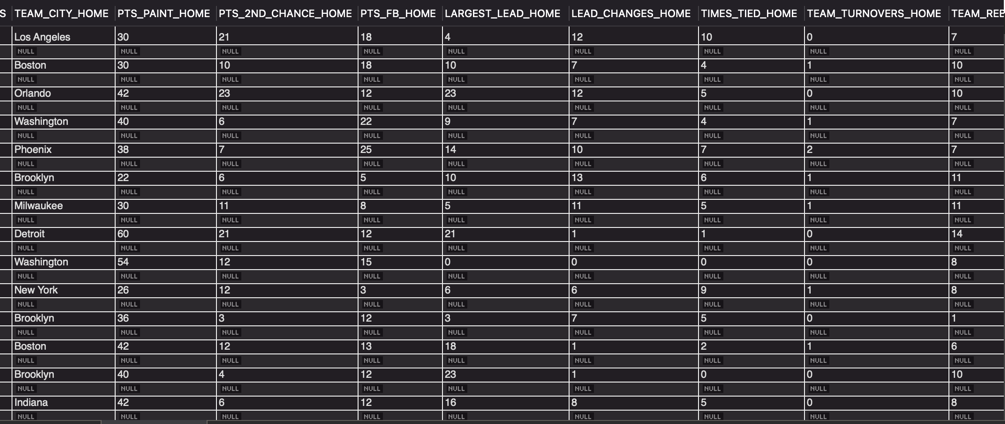
1. UPDATE 8 NULL MIN\_played records where input file MIN >= 60:00:00. Performed within a transaction in order to safely update column for only records that satisfied this condition. Isolated these records by identifying what fields made the table unique and used those in the WHERE clause of the update. Had to turn sql\_safe\_mode OFF because keys of table weren’t defined yet. Checked to see proper updates, then COMMITTED to server.
2. UPDATED FIRST\_NAME (same as NICKNAME field in input file) field for all records that were empty. Again, did so in another transaction to ensure proper execution before commitment. First, LOCATE position of first space in player name, then nest into LEFT clause to extract first name. Found that records successfully updated, EXCEPT for all instances of Nene. He doesn’t have a last name, so no spaces were found in his name and the cells remained empty. Used second UPDATE statement to update only his records. COMMITTED to server after checking proper update of records.
3. Added a LAST\_NAME column using ALTER TABLE statement for purposes of normalization. Similar to FIRST\_NAME update, did so within a transaction. Again, had to use second UPDATE clause so that LAST\_NAME field was empty for all instances of Nene because he doesn’t have a last name.

**Database Design**

* Assign keys to all tables, drop duplicates, separate compounds, and define relationships in order to work easily with data later on

Table: Game

1. Duplicate columns for team ID and team city. Before dropping the team city duplicates (the ones appearing later in the table), I had to transfer records from these duplicate fields which were complete into the city columns earlier in the table which had NULLs (used UPDATE clauses within transaction; turned OFF sql safe updates). COMMITTED after verifying proper changes. Then, DROPPED duplicate columns using ALTER TABLE
2. Duplicate column total TO’s for both home & away teams was DROPPED.
3. Added PK CONSTRAINT to GAME\_ID field (using ALTER TABLE) which uniquely defines every row in the table
   * + Received duplicate entry error for GAME\_ID, so DELETED entry and ran again
     + Received another duplicate entry error for GAME\_ID. But this time, since I got another error, I selected only this KEY field and GROUPED by it with a COUNT function to find which GAME\_ID’s had more than 1 entry. Found 68 games had duplicate entries. Used subquery to extract only these GAME\_ID’s, then another subquery to view the full data of each game along with its duplicate entry (136 records). The only difference between each GAME\_ID and its duplicate was that the duplicate had NULLs for all fields later in the table (see below).



* + - So, along with the GAME\_ID’s, the NULL presence for one field was used in the WHERE condition to delete the duplicates (DELETE clause written in a TRANSACTION). Verified that the duplicates properly got deleted using the same query to first find the duplicates [Resulting records: 62379, 62447(w/ dups) – 68(dups)]

1. Run Step 3 again
2. Added FK CONSTRAINT to TEAM\_ID\_HOME which references Team
3. Added FK CONSTRAINT to TEAM\_ID\_AWAY which references Team

Table: Game\_Inactive\_Players

1. Added PK CONSTRAINT to player\_ID and GAME\_ID fields
2. Added FK CONSTRAINT to GAME\_ID which references Game
3. Added FK CONSTRAINT to PLAYER\_ID which references Player
   * + Encountered an error “cannot add or update child row; a FK constraint fails.” After some observation and research, found that error occurred because this field had values which were not present in its parent field. Every inactive player should be in the Player table because it contains the complete list of all players, but every player isn’t necessarily going to be in the Game\_Inactive\_Players table. Using the NOT IN operator to compare the ID fields between these two tables, found that 117 DISTINCT PLAYER\_ID’s present in Game\_Inactive\_Players table were missing from Player table. So, I INSERTED the 117 missing PLAYER\_ID’s, along with mapping the other fields, into the Player table using a SELECT statement. Done so within a TRANSACTION to be safe and after confirming changes, COMMITED. [Resulting records: 4618, 4501(total) + 117(missing ID’s)]
     + Deleted 1 row that had NULLs for first and last name fields
4. Run Step 9 again
5. Added FK CONSTRAINT to TEAM\_ID which references Team

Table: Game\_Officials

1. Added PK CONSTRAINT to OFFICIAL\_ID and GAME\_ID fields
2. Added FK CONSTRAINT to GAME\_ID which references Game

Table: Games\_ALL

1. Added PK CONSTRAINT to GAME\_ID
   * + Received same error as in Step 3, so followed the same procedure to get rid of duplicate PK entries. 29 total duplicate entries. Unlike the Game table in Step 3, there was no unique identifier in the record/duplicate pair to use as a condition in the where clause so that only one of the identical rows is deleted (both records for each duplicate GAME\_ID were exactly identical for every field). So, I had to use a ROW\_NUMBER window function which numbered any duplicate records as 2. Then, this could be used in the where clause of the DELETE statement. But before that could be implemented, I needed to ADD a COLUMN called id so that it can be used as the row identifier when deleting the duplicates. I set the column as the PK and AUTO\_INCREMENTED values for all records. Again, used a TRANSACTION to be safe when deleting. [Resulting records: 25767, 25796(w/ dups) – 29(dups)]. Once the duplicates were successfully deleted, I DROPPED the id column as the intent was to set GAME\_ID as the PK
2. Run Step 14 again
3. DROP duplicate HOME\_TEAM\_ID & VISITOR\_TEAM\_ID fields
4. Added FK CONSTRAINT to TEAM\_ID\_home which references Team
5. Added FK CONSTRAINT to TEAM\_ID\_away which references Team

Table: Games\_Player\_Stats

1. Added PK CONSTRAINT to GAME\_ID and PLAYER\_ID fields
   * + Received same error as in Step 11, except it’s for the compound key combination. Using GROUP BY and COUNT, found 289 total duplicate entries. Followed same procedure as in Step 9 to delete all duplicate entries for the compound key. This time, I didn’t need to turn OFF sql\_safe\_updates to DELETE duplicates because I specified PK (id) values in the WHERE clause of the DELETE statement. I could’ve done the same thing in Step 9 too. Used a TRANSACTION to be safe when deleting. [Resulting records: 645664, 645953(w/ dups) – 289(dups)].
2. Added FK CONSTRAINT to GAME\_ID which references Games\_ALL
3. Run Steps 19 & 20 again
4. Added FK CONSTRAINT to TEAM\_ID which references Team

