ISTANBUL TECHNICAL UNIVERSITY
COMPUTER ENGINEERING DEPARTMENT

BLG 317E
Database Systems
Project Report

KickStats Team

150210926 : Mohamed Ahmed Abdelsattar Mahmoud
(mahmoud21@itu.edu.tr)

150220762 : Muhammed Yusuf Mermer
(mermer22@itu.edu.tr)

820220710 : Muhammed Can Özkurt
(ozkurtm22@itu.edu.tr)

150210907 : MHD Kamal Rushdi
(rushdi21@itu.edu.tr)

GitHub Repository: `KickStats Repo`

FALL 2024/2025

Contents

1 INTRODUCTION

The European Basketball Statistics Project aims to provide an interactive platform for analyzing and comparing basketball performance data and to manage it at the same time. This project focuses on leveraging a comprehensive dataset of European basketball games to create meaningful visualizations, perform statistical comparisons, and enhance decision-making processes for stakeholders. These abilities are provided for both an **Admin**, that manages the data, and a **User** who will see the representation of this data in an interactive and comprehensive way.

Our platform integrates a backend system powered by a **RESTful API**, ensuring efficient communication between the backend and the frontend. The frontend provides an intuitive interface for users to interact with data, filter games, and visualize results. The dataset comprises multiple tables, each serving a specific purpose, such as game metadata, statistical comparisons, and team performance history. A well-structured relational database schema forms the backbone of the system, adhering to normalization principles to maintain data integrity and minimize redundancy. To be able to achieve this, we have maintained and utilized our chosen dataset as will be demonstrated later.

This report outlines the methodologies, technologies, and design principles employed during the project's development. Starting from the initial data exploration to the implementation of interactive features, each component has been meticulously crafted to ensure scalability, reliability, and both the **User** and **Admin** satisfaction. The following sections provide detailed insights into the system architecture, database design, implementation strategies, and the outcomes achieved through this project.

2 OUTLINE AND TOOLS

In this project, we used several technologies to ensure efficient data handling, interactive features, and a smooth user experience. Each tool was chosen for its specific strengths and compatibility with the project's requirements. Below is a description of the technologies and why we used them:

2.1 MySQL

MySQL is an open-source relational database management system (RDBMS) that stores and organizes structured data. We used MySQL to create a robust database that could handle large volumes of basketball statistics, including game results, player performances, and team details. MySQL was chosen for its reliability, support for complex queries, and beginner-friendliness as all team members were learning SQL for the first

time. Another popular choice was PostgreSQL, but as our research showed it is best for big enterprises and projects which is not the scale of our project.

2.2 Flask

Flask is a lightweight web framework written in Python designed for building web applications. We used Flask to create the backend of our platform and develop a RESTful API to manage communication between the database and the front end. Flask was selected because it is a requirement for the course, so we didn't consider another backend framework.

2.3 React

React is a JavaScript library used for building interactive user interfaces. It allows developers to create reusable components and manage the state effectively. We used React to develop the frontend of our platform, ensuring that users could interact with the basketball data in a dynamic and visually appealing way. React was chosen because it is highly responsive and very popular, most of our team members wanted to introduce themselves to it as job opportunites and resources for it are plentiful.

2.4 Python Scripts (pandas)

Python, along with the pandas library, was used to preprocess and clean the dataset before loading it into the database. Pandas provide powerful tools for handling large datasets, making it easier to merge tables, handle missing values, and generate new insights. We chose pandas because of their efficiency in data manipulation tasks and their ability to easily process the data.

