

# **COMP 3380 Final Project**

**NBA 2024–2025 Relational Database**

## **Group Members**

Dylan Beyak — UserID: 7974864

Johnny Lee — UserID: 7890682

Kameron Toews — UserID: (add id)

# Overview of project and summary of data

## Introduction

This project is about a NBA Relational Database mainly focused on the season 2024-2025. This project includes Players, Teams, Games, Draft History, Coaches of teams and their stats for regular and playoff games, Arenas that NBA teams play in and Draft Combine history. It also shows how these main components of this database share relationships with one another.

## Summary Of Data

The reason we chose the Dataset was it had the correct relations of how we wanted to model the basketball database for what we wanted and an extensive amount of attributes to choose from and how we wanted to model our database. The total amount of rows that this dataset has is 37,262 rows. The list of attributes consisted of:

- Players personal information like height weight, position, Birthdate.
- Statistics for players for each game they played like FG, 3P, BLK, etc.
- Coaches how many seasons they played and their career stats like wins, losses and total games for franchise current and overall games either in regular and in playoff.
- NBA Teams had their ID, team name team abbreviation, year founded.
- Games played in the 2024-2025 season had the teams that played against each other, the date, the arena that it was played in and the final scores.
- Draft history contained the drafts and first picks and overall picks.
- Draft combine history shows measurements of players and stats like bench press, body fat %, etc.

Loads of cleaning was required to get rid of excess columns that were not gonna be used in the database as well there were columns were not formatted in the right way an example was heights were showing up as years so there was a lot of cleaning that needed to be done. Here are all the sources that we had for our tables are the links below:

- Table for **Arena.csv** : [https://geojango.com/pages/list-of-nba-teams?srsltid=AfmBOooHCjFL0n6ZRB-rIDjEjJ8x\\_ZAeYeU2I4F2A7WTUGfL7jPp9f40](https://geojango.com/pages/list-of-nba-teams?srsltid=AfmBOooHCjFL0n6ZRB-rIDjEjJ8x_ZAeYeU2I4F2A7WTUGfL7jPp9f40)
- Coaches stats tables like **PlayoffGameCoachStats.csv**, **RegularGameCoachStats.csv** and **Coach.csv** came from : [https://www.basketball-reference.com/leagues/NBA\\_2025\\_coaches.html](https://www.basketball-reference.com/leagues/NBA_2025_coaches.html)
- Link to **PlayerInformation.csv**, **Drafts.csv**, **DraftCombine.csv**, **Player.csv**, **team.csv**, **Organization.csv**, **PlayerInformation.csv**: <https://www.kaggle.com/datasets/eduardopalmieri/nba-player-stats-season-2425?resource=download>
- For schedule table that was broken up into tables like **Games.csv**: [https://www.basketball-reference.com/leagues/NBA\\_2025\\_games.html](https://www.basketball-reference.com/leagues/NBA_2025_games.html)
- There is an additional PDF for the EER diagram since it is too large and you need to zoom.

## Discussion Of Data Model

- The reason why it was broken down into these tables from the few tables we had was because it made the modeling and the database itself more clear and concise. If we had attributes that made sense to group together and didn't rely on the other attributes we wanted to split those tables up. There is logic as well behind the thought process like teams should be separated from players and games but still share a relationship. Other examples like coaches and their stats were split up since we wanted basic information to be listed about coaches but not all their stats to follow along everytime we wanted access to just the basic information (this also applies for players and player information table). The rest of the other tables we found from the dataset were already sectioned off for us to use and clean.
- The only tricky decision around this dataset for modeling that was a huge issue is we only had data for games for season 2024-2025 so we had to work with our model only being one season
- Our model cleanly fit into the relational database since we made sure to list out all primary keys, foreign keys and all attributes in the ER model before converting it into the relational database.
- No, we do not regret the changes that we made in our model but there were decisions that we had to adjust. This was more at the later stage when it came to the queries. The first one being drafts is not enough to find what team is a player on most recently since trades happen in the NBA so we had to implement this by changing the data model and correctly implementing the change in the data. Secondly, the way we stored teamIDs in the games table we did not need to have a relationship between team and game since we can path our way through joining players.
- Yes there are a few places that the model could have been modeled differently. A couple examples of this include the following:
  - Combining the `Player` entity and the `PlayerInformation` entity, that did not need to be separate.
  - The `Coach` entity and all the coaches stats like `PlayoffGameCoachStats` and `RegularGameCoachStats` these three tables could have been just one table and be filtered in queries.