Table: Player

1. Added PK CONSTRAINT to id

Table: Player\_Attributes

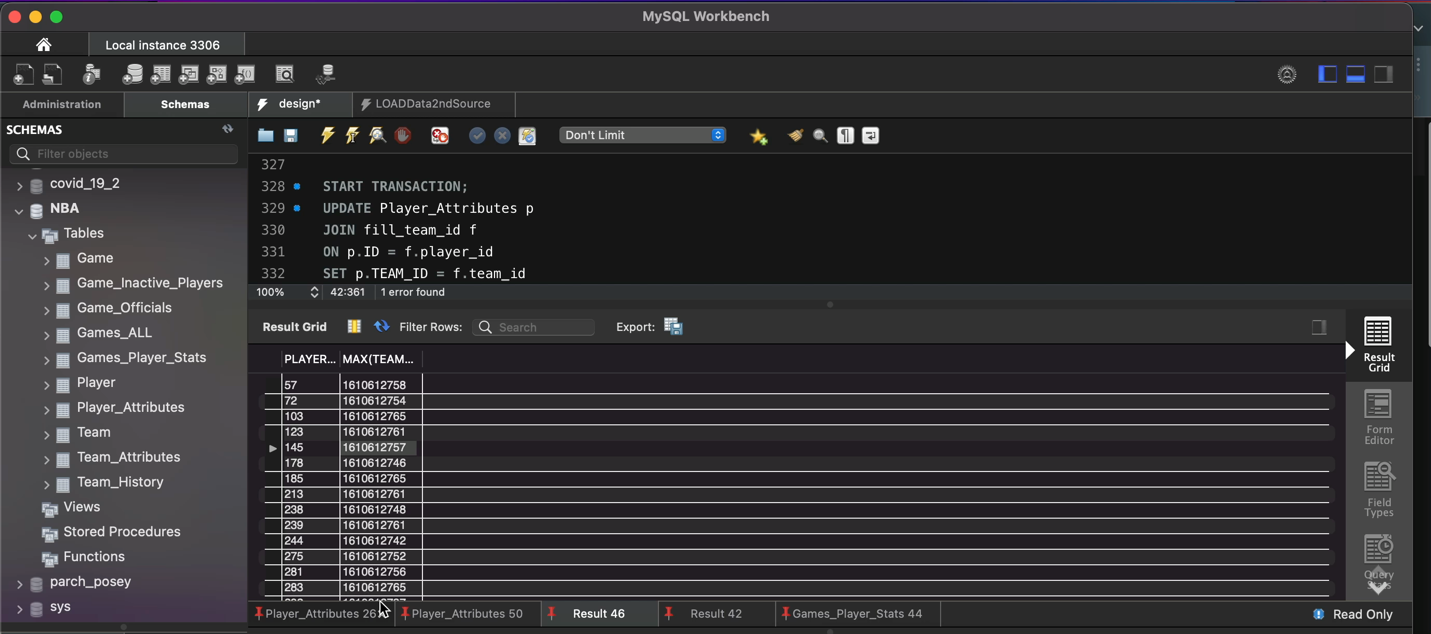
1. Added FK CONSTRAINT to ID which references Player
   * + NOTE: 118 player records from Player table are missing in this table
2. Added FK CONSTRAINT to TEAM\_ID which references Team
   * + Received same error as in Step 7: “cannot add or update child row; a FK constraint fails.” Similar to PLAYER\_ID field in Game\_Inactive\_Players, reason for error was because this TEAM\_ID field had values which were NOT IN its parent field. Following similar procedure to Step 7, used the NOT IN operator to compare this child field with its parent field and found 16 DISTINCT TEAM\_ID’s present in Player\_Attributes which were missing from Team table. Since there was only 16 missing teams from the Team table, I decided to manually add these records into the table. Used the Game table to find the team names of the 16 TEAM\_ID’s, and then searched any missing team info (state, year\_founded) on the web [<https://www.basketball-reference.com/>] which would complete the Team table for every column. Used a single INSERT statement within a TRANSACTION to add the records. Since these TEAM\_ID’s were all for historic teams, I could’ve added some of them to the Team\_History table but not all because some of the teams were actually unique by themselves and weren’t a predecessor of any newer version of a team. [Resulting records: 45, 30(total) + 15(missing ID’s)]
     + Of the 16 DISTINCT TEAM\_ID’s, one of them was 0 🡪 664 total player records in this table didn’t have an associated TEAM\_ID. The only other table I could use to fill any of these 0’s was Games\_Player\_Stats because it contained both PLAYER\_ID’s and TEAM\_ID’s. So, I first extracted all player ID’s in Player\_Attributes table which were associated with a TEAM\_ID=0 (664). Then, nested the query within a subquery so that I could view only the same player ID’s in the Games\_Player\_Stats table; it resulted in several TEAM\_ID’s for a single player ID because many players played for more than one team in their career. But this was a problem because each player ID in the Player\_Attributes table was associated with exactly one TEAM\_ID. After further research, I found that it wasn’t the most recent team the player played for, it was quite random. So, I GROUPED the aforementioned result by PLAYER\_ID and extracted the MAX TEAM\_ID so that each player ID was associated with exactly one TEAM\_ID. This way, the data was easily set up to update. After some research, I found that the best way to update one table using data from another table was to use an UPDATE JOIN statement. But, my result was stored in a subquery, so I INSERTED the player\_id,team\_id result INTO a CREATED TEMP table to easily use in the UPDATE JOIN statement. Updated the records within a TRANSACTION to be safe and after confirming some updated records, COMMITED (see below). Of the 664 TEAM\_ID’s that had 0’s, 463 got filled, leaving 201 still with 0’s. [Resulting records: 4299, 4500(total) - 201(where TEAM\_ID=0)]. For these remaining records, I just DELETED them for the time being as they were preventing from establishing the proper FK link. For future reference, exported these deleted records to an Excel file.
3. Run Step 25 again

Table: Team

1. Added PK CONSTRAINT to id

Table: Team\_Attributes

1. Added FK CONSTRAINT to ID which references Team

Table: Team\_History

1. Added FK CONSTRAINT to ID which references Team

**ERD Diagram**

* + - 1. Create EER by reverse engineering database.
      2. Then, synchronize model so that any future changes will be updated in model
      3. Save as a model file and export as a PDF to access later for reference or changes

**Querying Database**

* In order to obtain a proper analysis, I had to consider the dates for each data source.

Source 1

* + 1. earliest game date: 1946
    2. latest game date: 2021-05-16

Source 2

* + 1. earliest game date: 2003-10-05
    2. latest game date: 2022-03-12
  + based on above info, I had to find the common ground for analysis in terms of date. So, analysis will be performed for 2003-2021 nba seasons. The goal was to have a single table that contained all desired data for analysis. Since Source 1 doesn’t contain the desired NBA playoffs data, I combined the two data sources using a RIGHT JOIN condition while considering the aforementioned date situation. Upon seeing the result, I realized something very important. The Games\_ALL table from Source 2 didn’t have some key box score info for each game that I wanted for my analysis and which was in the Game table. For example, personal fouls, the number of attempts for each kind of shot, turnovers, points by quarter, etc. I could have truncated the columns in the Game table to match the same columns in the Games\_ALL table, but that would have left the majority of my questions for analysis unanswered. I wanted the detailed box score stats in the Game table to also be in the Games\_ALL table for every game. the reason for such differences stems from the differences in data collection between the two sources.
* To get the data I wanted, I went straight to the source and began working with python

Click here to see where I continued this portion of the project: [Querying Database (Cont.)](#querying_database)

**Python**

-to preface, I’ve worked with python in the past but have a very basic level understanding as I dive in.

* Since the more complete data I was hoping to obtain for the playoffs is exactly what WYATT WALSH had already done for the regular season, I went straight to his project repository on GitHub. I was looking for his code used to collect the data, which was found in a data\_collection.ipynb file under the notebooks folder. His development environment was jupyter notebooks. For myself, I already had python3 installed on my system and was looking to develop in PyCharm

IDE: PyCharm 2021.2.2 (Community Edition)

Python 3.10.0

* Before diving into code, I had to read up a lot on how API’s work with python, specifically the NBA API. Found that the API’s on nba.com are actually quite undocumented, which is why Swar Patel created the nba\_api client package hosted on GitHub to make it more accessible and to supply extensive documentation. To successfully work with the API, I followed the procedure below:

1. Download client package using pip3 in terminal (outside python shell)

- installed pip version 22.1

1. Download data analysis module *pandas* using pip3 in terminal (outside python shell)

* To pull data, endpoints needed to be used. Each API endpoint from stats.nba.com was contained in a class. Since I was concerned with playoff game stats and detailed box score summaries for those games, the two endpoints I needed to import as dependencies were:

1. Import *leaguegamelog* endpoint
2. Import *boxscoresummaryv2* endpoint

* Within each endpoint class, several different datasets and information may be contained. Upon calling each endpoint, different parameters can be passed to return specific data. Acceptable values for such parameters were documented within each endpoint’s analysis. For example, calling the *leaguegamelog* endpoint returns only one dataset which contains basic game data. So, I first wanted to obtain such info for all playoff games from 2003-2021.