3 ER DIAGRAM

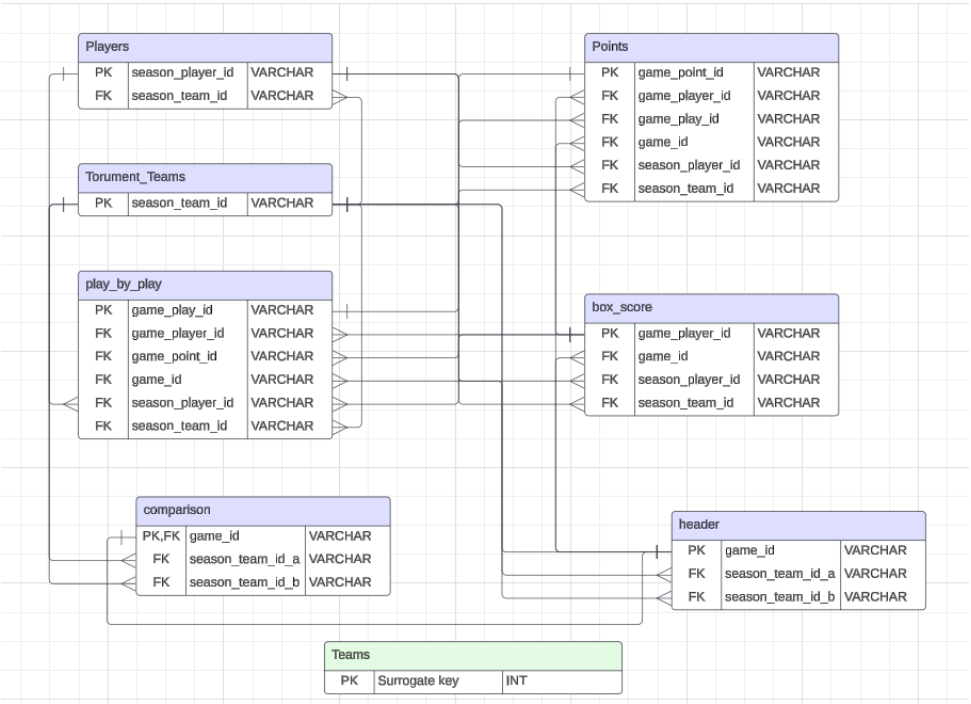


Figure 1: ER diagram [1]

All of our keys started by season (e.g E2007). This concept was used throughout all of our user view features. It allows the user to select the season data they want to visualize. Another design choice made was the header table as main and comparison as secondary due to them using the same Primary Key.

Note: Since our tables have a lot of columns, we have structured the ER Diagram so that it contains the important information, which is the primary keys and foreign keys.

3.1 Overview of the ER Diagram

The diagram represents the schema for our system designed to manage data related to sports tournaments, teams, players, and their performance in games. Key entities include players, teams, games, and their associated statistics.

3.2 Entities and Attributes

3.2.1 Players

- **Primary Key (PK):** `season_player_id` (unique identifier for a player in a specific season).

- **Foreign Key (FK):** `season_team_id` (links the player to their respective team for the season. (Tournament_Teams Table)

3.2.2 Tournament_Teams

- **Primary Key (PK):** `season_team_id` (unique identifier for a team in a specific season).

3.2.3 Points

- **Primary Key (PK):** `game_point_id` (unique identifier for a point scored in a game in a season).
- **Foreign Keys (FK):**
  - `game_player_id`: Links the point to a player. (box_socre table)
  - `game_play_id`: Links the point to a specific play (play_by_play table).
  - `game_id`: Associates the point with a specific game (Header table).
  - `season_player_id`: Links the point to the player for the specific season (Players Table).
  - `season_team_id`: Links the point to the team for the specific season (Tournament_Teams Table).

3.2.4 Play_by_Play

- **Primary Key (PK):** `game_play_id` (unique identifier for a specific play during a game).
- **Foreign Keys (FK):**
  - `game_player_id`: Links the play to a specific player. (Box_score table)
  - `game_point_id`: Associates the play with the point scored. (Points table)
  - `game_id`: Links the play to the game it occurred in. (Header Table)
  - `season_player_id`: Links the play to a specific player in the season. (Players table)
  - `season_team_id`: Links the play to a specific team. (Tournament_Teams Table)

3.2.5 Box_Score

- **Primary Key (PK):** `game_player_id` (unique identifier for a player in the game).
- **Foreign Keys (FK):**
  - `game_id`: Links the score to a specific game. (Header table)
  - `season_player_id`: Links the score to a specific player in the season. (Players table)
  - `season_team_id`: Links the score to a specific team in the season. (Tournament_Teams Table)