## Discussion of the database

For our project, we implemented our database using Microsoft SQL Server (MSSQL) hosted on `uranium.cs.umanitoba.ca` as instructed. We connected to `uranium` using `pymssql`. MSSQL was a bit different than using `sqlite3` like we did in class, but the differences were small, which made it easy to adjust.

The tables were created using an SQL script that was loaded into Python and executed by `pymssql`. For all text variables, we used `VARCHAR(100)` to ensure all text data would fit within the column. We also used `CHECK(LEN(attribute) > 0)` to ensure that text values were not empty strings. Our database had MSSQL keywords such as `State`, `Date`, `Length`, and `Weight` as column names in some of our tables. To tell MSSQL that an attribute name was being used literally and not as a keyword, we wrapped it in square brackets (for example, `[attribute]`).

For date columns, we added constraints to ensure the values were valid. For example, `[Date] DATE CHECK(YEAR([Date]) >= 2024)` ensured each game had a date in 2024, since our database only contained games from the 2024–2025 season. For integer columns, we validated ranges such as `RegFranchiseG INT CHECK(RegFranchiseG >= 0)` which ensured that a coach could not have a negative number of regular-season franchise games. Lastly, for columns that could only take on certain values, we used `CHECK` constraints. For example, `DraftType VARCHAR(20) CHECK (DraftType IN ('Draft', 'Territorial'))` ensured each row had a valid draft type.

Whenever a table referenced an attribute from another table, MSSQL enforced referential integrity and ensured that the referenced value existed. One issue we encountered occurred when inserting rows into the `Game` table.

Some games referenced arenas that did not exist in the `Arena` table, so we set the `Arena` value in the `Game` table to `NULL` for those rows.

We loaded the data into the database using the `dataloader.py` file. This file read the CSV files into Python using pandas to create DataFrames. Then each row was inserted using a prepared insert statement. In this file, we also implemented a helper method `_check_null` to convert pandas `NaN` values into `None` in Python, which were then converted into `NULL` values in MSSQL. Many CSV files required preprocessing, such as trimming whitespace, standardizing string values, removing invalid foreign keys, and replacing placeholder values with `NULL`.

Finally, we normalized the data in Python using pandas, including assigning IDs to tables that did not originally have them and ensuring that each table referencing foreign keys referenced the correct ones. Because of this preprocessing, we did not rely on auto-incrementing for IDs.

## Description of Interface

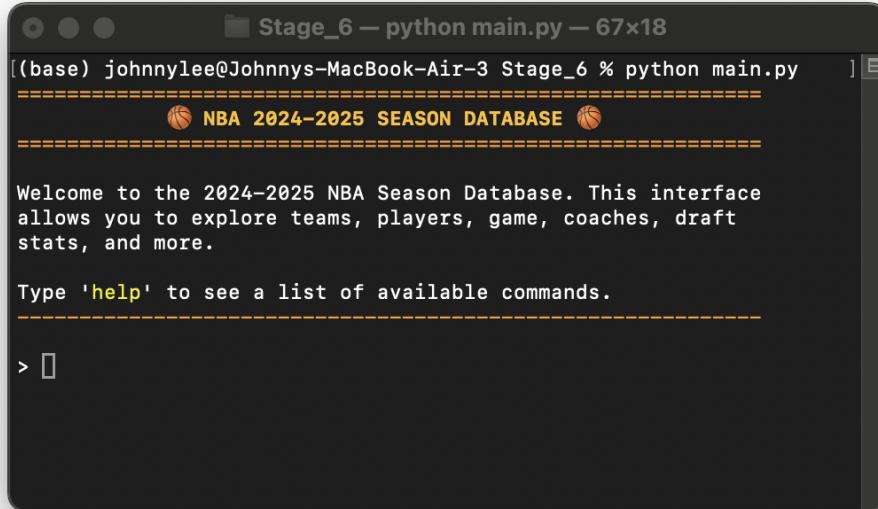
Our project uses a command-line interface (CLI) with an orange basketball-themed colour scheme, since we're working with NBA data. When the program is run, the user is greeted with a welcome message that briefly describes the database and the information available to them. The welcome message instructs the user to type `help` to see a list of available commands. All query results are printed using dynamically sized columns with orange headers that are underlined for readability.