1. Call *leaguegamelog* endpoint to obtain basic playoffs game data for all teams in a given nba season (pass random nba season as parameter along with ‘Playoffs’ for season type parameter). As mentioned earlier, expecting only one dataset to be returned

* Each dataset returned has additional methods in order to access the data in different forms. For example, it can be formatted in json, a dictionary, or pandas dataframe. Since PyCharm has a really nice window for viewing data in Dataframe form and its just a cleaner option overall, all data returned from API calls for this project will be in dataframe form. Upon observing the returned dataset for the random season, I noticed:
  + Every GAME\_ID was duplicated, meaning data for each game had two records: one for home team and another for away team. But, neither team record was distinguished as home or away
  + Data returned was only for a single season, but I needed such data for several seasons

-to address these concerns, I did the following:

b. I needed to create a function so that this procedure could be repeated for any nba season

passed as an argument instead of only getting data for a hard-coded season.

a. I wanted data for both home and away teams to be in a single record for each game; that is

how WYATT WALSH’s Game table was constructed as well. So, within the defined function of b., I

needed code which did that as well. I referred to WYATT WALSH’s source code on how to

separate the team records by home and away first, and then combine them into a single record

for each game. This included creating sub-functions which took the raw data returned from API

call and did the following: dropping columns that didn’t need to be distinguished as ‘HOME’ or

‘AWAY’, using concat across the vertical axis to combine home team and away team records into

single record, iterating the procedure for each GAME\_ID, and returning the combined/cleaned

dataframe for each season

1. Create list which holds all nba seasons for which basic playoffs game data will be collected (2003-2021)
2. Run the function created in 5b. within a for loop to iterate through each season in the list from step 6 and obtain a list of dataframes with basic playoffs game data for all nba seasons from 2003-2021
3. Concatenate list of dataframes from step 7 into single dataframe and hold in variable *playoffs\_df1* (1504 records, 54 columns)

* The next step was to get detailed box score data for every GAME\_ID in *playoffs\_df1*.

1. Extract all GAME\_IDs from *playoffs\_df1* into a list to pass as argument in next function (Step 10) which will obtain detailed box score summary for every game (1504 records)
2. Create a function, *get\_playoff\_box\_score\_summaries*, which will obtain detailed box score summary for every playoff game from 2003-2021
   1. Use the boxscoresummaryv2 endpoint class and make API call to return all dataframes. The class contains 9 separate datasets, but only 7 from the returned API call will be used. Once again, I referred to WYATT WALSH’s source code to obtain a similar format to what he had already created for season data in the Game table.
   2. Within the function, each returned dataframe from the API call was individually extracted. Any changes that needed to be performed for each dataframe were made, like changing column types or separating stats into home team and away team -like previously- by renaming columns (suffixed “\_HOME” and “\_AWAY”) were made. Then, each dataframe was concatenated with the next dataframe across the vertical axis to ultimately create one thorough and detailed box score summary for each playoff game. The function returned this dataframe
3. Create an empty list, *playoffs\_df2*, which will hold the detailed box score summary for every playoff game
4. Using list comprehension, fill the list from step 11 by calling the function from step 10 within a for loop for every GAME\_ID in the list from step 9
   * 1. I experienced so much trouble in this particular step. First of all, it was taking an extremely long time for the code to run (>30min). But, I figured it makes sense because of the amount of work each function call was doing for each game. And at the end of several runs, I kept receiving the following error message: “HTTPSConnectionPool(host='stats.nba.com', port=443): Read timed out. (read timeout=30)”. From reading several threads and forums of people experiencing the same issue with this NBA API [<https://github.com/swar/nba_api/issues/176>], I found that it was due to limitations on the number of API calls that can be made against stats.nba.com. several people suggested that incorporating a sleep between each API call worked for them. Essentially what this function does is delay your code execution by the number of seconds you pass as an argument. Some reported an optimal number to be .600s. So, I first imported the sleep function from the time module, and then incorporated it into my list comprehension to delay execution of each API call by .600s. It worked
5. A list of tuples was returned from execution of step 12, where the first element in the tuple was None and the second was the desired dataframe for each GAME\_ID (None, dataframe)
   1. In order to access the dataframe from each tuple and ultimately concatenate all of them into a single dataframe, I created another list comprehension and used indexing to extract the dataframe from each tuple and store it in a list which can then be concatenated across the horizontal access to create a single dataframe
6. *playoffs\_df2* (1545 records, 96 columns)from step 13 contained 1545 records when there were only 1504 total GAME\_ID’s (Step 9). Upon observing the dataframe, I found duplicate GAME\_ID’s towards the end of the table (41 duplicated records). I also noticed that the second instance of the duplicates contained NaN’s for almost all columns. So, those were the records I was looking to drop from the dataframe
7. use .drop\_duplicates dataframe method to drop all duplicate records

* now that I have the two desired dataframes, I just need to combine them into a single dataframe

1. concatenate *playoffs\_df1* and *playoffs\_df2* across the vertical axis to combine basic playoffs game data and detailed box score summary data for every playoff game from 2003-2021 into a single dataframe, *final\_playoffs\_df* (1504 records, 150 columns)
2. write *final\_playoffs\_df* to a CSV file (named playoffs\_data) within the current directory which can then be imported into MySQL and used to continue the analysis

Variable Pane:

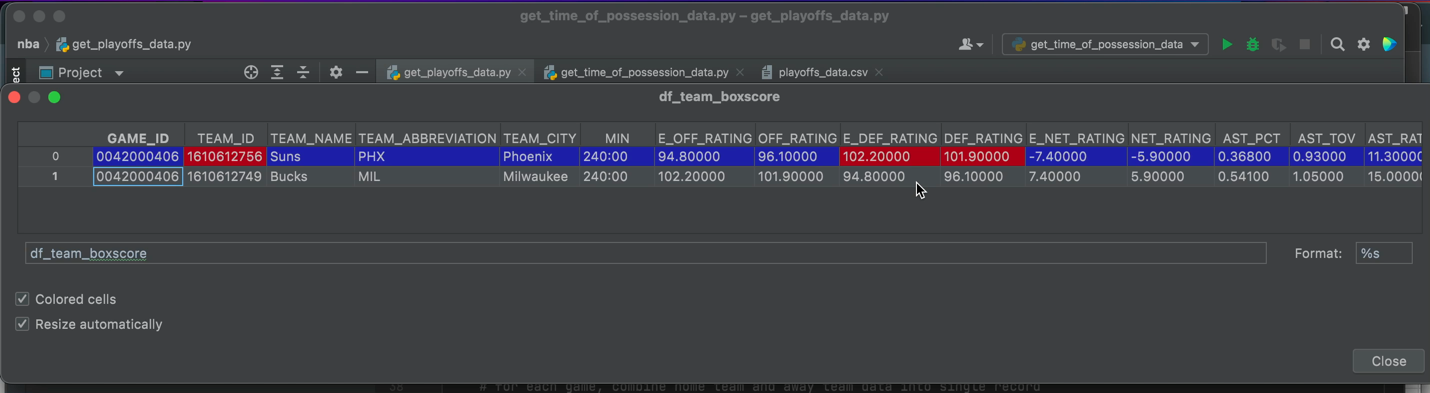


* one resource that proved to be extremely useful in how to work with this NBA API and python in general was this particular video [<https://www.youtube.com/watch?v=odCb5jczq9Y&ab_channel=LearnWithJabe>] by Learn With Jabe on YouTube. So, thanks a lot to Jabe in addition to WYATT WALSH whose code repository on GitHub outlining how he collected data from the NBA API to create many of the datasets which I’m already using in my analysis served as an awesome guide to create this playoffs game dataset for myself using Python.

**Pull Time of Possession Data from NBA API**

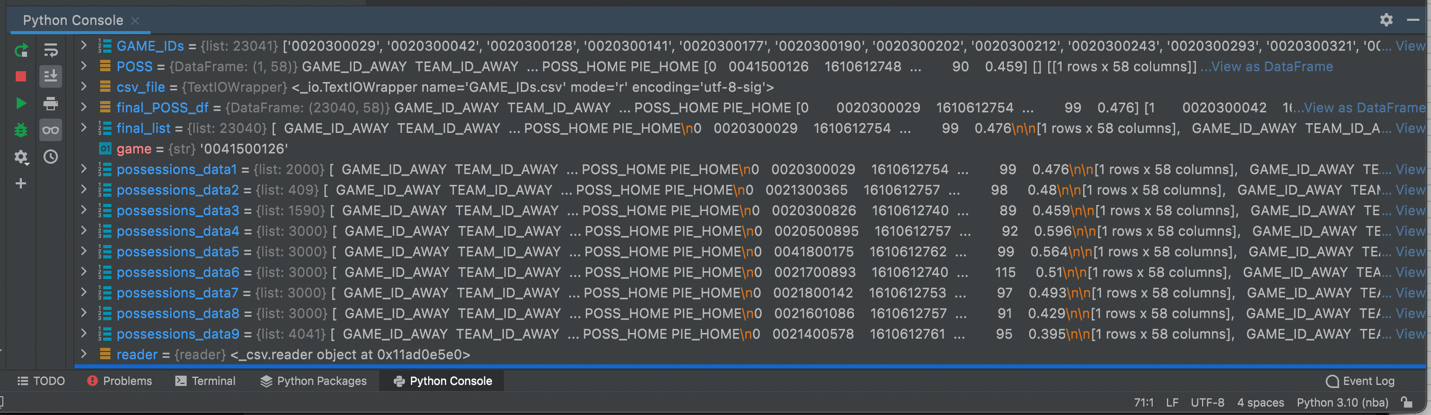
**1) Number of Possessions per Team per Game**