3.2.6 Comparison

- **Primary Key (PK):** `game_id` (unique identifier for a game).
- **Foreign Keys (FK):**
  - `game_id` : Links comparison table to header making comparison secondary table while Header primary table to enable logical connections throughout the database (Header Table)
  - `season_team_id_a`: Represents Team A in the comparison. (Tournament_Teams Table)
  - `season_team_id_b`: Represents Team B in the comparison. (Tournament_Teams Table)

3.2.7 Header

- **Primary Key (PK):** `game_id` (unique identifier for a game).
- **Foreign Keys (FK):**
  - `season_team_id_a`: Represents Team A in the game. (Tournament_Teams Table)
  - `season_team_id_b`: Represents Team B in the game. (Tournament_Teams Table)

3.2.8 Teams

- **Primary Key (PK):** `Surrogate Key` (auto-incremented integer key for teams).

3.3 More details about relationships:

1. **Players and Tournament_Teams:** Players are associated with teams for a specific season through the `season_team_id` foreign key.
2. **Tournament_Teams and Points:** Points scored in games are linked to teams for a specific season.
3. **Play_by_Play and Points:** The `play_by_play` table references the `Points` table to record specific plays that resulted in points scored.
4. **Play_by_Play and Games:** Each play is associated with a specific game through the `game_id`.
5. **Box_Score and Players:** The `box_score` table tracks player statistics in games.
6. **Comparison and Header:** The `comparison` table uses the `header` table to define comparisons between two teams (Team A and Team B), and they are connected through `game_id`.
7. **Teams:** Provides teams' logos and full names mapped to their abbreviations.

3.4 Cardinalities:

All of our relationships were One-to-Many **(1:N)**, except for one relationship between tables Header and Comparison which was One-to-One **(1:1)** because of the design choice explained in the Entities and Attributes subsection.

4 INDIVIDUAL CONTRIBUTIONS

As the **KickStats** team, we demonstrated excellent collaboration and communication throughout the development of the **KickStats** project. Each member contributed their expertise, supported others when challenges arose, and ensured alignment in both design and implementation. Our collective effort and synergy allowed us to create a cohesive and well-structured system, showcasing the strength of teamwork and mutual respect. Note: This was a very brief summary of what has each team member done throughout this project. For more detailed info the GitHub repository can be inspected.

4.1 Mohamed Ahmed Abdelsattar Mahmoud

In the project, I focused on the development of the **Header** and **Comparison** modules (`CUP_COMPARISON` - `LIG_COMPARISON` - `CUP_HEADER` - `LIG_HEADER`), contributing to both

backend and frontend functionalities, with some additional focus on the admin view and the home page.

- **Header Module:** Designed and implemented the `Header User View`, including backend queries like `get_(cup/lig)_header` and `get_distinct_games_with_like`. The frontend features detailed game statistics, dynamic visualizations, and an interactive winner section.
- **Comparison Module:** Developed the `Comparison User View`, implementing key functionalities like `get_win_loss_history`, enabling comparative team analysis, and optimizing queries for efficient data retrieval.
- **Admin Module and Home Page:** Enhanced on the general user experience by adding UI elements and constraints on inputs (using Regular Expressions) for data consistency and integrability.
- **Integration and Testing:** Ensured seamless integration of all modules with the project and conducted debugging.
- **Collaboration and Documentation:** Assisted with ensuring the dataset's consistency and compatibility at the beginning by writing data processing scripts.
- **Last Touches:** Designed and implemented the About and Contact pages for a better user experience.

4.2 Muhammed Yusuf Mermer

In the project, I contributed to core aspects of the system, focusing on data integration, visualization, and system design. Below is a summary of my responsibilities:

- **Teams Module:** Ranked teams based on their strategies using spider charts and graphs, highlighting key metrics like rebounds, assists, and blocks by normalizing values to show strengths and strategies clearly.
- **Points Module:** Implemented detailed shot analysis, showing exact court locations, shot types (e.g., two-pointer, three-pointer), and dynamic visuals to distinguish successful and missed shots.
- **Project Structure:** Set up the project's frontend with reusable React components and backend with RESTful APIs, ensuring seamless communication between the two systems.