The help menu is divided into three sections. The first section lists the simple queries, which display entire tables. These simple queries are used as reference tools for getting the input needed for complex queries. For example, if a complex query requires a GameID, the user should run the game table query to find the GameID of interest, then use that value as input for the complex query. The second section lists the complex queries. Finally, the third section lists system-level commands like clearing or repopulating the database, clearing the screen, or exiting the program. Each command in the help menu includes the command the user must enter, along with any parameters, and a short description of what the command does.

The interface is implemented in Python and is split into 3 different python files which are `interface.py`, `database_manager.py`, and `query_manager.py`. In `database_manager.py`, we created the `DatabaseManager` class, which serves as the central manager for all database operations. It manages the connection by reading the configuration file and runs SQL scripts to create tables, as well as clear the entire database. It also orchestrates data loading and querying through its `DataLoader` (defined in `dataloader.py`) and `QueryManager` (defined in `query_manager.py`) instance variables. In the `query_manager.py`, we defined the `QueryManager` class, which handles all queries sent to SQL Server, again using prepared statements to prevent SQL injection. Finally, we created an `Interface` class that handles user interaction. It displays the welcome message, help menu, and neatly formatted query results. Additionally, it parses user input and sends commands to the `DatabaseManager`.

## Diagrams of Interface

Below are 3 screenshots of the interface in action.



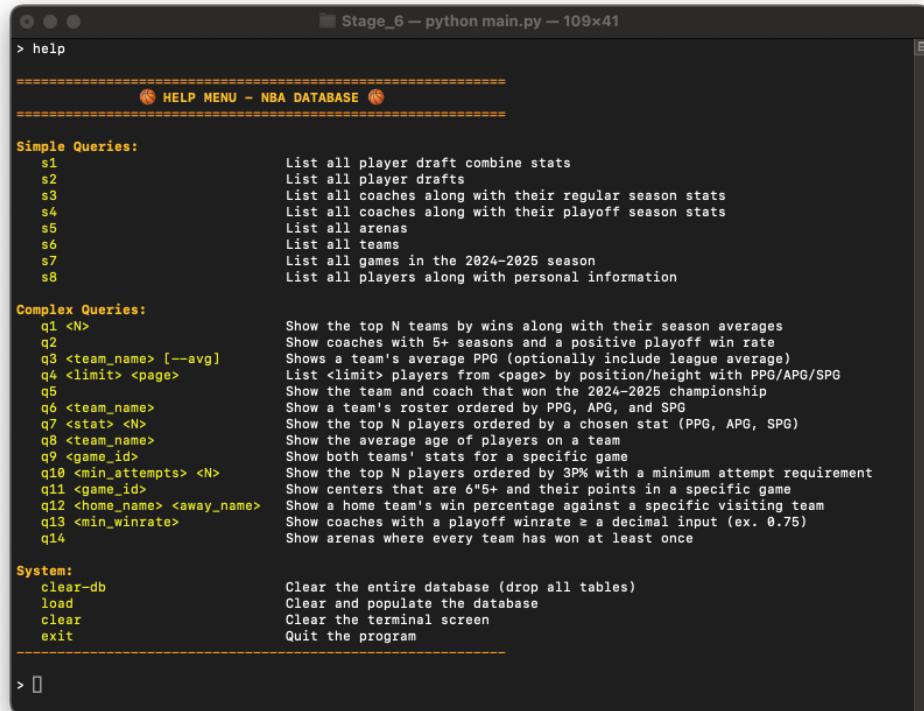
```
Stage_6 — python main.py — 67x18
(base) johnnylee@Johnnys-MacBook-Air-3 Stage_6 % python main.py
=====
🏀 NBA 2024–2025 SEASON DATABASE 🏀
=====

Welcome to the 2024–2025 NBA Season Database. This interface
allows you to explore teams, players, game, coaches, draft
stats, and more.

Type 'help' to see a list of available commands.

> []
```

