COMP 3380 Final Project

NBA 2024–2025 Relational Database

Group Members

Dylan Beyak — UserID: 7974864

Johnny Lee — UserID: 7890682

Kameron Toews — UserID: 7975017

University of Manitoba
December 2025

Overwiew 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 an other.

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 chose 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.
- Statsistics 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.

A lot of cleaning and preprocessing was required before the data could be used in our database. We used `pandas` in Python to go through each dataset and fix issues like incorrect formats, missing values, and inconsistent attribute names. Some columns had to be removed because they were not useful for our model, and certain attributes were formatted incorrectly. For example, some height values were being read as years, and some dates, arena names, and team references needed to be corrected. We also combined separate tables and standardized the structure of each file, for example correcting column names, data types, and formatting, and made sure all keys lined up properly so they would load into our SQL tables without errors. Here are all the sources we used for our tables:

- Table for `Arena.csv` : 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
- 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
- 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 didnt 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 seperated 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 seperate.
  - 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.

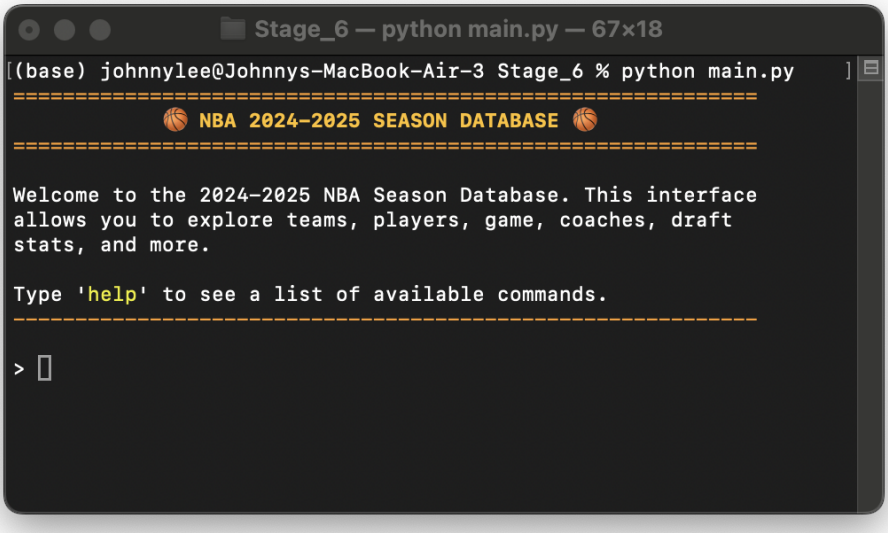

Figure 1: Welcome Message Screenshot

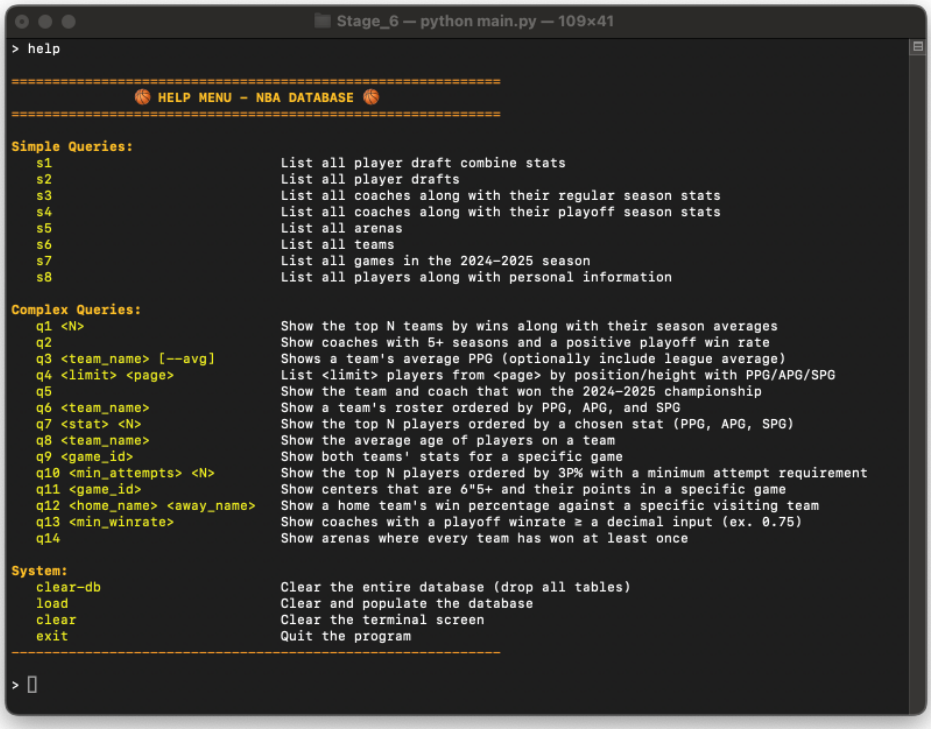

Figure 2: Help Menu Screenshot

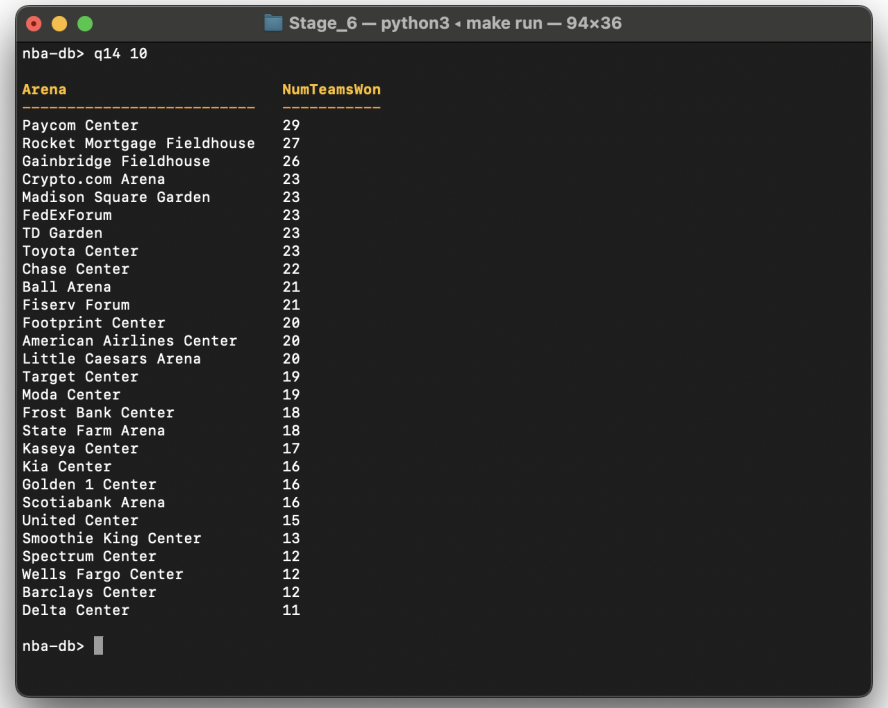

Figure 3: Query Results Screenshot

Description of Queries

In addition to 8 simple queries, which are used to ensure 100% of our dataset is accessible to the user, we also implemented 14 complex queries that enable analysts to draw important statistics from our dataset.

Using the interface for the NBA Database a user can easily see what to input for a given query and what it will return to the user. Figure 2 in the "Diagrams of Interface" section nicely outlines exactly what each of the 14 complex queries will return.

Justifications