* the first set of data I tried to extract was the number of possessions each team has in a game for all games in my analysis from 2003-2021. This data could be directly pulled using the nba\_api client which I’ve used previously above and lies within the *boxscoreadvancedv2* endpoint

1. Create new python file called get\_number\_of\_possessions\_data
2. obtain *boxscoreadvancedv2* dependency from nba\_api package
3. call API and pass random game\_id from dataset to get a sample of what the data looks like. Return all data frames from response
   1. Based on api documentation, two datasets will be returned: one for players and one for teams. Since I’m only concerned with teams, extract only the team data frame from the response.
   2. For each GAME\_ID that is passed, two records are returned as shown above: one for each team in the matchup. I want to combine the records into a single row where the individual records are distinguished by “\_HOME” team or “\_AWAY” team.
      1. I first had to extract the individual rows using .loc[[]].
      2. Then, using the .join() method for dataframes, I joined the two separate rows from previous step side-by-side and passed lsuffix and rsuffix arguments in method to denote the home team and away team in each row

1. Now that I’ve obtained the data for one game, create a function so that the procedure can be repeated for all games in my dataset.
   1. First, I had to obtain a list which will hold all GAME\_IDs from my dataset and can be passed as an argument to the function. Since I already have this info in my database, I queried this column only from the *season\_and\_playoffs\_data* table and exported it to an excel file (23041 GAME\_IDs). Raw GAME\_IDs that the NBA stats API uses are in the form 00’8-digits’. Since both Excel and MySQL ignore the leading 0’s (a rule of Excel and MySQL), my data was only in the form of the 8-digits. So, I had to add back the leading 0’s to all GAME\_IDs for the API to read them properly and return the correct data
      1. to convert all GAME\_IDs, in an adjacent column for the first GAME\_ID, I converted the cell format to a custom one (0000000000) which holds a total of 10 digits. if any number less than 10 digits is added to the cell, the remaining digits will be leading 0’s. using a formula, I inputted the GAME\_ID’s 8-digits into the cell, and copied the formula to the end of the list. Then replaced original GAME\_IDs column with this new column by copying and pasting as “Values & Number Formatting”.
      2. Next, I needed to read the csv file into a list. So, I imported the csv module and also copied the csv file into the same directory as the project folder so that it can be read by python.
   2. Since I’ve already obtained a sample of the data for one game, I used the same statements in this created function (*get\_possessions\_data*) which will then be used to repeat the procedure for every GAME\_ID that is passed once called upon
2. Create an empty list, *possessions\_data,* which will hold data for every game
3. Use for loop to pass every game from GAME\_IDs into the function in step 3 and populate the list in step 4 with the returned dataframe for each game. Include sleep function to delay each call to the API so that timeout errors are avoided
   1. After running the loop and iterating through the entire GAME\_IDs list, I encountered too many errors and assumed it may be because of the numerous calls to the API. So, I decided to break up retrieving the data into separate sections; instead of running the function once, I ran it ~10 different times (~3000 GAME\_IDs for each run) and populated the result of each run into different lists. Each subsequent run picked up from where the preceding run left off. To break down the GAME\_IDs into subsections, I iterated through a range of the entire list in the for loop for each function execution.

* + 1. One particular index (2409) was giving me a lot of trouble, and I couldn’t identify why. So, I skipped it for now and will revisit later
* For some reason, the GAME\_ID for this particular index was returning no data from the API call and I kept receiving a “IndexError: list index out of range” error when its actually not out of range because all preceding and subsequent GAME\_IDs worked. So, I just left this datapoint out

1. Combined individual lists into single list of all dataframes
2. **Combined list of all dataframes into single dataframe using pd.concat across the horizontal axis
3. Write Step 7 dataframe to csv file called *number\_of\_possessions\_data*

2) **Time of Possession (TOP) per Team per Game**

* The second set of data I needed to extract was the total time of possession for each team per game. I found the endpoint corresponding to this stat in the NBA API client, but this particular stat was not in there. So, the alternative method of extracting the data was by scraping the NBA site where this stat lied. While on the stats site, I noticed that the NBA wasn’t reporting this stat for each game, but rather displays the stat as an average of all the games that a team has played for a particular season type (typically how NBA reports most stats like PPG, APG). This is an important fact to note because the possessions data I just extracted in step 1 is on a per game basis, so I will have to perform some slightly different calculations in order to get the final desired stat for every game.

1. Created new python file called *get\_time\_of\_possession\_data*
2. Import requests module to work with NBA server
3. Import pandas
4. Navigate to webpage which contains stat of interest [<https://www.nba.com/stats/teams/touches/?Season=2021-22&SeasonType=Regular%20Season>]
   1. Open up developer tools in order to find API endpoint which is displaying the stats on the page (endpoint name: leaguedashptstats)
   2. Obtain Request URL from Headers tab of endpoint in developer toolbar

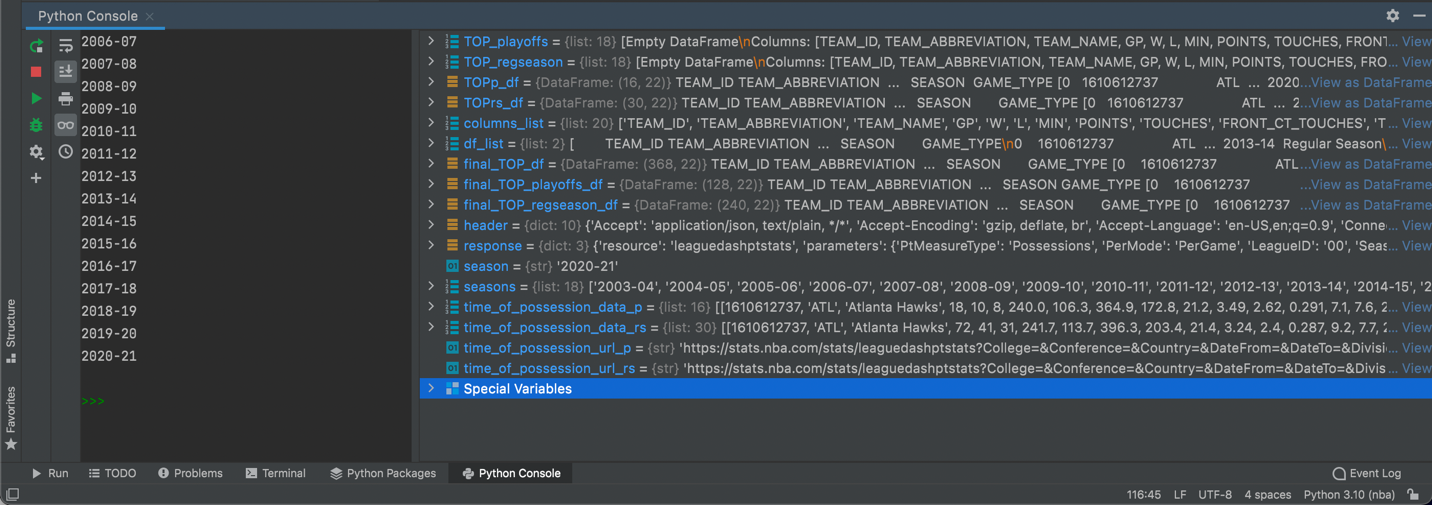
* As a test, I will obtain the desired data of all teams for the current regular season (2021-22)

1. In python script, Define url from step 4b
2. Create custom request header in order to successfully obtain response from server
3. Obtain response from server in json form
4. Use indexing to access desired data from returned json
5. Define column names for dataframe which will hold data
6. Create dataframe using data from step 8 and column labels from step 9
7. Add two columns to dataframe from step 10 to identify season year and season type

* Since I was able to successfully obtain the data for a single regular season, now I just need to repeat the procedure for all regular season and playoff game types for all years in my analysis (2003-2021). I would need to implement two separate for loops:

1. Iterate through every season for regular season games and repeat the procedures from steps 5-11. Within the loop, append each season’s data into an empty list. Then, concatenate the list of dataframes into a single dataframe called *final\_TOP\_regseason\_df*

2. Iterate through every season for playoffs games and repeat the procedures from steps 5-11. Within the loop, append each season’s data into an empty list. Then, concatenate the list of dataframes into a single dataframe called *final\_TOP\_playoffs\_*df (NOTE: I had to go back and edit the season type parameter in the original url from step 5 so that playoffs data was returned in the response)