Figure 1: Welcome Message Screenshot



```
Stage_6 — python main.py — 109x41
> help
=====
🏀 HELP MENU – NBA DATABASE 🏀
=====

Simple Queries:
s1 List all player draft combine stats
s2 List all player drafts
s3 List all coaches along with their regular season stats
s4 List all coaches along with their playoff season stats
s5 List all arenas
s6 List all teams
s7 List all games in the 2024–2025 season
s8 List all players along with personal information

Complex Queries:
q1 <N> Show the top N teams by wins along with their season averages
q2 Show coaches with 5+ seasons and a positive playoff win rate
q3 <team_name> [--avg] Shows a team's average PPG (optionally include league average)
q4 <limit> <page> List <limit> players from <page> by position/height with PPG/APG/SPG
q5 Show the team and coach that won the 2024–2025 championship
q6 <team_name> Show a team's roster ordered by PPG, APG, and SPG
q7 <stat> <N> Show the top N players ordered by a chosen stat (PPG, APG, SPG)
q8 <team_name> Show the average age of players on a team
q9 <game_id> Show both teams' stats for a specific game
q10 <min_attempts> <N> Show the top N players ordered by 3P% with a minimum attempt requirement
q11 <game_id> Show centers that are 6'5+ and their points in a specific game
q12 <home_name> <away_name> Show a home team's win percentage against a specific visiting team
q13 <min_winrate> Show coaches with a playoff winrate ≥ a decimal input (ex. 0.75)
q14 Show arenas where every team has won at least once

System:
clear-db Clear the entire database (drop all tables)
load Clear and populate the database
clear Clear the terminal screen
exit Quit the program

> []
```

Figure 2: Help Menu Screenshot

Arena	NumTeamsWon
Paycom Center	29
Rocket Mortgage Fieldhouse	27
Gainbridge Fieldhouse	26
Crypto.com Arena	23
Madison Square Garden	23
FedExForum	23
TD Garden	23
Toyota Center	23
Chase Center	22
Ball Arena	21
Fiserv Forum	21
Footprint Center	20
American Airlines Center	20
Little Caesars Arena	20
Target Center	19
Moda Center	19
Frost Bank Center	18
State Farm Arena	18
Kaseya Center	17
Kia Center	16
Golden 1 Center	16
Scotiabank Arena	16
United Center	15
Smoothie King Center	13
Spectrum Center	12
Wells Fargo Center	12
Barclays Center	12
Delta Center	11

Figure 3: Query Results Screenshot

## Appendix

Dylan's contributions

- Stage 1: Helped look for the datasets that connect to each other, created the first timeline with deadlines, reviewed Johnny's work for stage 1 and made any necessary corrections to it
- Stage 2: Created Rough draft EER model in drawio and wrote the 1 paragraph reminder of chosen data, wrote all justifications rough draft
- Stage 3: Created the final EER model with Johnny and Kameron's feedback, finalized justifications of EER Model and did some of the final relational model, completed Merging, normalizing and cleaning in this stage as well with documentation of my steps.
- Stage 4: Wrote a few English queries while participating in review for Kameron's stage providing any feedback if possible. Made sure the EER diagram was updated for stage 4
- Stage 5: Made sure the EER diagram was updated for any feedback given, did a few implementations of queries written in English and in SQL and wrote why an analyst might care about them. Participated in review again making sure final copy was well done.
- Stage 6: Helped think of the design of what the interface may look like. Wrote the SQL injection prevention plan. Participated in review making sure interface design was well structured and making any comments or adjustments as needed. Kept the EER model updated

- Final Project Report: Given queries for the interface to implement in Python. Started my account for authentication for where the database lives. Participated in review making sure any necessary tweaks or bugs were fixed before final submission

Johnny's contributions

- Stage 1: I was the organizer for this stage, meaning I did the bulk of the work. I found most of the datasets, did most of the Stage 1 write-up, and asked my group members for feedback.
- Stage 2: I helped review Dylan's work and made corrections where I felt it was necessary.
- Stage 3: Helped refine the EER diagram and assisted Dylan with normalizing the dataset. I completed my portion of the normalization using `pandas` in Python.
- Stage 4: I helped review Kameron's work and made corrections where I felt they were necessary. I also wrote a few English queries.
- Stage 5: Wrote queries 9, 10, and 14, as well as reviewed Dylan's and Kameron's queries. I also helped ensure the EER diagram was correct up to this point.
- Stage 6 + Final Project Submission: I was the organizer for this stage, so I took the lead on implementing the command-line interface. This included planning the codebase architecture by separating the system into `interface.py`, `database_manager.py`, `query_manager.py`, `data_loader.py`, and `main.py`. I implemented the user interaction logic, formatted query outputs, created the help menu and welcome message, and integrated all system-level commands, including creating, clearing, and populating the database. I also wrote most of the write-up, as well as worked with my group members and incorporated their feedback throughout the process.
- Final report: Wrote the **discussion of database** and **description of interface** sections.