1. `Q1:` This query will allow an analyst to quickly see the top performers for the season and the relevant important stats.
2. `Q2:` This query will allow an analyst to select the coaches that perform the best in the playoffs.
3. `Q3:` This query enables an analyst to grab the most relevant stat for a team in nba and gives the option to see how that team's ppg compares to the league average.
4. `Q4:` This is an important query because it allows an analyst to retrieve the primary stats for all players in the league while grouping them according to the role they play in their team.
5. `Q5:` While an analyst might already know this information as general knowledge. It is still important for every sports database to have a query that returns the team that won the championship that year.
6. `Q5:` An analyst would find this query useful, since it allows them to evaluate the most important contributors to a teams scoring by viewing the teams roster ordered first by points per page, then by assists per game , and lastly through steals per game.
7. `Q7:` This query allows an analyst to retrieve the leaders of the 3 major statistical categories, which enables a analyst to contract new graphics and datasets with the top performers in the league.
8. `Q8` This Query allows an analyst to evaluate whether a team should start considering rebuilding or push for a championship, while also evaluating a teams future.
9. `Q9:` This query allows an analyst to evaluate individual games and each teams performance in that game.
10. `Q10:` This is a fun cool stat but it also allows an analyst to find the best 3 point shooters in the league. Additionally this query allows an analyst to find the best shooters with a user specified minimum amount of attempts. This enables a given analyst to select for the best 3-point shooters for an specified volume.
11. `Q11:` This query allows an analyst to evaluate the performance of centers as well as potentially allow an analyst to see if height is correlated with ppg.
12. `Q12:` This query allows an analyst to evaluate how well one team performs against another
13. `Q13:` This query enables analysts to identify the top-performing coaches who have recently reached the NBA playoffs and evaluate their success based on historical playoff performance.
14. `Q14:` This query enables analysts to identify if a certain arena is unbiased towards a certain team winning.