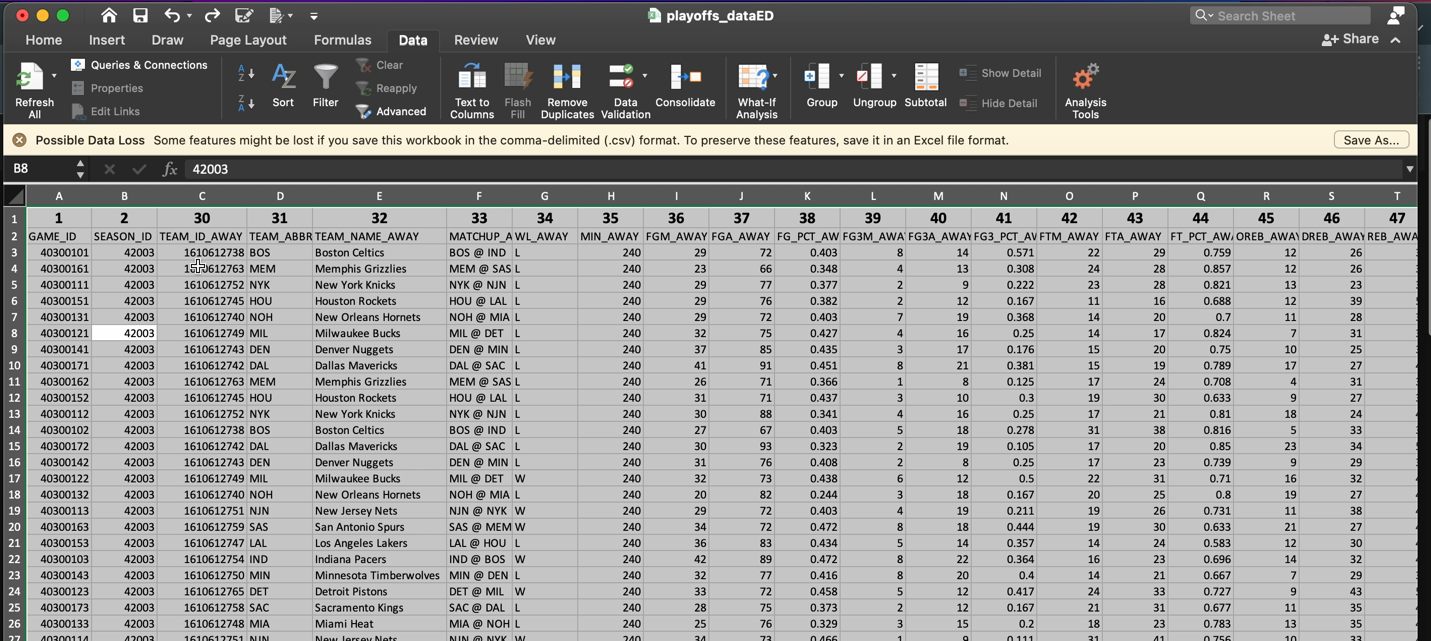
* + In obtaining the data, i noticed a limitation because I was receiving some empty dataframes. The earliest year for which I can obtain data through this web scraping method is 2013-14. That will result in missing data for 10 out of the total 18 seasons in my analysis for this particular stat. however, since I really want to observe changes in this stat, I will work with what I have.

1. Place *final\_TOP\_regseason\_df* and *final\_TOP\_playoffs\_*df dataframes into a list (df\_list) so that they can be combined to create a single dataframe called *final\_TOP\_df*
2. Retrieve columns of interest from dataframe in step 12
3. Write final dataframe to csv file called *time\_of\_possessions\_data*

* to help guide me in obtaining this second set of data from the NBA API using web scraping, I once again referred to Learn With Jabe’s tutorial on YouTube [<https://www.youtube.com/watch?v=IELK56jIsEo>]. So, many thanks to Jabe for the indirect help in obtaining the data I was looking for.

**MySQL**

**Working with *playoffs\_data***

* one of the first things I did after creating the playoffs dataset was ensuring the integrity and completeness of the data. So, I picked ~5 random records from the CSV file and fact checked them against data on ESPN.com, basketballreference.com, and stats.nba.com; found that the data looked good.
* Before the playoffs\_data CSV file could be loaded into MySQL, I recognized some transformations I needed to perform using Excel in order to avoid some of the errors or problems that may appear during loading
  + The GAME\_DATE\_EST column denoting the date of every playoff game was in the format: “2004-04-17T00:00:00”. I inserted an adjacent column and used Excel’s LEFT function to extract only the date portion as the trailing 0’s denoting time is not needed. Since a formula was used, I then copied and pasted the values only into the original column. The placeholder column was then deleted
    1. Similar procedure was followed for the only other date column in the dataset, LAST\_GAME\_DATE\_EST
  + To make things easier later on, I wanted to rearrange the order of the columns to match the Game table which I’ve already imported into MySQL because the fields for both datasets are essentially the same; only difference is the Game table holds season data and this new dataset holds playoffs data. To do so, I inserted a new row at the very top of the table, numbered each column in the order I wanted it to appear from left to right by referencing the Game table, and then used Data Sort to sort by this numbered row. Once sorted, deleted inserted row
  + For the GAME\_DATE\_DAY field, I replaced all commas with spaces because I remember encountering a problem with them when using the LOAD DATA command in MySQL

**MySQL**

**Import *playoffs\_data***

* To import using the LOAD DATA command, I first created a table (playoffs\_data) into which the data will be loaded and defined its columns
  + 1. Since the format of this dataset is the exact same as the Game table which has already been loaded into the NBA database, I was able to copy and paste the table definition from that table for this new table.

1st Attempt:

.146s ; Records: 1504 Warnings: 13218

* + 1. Similar to LOAD DATA commands previously done, every input field that contained empty cells raised a “truncated” or “incorrect integer value” warning. So, I set preprocessing transformations for these fields and replaced empty cells with NULLs using NULLIF, else keep original value
    2. Used STR\_TO\_DATE to transform the only two input file date columns
    3. The last column raised a “truncated” warning for every row. Faced similar issue before, so changed LINES TERMINATED clause from \n to \r\n.

Dropped existing table before running again

2nd Attempt:

.130s ; Records: 1504 Warnings: 195

* + 1. TEAM\_REBOUNDS\_HOME, TEAM\_REBOUNDS\_AWAY, PTS\_OFF\_TO\_HOME, PTS\_OFF\_TO\_AWAY, and ATTENDANCE fields had some empty input cells. Used NULLIF to replace them with NULLs, else keep original values

Dropped existing table before running again

3rd Attempt: success

.099s ; Records: 1504 Deleted: 0 Skipped: 0 Warnings: 0



* Before moving on, checked some records against the original input file to ensure data was properly loaded into MySQL database. Everything looked good.
* Perform some **data normalization** and edit **database design** to account for this new table. Will refer to the Game table since table structure is the same
  + Team ID and Team City have duplicate columns for both home and away teams. Before dropping these duplicates, used the != (not equal to) operator to compare every record between the 1st and 2nd instance of the duplicate pairs to ensure they’re both exactly the same. DROP COLUMNS
    - 148 columns 🡪 142 columns
  + Added PK CONSTRAINT to GAME\_ID
  + Added FK CONSTRAINT to TEAM\_ID\_HOME which REFERENCES Team
  + Added FK CONSTRAINT to TEAM\_ID\_AWAY which REFERENCES Team
* Although both the Game and Playoffs\_Data tables (main tables which will be used for my analysis) have 140+ columns, I won’t need every single one of those columns. I loaded the complete tables and set their relationships because the data in its entirety can prove to be useful for any future analyses. For this purpose and to make the analysis stage more convenient for myself when querying, I extracted only the columns of interest from both tables and used them to create new tables: *season\_game\_data*, *playoffs\_game\_data*
  + INSERT values into *season\_game\_data* & *playoffs\_game\_data* using their primary tables: Game and Playoffs\_Data, respectively. For *season\_game\_data*, be sure to filter for data from 2003-2021 when inserting values. Do so within a TRANSACTION to be safe, then COMMIT to server
    - From 140+ columns 🡪 95 columns for both tables
  + Needed to set keys and relationships for these new tables. Since they were already set for their parent tables, just copied and pasted the statements for these tables.