- **Dataset Manipulation:** Merged dataset columns to create foreign keys for primary keys, resolving inconsistencies to establish proper database relationships.
- **Admin View Design:** First design of the admin view for CRUD operations with features like filtering, pagination, and column selection, ensuring data integrity through validations.
- **Team Logo and Name Connections:** Linked team logos and names to the database using custom scripts for accurate matching, even with name variations.

4.3 Muhammed Can Özkurt

In the project I was responsible for Play By Play and Box Score tables for both the Euroleague and Eurocup (`CUP_PLAY_BY_PLAY` - `LIG_PLAY_BY_PLAY` - `CUP_BOX_SCORE` - `LIG_BOX_SCORE`). I contributed to backend and frontend functionalities.

- **Play By Play Module:** Provided detailed basketball game events, highlighting key moments like the minute, player, and play description. It provides a structured and interactive view, showing player names, teams, and current scores for each event. This feature helps users analyze game dynamics and critical turning points.
- **Box Score Module:** Implemented detailed player statistics for basketball games, offering a comprehensive overview of individual performances. It highlights key metrics like points, two and three point field goals, free throws, rebounds, assists, steals, and turnovers.
- **POSTMAN Testing:** Conducted detailed debugging using POSTMAN software [2] to ensure the correct calling of APIs before building the frontend to ensure a smooth interaction between all elements.

4.4 MHD Kamal Rushdi

In the project, I focused on the development of the **Players** modules (`CUP_PLAYERS` - `LIG_PLAYERS`), contributing to both backend and frontend functionalities.

- **Player Module:** Designed and implemented the `Player User View`, including backend queries like `get_paginated_(cup/lig)_Player` and `get_distinct_teams_by_year`. The frontend features detailed Player statistics, dynamic visualizations, and an interactive Players points pieChart.
- **Testing final functionalities:** Ensured all the functionalities of all modules are working as intended and are following the same design choices.

- **ER diagram:** Drew the final ER diagram with all the changes, cardinalities, final design choices, and relations. Made sure all of its tables and logic follow the principles taught by DB course.

5 DATABASE PREPARATION

5.1 Connecting Tables

At first glance, the tables in our dataset did not seem clearly related to one another through primary key relationships [see Table 1 and Table 2]. However, upon closer inspection, we identified several common columns that allowed us to merge these tables and derive the primary keys of other tables. By analyzing the column patterns, we determined how these relationships could be established. Using our script, `scripts/merge_ids.py`, we merged the required columns to form new keys [see Table 3].

CUP_PLAY_BY_PLAY:

game_play_id	player_id	game_id
U2007_001_003	U2007_P000168	U2007_001
U2007_001_004	U2007_P000168	U2007_001
U2007_001_005	U2007_P000643	U2007_001

Table 1: Example of "Play by Play" structure before merging.

CUP_BOX_SCORE:

game_player_id
U2007_001_P000168
U2007_001_P000168
U2007_001_P000643

Table 2: Sample of primary key structure for "Box Score".

(new) CUP_PLAY_BY_PLAY:

game_play_id	game_player_id
U2007_001_003	U2007_001_P000168
U2007_001_004	U2007_001_P000168
U2007_001_005	U2007_001_P000643

Table 3: Sample of "Play by Play" structure after merge.

Table 4 shows the complete list of merged columns and the resulting foreign keys, which were used to establish relationships with primary keys in other tables.

After completing these merges, we removed the columns used in the merging process, as they were no longer needed for subsequent analysis.

While merging, we encountered additional challenges that required attention. Specifically, some columns lacked zero-padding, causing mismatches between foreign key values

Foreign Key	Merged Columns
game_player_id	game_id, player_id
game_play_id	game_id, number_of_play
season_player_id	season_code, player_id, team_id
game_point_id	game_id, number_of_play
season_team_id	season_code, team_id
season_team_id_a	season_code, team_id_a
season_team_id_b	season_code, team_id_b

Table 4: Merged column names and their resulting foreign keys.

and their corresponding primary key values [see Table 5 and Table 6]. To address this, we manually inspected all relationships and created additional scripts to identify and resolve these inconsistencies. These preprocessing steps ensured that our merge script worked seamlessly.