Conclusion

For this project, our group used a relational database to model NBA statistics from the 2024/2025 season. However, this raises the question: could a graph database or document database model our data better?

Due to the interconnectivity of our datasets and the intricate joins required to construct our complex queries, our project could not be rewritten as a document database without additional difficulty and a loss of efficiency. Many of our complex queries use aggregate functions, complex joins, set theory, and grouping. As such, recreating them in a document database would be a nightmare, potentially requiring us to simplify our queries. A document database might be a better fit for a live presentation of play-by-play stats, where it could leverage its lightning fast writes and simple lookups without having to worry about storing complex nested relations.

While our relational database and queries could be recreated in a graph database using graph traversals, it would be overkill for our use case. SQL excels at capturing the simpler, well defined relationships captured in our data (e.g.,finding player stats for a team). A graph database would become far more useful if we were presenting very advanced analytics that focused heavily on deep relationships/traversals (e.g., find the players that passed to a player that passed to player that scored. Also known as a Hockey assist). Additionally, due to the structured and tabular nature of our data– where simple relationships can be easily and efficiently modeled in SQL–a relational database was a natural fit. In contrast, a graph database would require more storage to store the same data (due to needing to store nodes, edges etc.).

Overall, the structured, tabular nature of our data pairs best with a relational database, especially given our complex queries and the tabular data they return. While document and graph databases could support different types of queries, like real-time feeds or advanced connectivity analytics, they would either make our complex queries harder to construct or far less efficient.

Teaching tool

Could our database be used as a teaching tool for COMP 3380 in the future ? Absolutely, our database covers the vast majority of the topics covered in COMP 3380. It includes disjointed subsets, all 3 types of relations (1-to-1, m-to-m, and m-to-1), derived attributes, composite attributes, weak attributes, foreign keys and much more. Students would be able to write simple and complex queries using either the whole table or sections of it. One section that is well suited to assignment and test problems is the coach - Instructs - team -PlayOnTeam - player - PlayInGame - Game subsection. Using this subsection of the Database/EER diagram, a professor could ask students to construct all kinds of interesting queries.

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.

Kameron's contributions

- Stage 1: As Johnny was the leader for this sections I fulfilled the auxiliary role of double checking his work and providing feedback and corrections where necessary.
- Stage 2: Dylan took the lead in this section and I simply reviewed his work, adding feedback and corrections when necessary.
- Stage 3: Helped provide input to refine the final EER diagram for submission. Additionally, I created a updated timeline for the group as well as a more detailed plan for stages 4-6.
- Stage 4: I took the lead in this stage since I was more familiar with the NBA and sports statistics. In this stage I constructed the majority of the English queries. Additionally, I crafted a serious of justifications to explain why the queries would be relevant to a sports analyst. Lastly, I reviewed, corrected where necessary, and approved of Johnny's and Dylan's additions to the query list.
- Stage 5: I also lead this stage of the project. Within this stage I translated the majority of the English queries from stage 4 into SQL while incorporating the feedback provided by the grader for stage 4. As part of this stage I made sure that our queries were crafted using views,CTEs, set theory, joins, etc. Additionally I lead a team review of all of the queries and incorporated my teammates feedback into my query design while also providing my own feedback on the queries they contributed.

- Stage 6 + Final Project Submission: Johnny took the lead in this stage and honestly went above and beyond, so there was not too much for me to do. However I still error tested the interface with many types of invalid input to ensure the interface was robust to user error. Additionally, using my knowledge of the NBA, I validated the output of all the complex queries to make sure they actually returned what we intended for them. Specifically I found errors in queries outputs for queries 1,3,4, and 10. I also fixed the SQL for queries 12 and 13 as they did not work.
- Final Report: Wrote the **Conclusion and Queries** sections, as well as the **Teaching tool and justification** subsections.