Table: *season\_game\_data* (21537 records)

* + 1. Added PK CONSTRAINT to GAME\_ID field
    2. Added FK CONSTRAINT to TEAM\_ID\_HOME which references Team
    3. Added FK CONSTRAINT to TEAM\_ID\_AWAY which references Team

Table: *playoffs\_game\_data* (1504 records)

* + 1. Added PK CONSTRAINT to GAME\_ID
    2. Added FK CONSTRAINT to TEAM\_ID\_HOME which REFERENCES Team
    3. Added FK CONSTRAINT to TEAM\_ID\_AWAY which REFERENCES Team
* Also added new column GAME\_TYPE into both tables after Season column. This will make working with the data easier later on.
  + Used UPDATE statement to fill entire column with following values for both tables: “Regular Season” & “Playoffs”
* Now, I have both working tables from which I can query data to perform my analysis. But, based on the questions I’m trying to answer, I would have to do so separately for season data and playoffs data. To make things more convenient, I could just combine both tables into a single table using the UNION clause and obtain data for either the season or playoffs by just including the GAME\_TYPE column as a filter in the WHERE clause. And since the two tables have the exact same structure, performing this union of the two datasets can be easily done [Resulting table: 23,041 records]. This is the vision I had from the start of the project for this stage when I would start querying the database to perform my analysis. I dove into the Python portion of the project because this vision wasn’t possible with the data I had at that stage, and I knew it would be possible after obtaining the playoffs dataset from the NBA API.
  + - I could just integrate the UNIONED result of the combined datasets within a CTE to treat it as any other table within the database so that operations can be efficiently performed. However, this form would prevent me from seeing how this combined dataset relates to other tables within the database (ie. see it in the EER model). So, I created yet another table for this combined (UNIONED) dataset called *season\_and\_playoffs\_data* which would replace both the *Games\_ALL* and *Game* tables previously represented in the EER diagram.
      * Copied and pasted table definition and keys constraints from one of the two sub tables.
  + Updated EER diagram by opening previously created model and using “Synchronize Model” to only update changes for the model from database source. Also, cut *Games\_ALL* and *Game* tables from model as they’ve now been replaced by single *season\_and\_playoffs\_data* table. Saved changes

**Querying Database (Cont.)**

* Now that I have all the data of interest, I can continue querying the database to perform my analyses.
* As I was preliminarily querying the *season\_and\_playoffs\_data* table, I found there was one stat I’m particularly interested to include in my analysis which I didn’t have: the Time of each Possession for each team in every game. This stat would be interesting to compare between the regular season and playoffs because it can indicate if teams hold onto the ball longer each possession and play at a slower pace, since every possession at this point in the season bears more meaning. The first thing I did was refer to the NBA Stats Glossary to ensure I was interpreting the stat correctly [<https://www.nba.com/stats/help/glossary/>]. Then using stats.nba.com, I found that the NBA reports this stat as a total time of possession for each team per game in units of minutes, not time per possession. To get time/possession, all I need to do is calculate the following then:

However, the total number of possessions that a team has in a game was in a different stats table. I

realized if I wanted to pull this data from the NBA API, I would need to use two different endpoints and methods based on the two desired stats appearing in different locations. Since I really want to analyze this stat for my analysis, I went back to working in Python and pulling the data using the NBA API.

Click here to see how it was done: [Pull Time of Possession from NBA API](#time_of_possession)

After successfully pulling the data, I needed to derive the time of possession stat using the two datasets. I’m hoping to accomplish this in Excel using formulas.

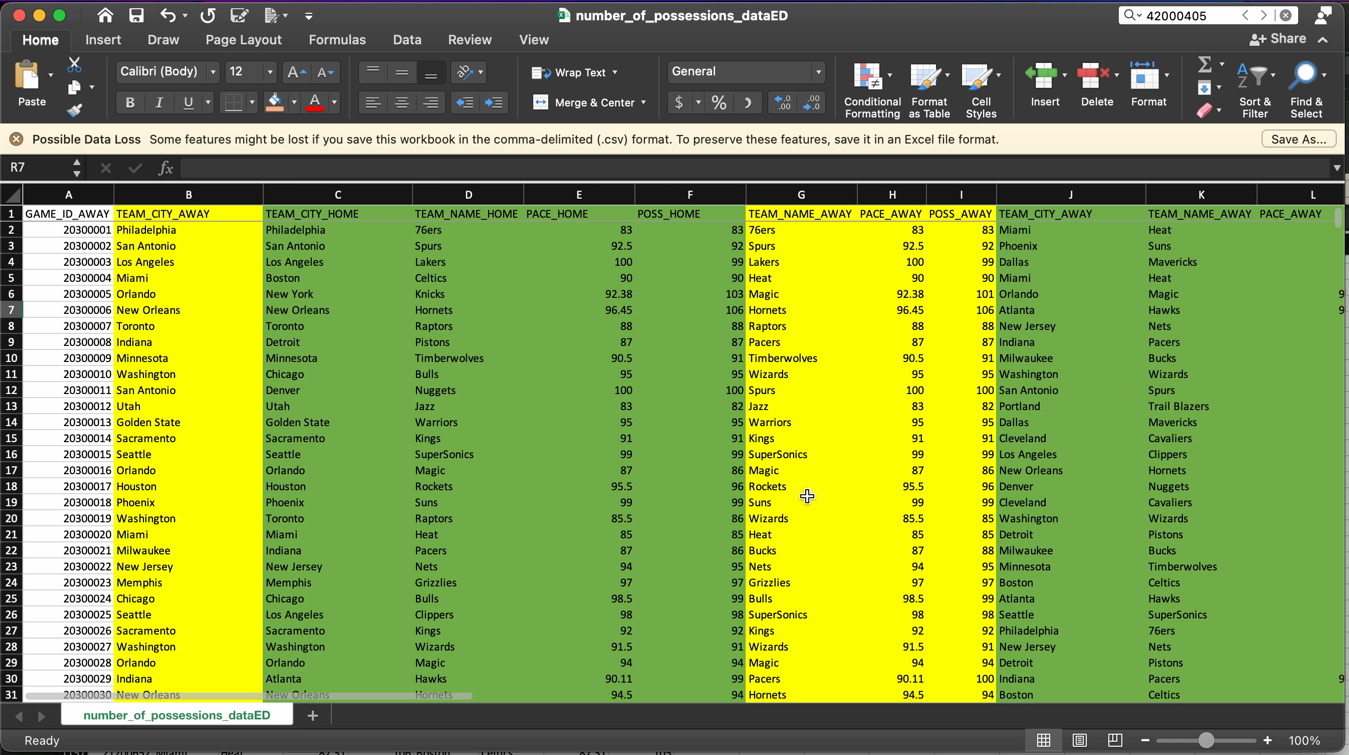
* + Upon opening the *number\_of\_possessions\_data* dataset, I noticed something off. For some games (GAME\_ID’s), the matchups for who played as the home team and who played as the away team were mixed up from the original data in the MySQL database. To fix this:

1. I exported matchup columns (who played as home and who played as away) from original data in database to an Excel file named *Reference*.
2. Then, I copied and pasted the data from step 1 to the right of the table in *number\_of\_possessions\_data* file and skipping some columns. This way, I could compare the matchups side-by-side using an IF clause which populates the home team and away team columns with the appropriate data. To align the data and make it easier to work with, I sorted the GAME\_IDs from smallest to largest in both of the datasets being compared. (Using SUMPRODUCT function, I found that 11581 home team names were mixed up)

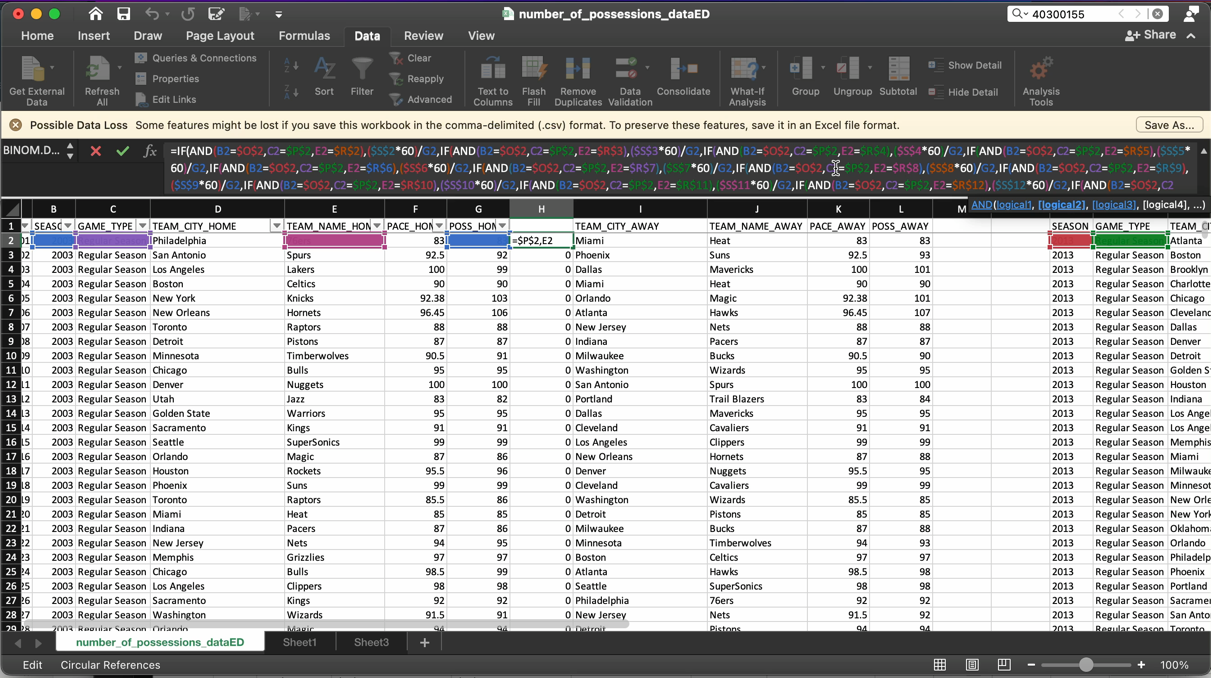
SOLUTION: See below where yellow highlighted cells represent original data in