CUP_POINTS:

game_player_id	game_play_id
U2007_001_P000168	U2007_1_4
U2007_001_P000168	U2007_1_262
U2007_001_P000643	U2007_1_80

Table 5: Potential issue when merging columns without addressing inconsistencies

(new) CUP_POINTS:

game_player_id	game_play_id
U2007_001_P000168	U2007_001_004
U2007_001_P000168	U2007_001_262
U2007_001_P000643	U2007_001_080

Table 6: Corrected "Points" structure after resolving zero-padding issues.

5.2 Dataset Utilization

After we inspected the dataset and merged the needed primary and foriegn keys, we noticed that there are further enhancements that we can do before going to use the dataset in the project, and we will take a glance on the scripts used to utilize the tables in this section.

Note 1: All scripts have been written in `python` due to its simplicity.

Note 2: We will not be able to show all the scripts in here, but we will show the most important ones of them.

5.2.1 Handling Empty Integers and Corrupted Values

*We will take tables `CUP HEADER` and `LIG HEADER` as examples here.

In some cases, we found that some columns work as integers, but at the same time there

are `null` values in them, and this caused some problems in the backend. So, an easier solution was to use a **sentinel** in such cases, like what we did to both `CUP HEADER` and `LIG HEADER`.

Moreover, there were corrupted values in the scores columns, as we found that some games - in either *euroleague* or *eurocup* - have a *game-time* of **40:00:00 minutes** and the final scores are the same as the **4th quarter's score**, so logically the game had no extra time, and therefore should have **0** in the *4 extra time quarter scores* columns like **90%** of the cases, but this was not the case for some records, and the same score of the final quarter was written into the extra quarters. Accordingly, we managed to write 0 in the quarter scores columns for making the dataset consistent. Here is the most important part in the `Python` script that we wrote that shows how we did manage both the empty integer and corrupted entries cases:

```
# Replace empty cells in the specified columns with 0
for column in columns_to_process:
    df[column] = df[column].fillna(0)

# Compare and update values for specific conditions
# Compare score_extra_time_1_a with score_quarter_4_a
df['score_extra_time_1_a'] = df.apply(
    lambda row: 0 if row['score_extra_time_1_a'] == row['
       score_quarter_4_a'] else row['score_extra_time_1_a'],
       axis=1
)

# Compare score_extra_time_1_b with score_quarter_4_b
df['score_extra_time_1_b'] = df.apply(
    lambda row: 0 if row['score_extra_time_1_b'] == row['
       score_quarter_4_b'] else row['score_extra_time_1_b'],
       axis=1
)
```

5.2.2 Handling Integer columns containing strings

*We will take tables `CUP BOX SCORE` and `LIG BOX SCORE` as examples here.

In some cases, the dataset had columns that contained only integers, but for some odd cases there were also strings that was a **barrier** in the dataset consistency. An example of that is the `dorsal` column in both `CUP BOX SCORE` and `LIG BOX SCORE` which represents

the **player jersey's number**. But the problem was that these 2 tables contained records for both *individual players and whole teams*, so for the records of teams, the `dorsal` entry was written as **TOTAL**, which is a string. So, to solve that problem we have replaced **TOTAL** with a **sentinel** (**-1**) as can be seen from this line in our script:

```
# Replace "TOTAL" in the "dorsal" column with -1
df['dorsal'] = df['dorsal'].replace("TOTAL", -1)
```

5.2.3 Adding columns to use

*We will take tables `CUP HEADER` and `LIG HEADER` as examples here.

In the 2 tables of `Header`, we noticed that we can make use of an additional column called `winner`, which shows which team has won the game, and this is determined by the final scores in the game, so that this column can have only 3 values: `team_a`, `team_b`, and `draw`, and this can be seen clearly from the script we wrote in this line:

```
# Create the 'winner' column based on the conditions
df['winner'] = df.apply(
    lambda row: 'team_a' if row['score_a'] > row['score_b']
    else ('team_b' if row['score_a'] < row['score_b']
    else 'draw'),
    axis=1
)
```

5.3 Transferring Tables to Database

5.3.1 Creating Tables:

The process of creating tables from the provided MySQL file was relatively straightforward. We carefully reviewed and documented all updated column names for each table in the file to ensure consistency and accuracy during the migration.