*number\_of\_possessions\_data* file, green highlighted cells represent corrected

data, and light orange highlighted cells represent original data from database

 being compared.

* + Since the level of detail of the two tables that contain the stats to perform calculation (1) are different (*number\_of\_possessions\_data* is in per game basis, *time\_of\_possessions\_data* is in per season basis), I couldn’t do the calculation row-by-row. So, for every regular season game that a specific team played in a specific season, I divided the teams average TIME\_OF\_POSS stat for that season and game type by the number of POSS that team had for each game. To do this in Excel was a nightmare for 30 teams (see below for an example of what the formula and data looked like)



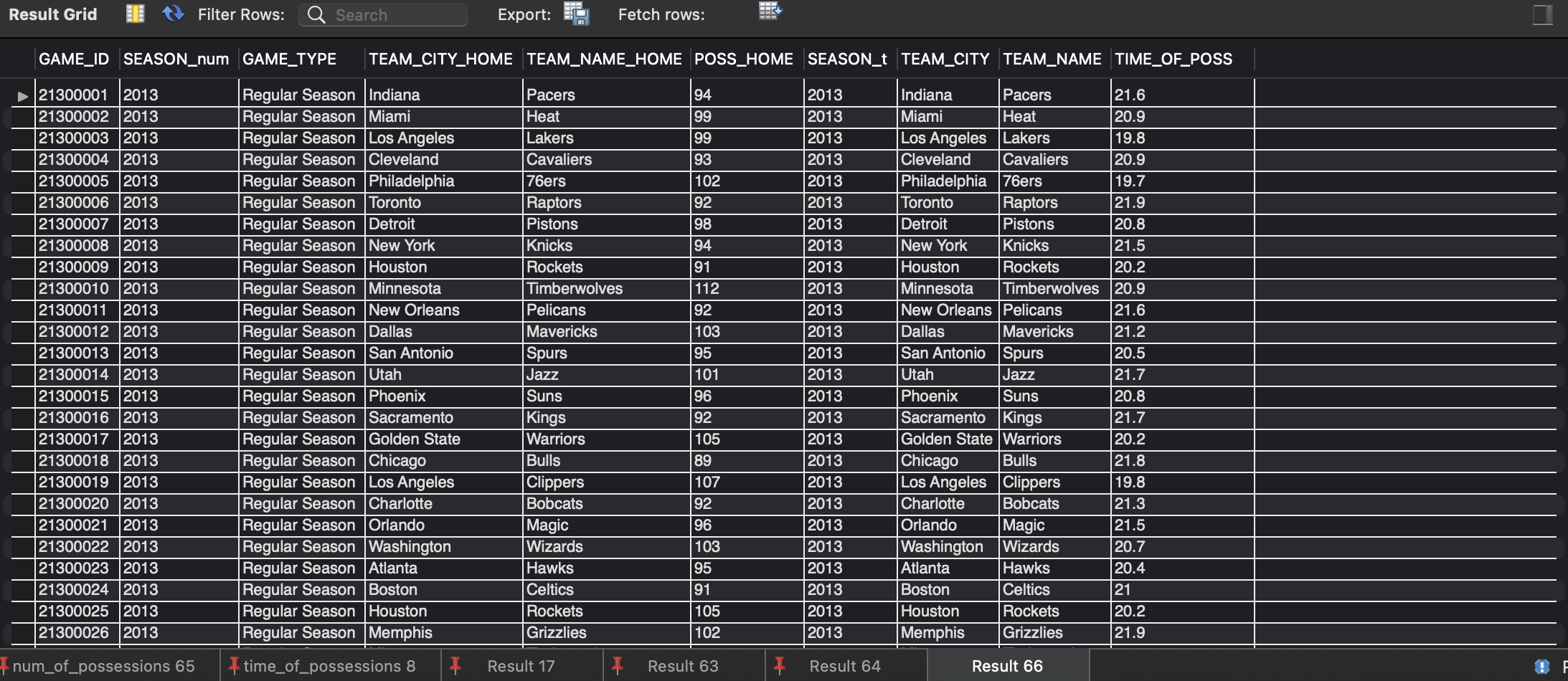
NOTE: the formula in the vid was only for the 2003 regular season. I would have to repeat the procedure

for all seasons up to 2021 and then repeat the same procedure for the playoffs. As you can imagine, not

an efficient method by any means.

* + I knew there had to be a better method of getting to the data I wanted from these two tables, so I resorted to loading each of the tables into MySQL and working with SQL to perform the calculations

**Working with MySQL to derive TOP stat for every game in analysis**

1. load both csv tables (*number\_of\_possessions\_data* and *time\_of\_possessions\_data*) into NBA database tables *num\_of\_possessions* and *time\_of\_possessions*, respectively
2. the first thing I did after loading the data was inserting values for the one index (GAME\_ID= 20300778) that didn’t return any data for number of possessions. I figured I could just find the average number of possessions of all games in the same season and game type as the missing data for each team and use those values as replacements.
   1. Just realized that this record does not even fall in the date range for this stat’s analysis, but it served as good practice regardless
3. To calculate (1) for each team in every game, I had to perform the calculations for the home teams and away teams separately. JOINING the *num\_of\_possessions* table and *time\_of\_possessions* table on similar fields will allow easy calculation of the derived stat as both of the required stats will be in the same row. Performing the calculation separately for each team was done by the following:
   1. For the HOME team, the ON clause in the JOIN contained TEAM\_NAME\_HOME (ex. below)
   2. ****For the AWAY team, the ON clause in the JOIN contained TEAM\_NAME\_AWAY
4. Nest the results from 3a and 3b in subqueries so that only the columns of interest can be extracted and formula (1) calculated for each row
5. ADD new columns named TIME\_per\_POSS\_HOME and TIME\_per\_POSS\_AWAY into *season\_and\_playoffs\_data* table which will hold the results from step 4
   1. Had to go back and update the ERD diagram to include these new columns in the *season\_and\_playoffs\_data* table
6. Store the two separate queries from step 4 into temporary tables so that they can be used in cross-table UPDATE JOIN statements to update the new added columns in step 5
7. Include *season\_and\_playoffs\_data* and temporary tables from step 6 in UPDATE JOIN statements to fill matching records in TIME\_per\_POSS\_HOME and TIME\_per\_POSS\_AWAY columns

Now that I have the time per possession data (in seconds) for each team in every game of my analysis (excluding 2003-2012 seasons), I can just repeat the procedure I’ve already done below for all other stats for this stat to determine if there is a significant difference between how teams play each possession in the regular season vs the playoffs.

**Stats Comparison Between Regular Season & Playoffs**

* One of the stats I’m interested in comparing between the regular season and playoffs for each team is the number of field goal attempts (FGA).
  + Since every record contains both the home and away team’s stats for each game, I needed to take this organization of data into consideration when querying the table. Every team plays both home and away games, so I needed to query the data separately for each of these scenarios. First, I obtained the data for the regular season.

1. The AVG of this stat was obtained for all home games and then for all away games for each team using GROUP BY. Filtered for “regular season”. Then, I joined the data for home and away games so that I could add the averages together for each team to get a total average. (NOTE: calculating the total average this way works because the data size is the same for both home and away games)

* Now, I obtain the data for the playoffs

1. Repeat the same procedure from step 1, except filter for “playoffs”

* As I was querying and observing the data, I noticed some inconsistencies. For the Los Angeles Clippers team, some records listed their city name as “LA” and team name as “LA Clippers” while other records listed city name as “Los Angeles” and team name as “Los Angeles Clippers”. After referring to the Team table, I found that the consistent city name and team name should be “Los Angeles” and “Los Angeles Clippers”, respectively. So, I UPDATED all records with names “LA” and “LA Clippers” in them (523 records) to the proper name (performed within TRANSACTION and separately for instances of home team [263] and away team [260] )

1. Run steps 1 and 2 again to account for new changes
2. Join the tables from Steps 1 and 2 to create a single table which compares the FGA stat between the regular season and playoffs for each team. Use a LEFT JOIN because not every team may have made the playoffs and for those teams, NULLs would appear in the playoff stat column



* Since I’ve built the foundation by obtaining the data for one stat, I can just copy and paste the same queries to repeat the same procedure for any stat I’d like to compare between the regular season and playoffs for each team.

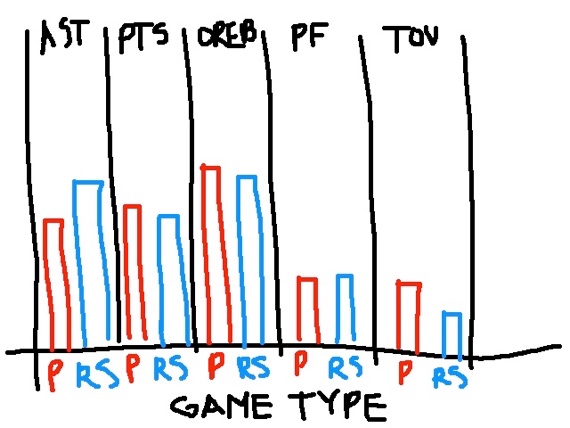
1. Repeat Steps 1, 2, and 4 for FG3A stat
2. Repeat Steps 1, 2, and 4 for AST stat
3. Repeat Steps 1, 2, and 4 for TOV stat
4. Repeat Steps 1, 2, and 4 for PF stat
5. Repeat Steps 1, 2, and 4 for PTS stat
6. Repeat Steps 1, 2, and 4 for PTS\_PAINT stat
7. Repeat Steps 1, 2, and 4 for OREB stat
8. Repeat Steps 1, 2, and 4 for TIME\_per\_POSS stat
   1. NOTE: since the data for this particular stat is not available for all games in my analysis, I needed to add an additional condition in the WHERE clause that excluded all NULL values of this stat

**Statistical Significance**

* Now that I have all the data from querying the database, I wanted to see if there is in fact significant differences of each stat between the regular season and playoffs for every team. I decided this can be done through performing statistical significance testing. Doing such analysis would be most convenient in Excel, so I exported each of the tables from Steps 4-11 into Excel (file name: Statistical Significance).
  + Remove New Orleans/Oklahoma City Hornets from each of the tables before performing the analysis because it was a temporary team which only played for about 2 seasons and never made the playoffs. Also remove the Sacramento Kings only for the TIME\_per\_POSS stat because they never made the playoffs between 2013-2021
  + The specific type of test performed was a Dependent Samples t-Test because each stat was being compared at different times of the year for the same subjects (all NBA teams).
  + I also created VIEWS for the same tables that were exported into Excel for statistical testing so that the differences in stats could be visualized in Tableau. I selected each VIEW (9 total) and exported them into separate sheets in Excel (file name: STATS)

**Tableau**

1. Connected to the STATS data source and drag/dropped each stat contained in separate sheets into the data source page window
2. Created relationships between each stat on TEAM NAME
3. I figured the best visual to use in order to pick up on some of the subtle differences between each stat was a bar graph, specifically a side-by-side bar graph to compare each stat between the regular season and playoffs for every team.
   * However, I had some trouble visualizing this in Tableau. Ultimately, the vision I had in mind for the visual was something like below for each team:



I was having trouble representing each stat category along the top and then dividing the category

into regular season R and playoffs P. I figured it had to do with the way I was representing the

data in the Excel source file. I tried several different formats of the data in Excel and would

connect to the revised Excel source to see if it worked. I found that having the data in separate

sheets was the problem. Also, instead of having two separate fields for each stat for every team

(suffixed RS and P), I needed a single stat category field which held both the regular season and

playoffs data. So, I combined all stats from the separate sheets into a single table and created

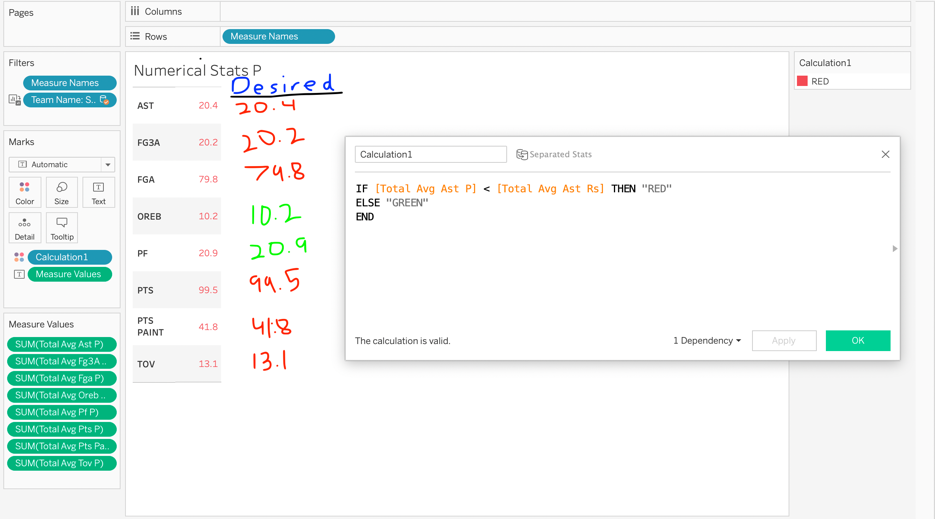
separate stat category fields for each stat. to distinguish between regular season and playoffs in

each stat field, I created a separate field called GAME\_TYPE.

1. Connected to revised STATS data source and created side-by-side bar chart.
   * 1. Confirmed accuracy of data with each VIEW in MySQL
     2. Added filter for TEAM NAME
2. One thing I wanted in my visualization was for the regular season and playoffs bars to change colors based on the teams selected. After reading several forums on Tableau Community and downloading various workbooks, I found the way to accomplish this was to create a calculated field which will essentially serve as a conditional formatting tool based on the team selected. Then, drag/drop this field into the “Color” icon [<https://community.tableau.com/s/question/0D54T00000C5bgFSAR/change-color-based-on-parameter>]



1. I thought that the side-by-side bar chart wouldn’t be good enough in a dashboard alone. As I got closer to developing the visuals in Tableau, one of my initial thoughts was that I wanted a table which compared the numerical stats between regular season and playoffs side-by-side for each team. This is typically how many stats are represented when I watch the games on TV, so I somewhat wanted to emulate that. I didn’t really know where to start, so I started searching through dashboards and visuals on Tableau Public for some inspiration of the vision I had. I came across Francis Victa’s dashboard which contained something very similar to what I wanted, so I downloaded his workbook and used it as inspiration for what I wanted to accomplish.
2. Create sheet which contains table of stat names and playoffs data
3. Create another sheet which contains table of regular season data only
   * 1. In dashboard creation, combine these two separate tables side-by-side
4. Another element I wanted to add to my final dashboard was displaying the team logos for each team that was selected from the filter. I figured this would really make my dashboard stand out along with the bars in the bar chart corresponding to each teams’ colors. I found this was rather simple to do and used the following blog as a guide [<https://www.tableau.com/about/blog/2016/2/how-use-custom-shapes-filters-your-dashboard-50200>]
5. Apply Team filter to new sheets that were created in steps 6a, 6b, 7
6. The tables created in steps 6a & 6b worked fine, but I wanted each stat to change color based on if the regular season value was less than or greater than the playoffs value for every team, and vice versa. The values less than would be colored RED and the values greater than would be colored GREEN. That way, one can easily tell how the stat compared between the two game types. However, I was having some trouble in implementing this. I knew it was a matter of conditional formatting, but implementing it in steps 6a & 6b would change the color of all stats instead of the single stat that I was comparing (see below).



I tried to find some solution to this online, but couldn’t really find anything. So, I decided to utilize Tableau Community and posted my question there hoping that someone may lead me in the right direction with an answer. Surprisingly, it didn’t take long for someone to respond with an attached example workbook,

but I couldn’t understand their suggestion for quite some time. They advised I incorporate a placeholder

for each stat and implement the IF condition for each of those placeholders separately. Eventually, I

figured it out and the results were exactly what I wanted.

1. Create dashboard using all previously created sheets
2. Edited the display for various mobile platforms