5.3.2 Loading Tables:

Loading the tables into the database required more attention due to specific MySQL constraints. One challenge was that MySQL mandates files to be placed in a specific directory, as it does not have access rights to system folders by default. This required additional preparation to place the files in the correct location.

After the initial attempts to load data, we encountered several inconsistencies in the dataset that needed to be addressed. For instance:

- Some columns had missing values where the preparer did not provide any data.
- MySQL was unable to handle empty strings or invalid values such as `infinity`.

To manage these issues, instead of directly loading data into problematic columns, we used intermediate variables with the same names as the columns (e.g., `@valuation_per_game`). This allowed us to preprocess and clean the data before updating the actual table columns. For example:

- `UPDATE LIG_PLAYERS SET valuation_per_game = NULLIF(@valuation_per_game , '');`
- `UPDATE LIG_PLAYERS SET valuation_per_game = NULLIF(@valuation_per_game , 'inf');`

This approach was applied selectively to only the necessary columns to avoid unnecessary performance overhead during the initial setup process.

5.3.3 Building Foreign Key Relations:

After properly connecting the tables as explained in subsection 5.1, we initially assigned foreign key relationships during the `CREATE` statement for MySQL tables. However, while performing the load operation, we encountered numerous errors and warnings. These issues arose because, despite having well-connected tables, certain records referenced foreign keys that did not exist as primary keys in their parent tables. This inconsistency occurred because some tables were populated earlier than others or lacked complete information.

To address this problem, we decided to modify our approach. Instead of defining relationships between tables during the `CREATE` statement, we used `CREATE` solely to define the table structure and its columns. After successfully `LOAD`ing the necessary data from CSV files, we identified and eliminated problematic rows where foreign keys referenced non-existent primary keys. For example, we used the following query to handle such cases:

```
DELETE FROM CUP_HEADER
WHERE season_team_id_a NOT IN
    (SELECT season_team_id FROM CUP_TEAMS)
OR season_team_id_b NOT IN
    (SELECT season_team_id FROM CUP_TEAMS);
```

In this query, `season_team_id_a` and `season_team_id_b` are foreign keys referencing the `season_team_id` column in the `CUP_TEAMS` table. If a foreign key value did not exist as a primary key in the parent table, the corresponding row was deleted. Since the `CUP_HEADER`

table did not contain any additional foreign key columns, no further `OR` conditions were needed.

After repeating this process for all tables, we ensured that no rows or columns remained that could cause relational integrity issues. We then used the `ALTER` statement to establish the necessary foreign key relationships between tables. To maintain data integrity and ensure consistency across tables, we applied the `ON DELETE CASCADE` and `ON UPDATE CASCADE` options for all foreign key constraints. These options prevented orphaned records and ensured that updates or deletions in parent tables were properly reflected in child tables, and this can be clearly seen from the file `backend/table_initialization/foreign_keys.sql`.

5.4 Team Logos and Full Team Names

In the process of building our user views, we aimed to present the data in a more visually appealing and comprehensive way. After all, how can you effectively showcase a basketball competition without team logos or even the full names of the teams? In this subsection, we address how we incorporated team logos and full names into our database and how they were stored in a MySQL table.

The `Header` table played a central role in this process. It contained the `season_team_id`, which allowed us to extract team abbreviations, and the `team` column, which held the full names of the teams—data that was not available in most other tables. To leverage this, we used our script, `scripts/extract_teams.py`, to identify all unique team names along with their abbreviations and saved the results to a new CSV file.

This operation was performed for both the `LIG` and `CUP` tournaments. We then merged the unique team data from both tournaments using the script `scripts/merge_cup_lig_teams.py`, which ultimately produced a consolidated CSV file containing all distinct team names and their abbreviations.

While we now had all the team names and abbreviations, we were still missing the team logos. After researching how to obtain these logos, we discovered a GitHub repository that made the task significantly easier. Special thanks to Timur Kulenović's sports-logos repository for providing Euroleague team logos. Although the repository primarily featured Euroleague teams, many of these teams also participated in Eurocup, so using the available logos from the repository was sufficient for our purposes.

Obtaining the logos was only the first step. We needed to link the logos to the corresponding teams in our CSV file. However, the team names in the sports-logos repository did not always match the names in our new CSV file. To address this, we used a custom script called `scripts/team_logo_search&merge.py`. This script followed these steps:

1. Split the logo file names into individual words and store them.
2. Split the full team names in the CSV file into individual words and store them.
3. For each team in the CSV file, check if any words from the team name also appear in the logo file names:
    - If matching words are found, select the logo file with the highest number of matching words and assign its URL to a new column, `image_url`, in the CSV file.
    - If no matching words are found, leave the `image_url` column empty for that team.

This approach worked exceptionally well. Nearly all teams were correctly matched with their logos, even for teams whose logos and names changed across seasons due to sponsorship changes. The logic of selecting the logo file with the most matching words ensured that the appropriate logo was assigned to each team. After completing this process, we added the updated CSV file with the `image_url` column to our database, following a similar method as other tables.

6 WEBSITE

6.1 Website Structure and Goal

The goal of the website is to provide detailed information about Euroleague and Eurocup basketball tournaments to users. Visitors can explore how their respective teams performed during the season, analyze their strategies, evaluate player performances, and compare teams with one another. The site consolidates all this information in a visually appealing and user-friendly manner for every basketball enthusiast. See Figure 2 for the website's sitemap.

Figure 2: Sitemap [3]

As shown in the sitemap, the main menu initially asks users to select one of two tournament options: Eurocup or Euroleague (see Figure 3).

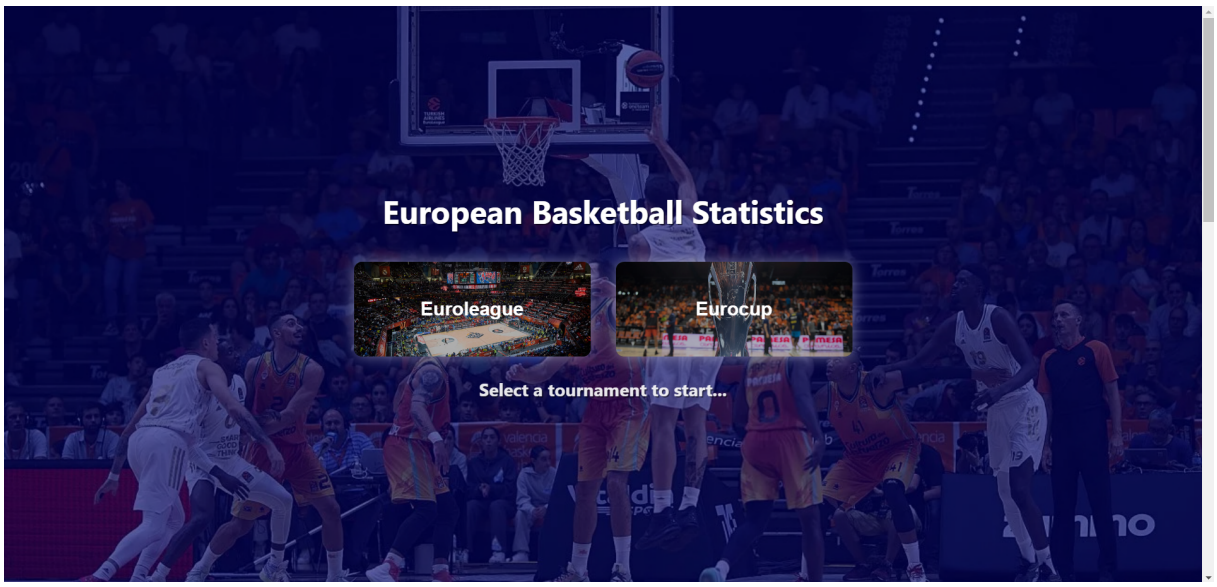


Figure 3: Main Menu Screen for Tournament Selection

After selecting a tournament, the page automatically scrolls down, prompting users

to choose one of seven available options for analysis. These options are consistent across both tournaments (see Figure 4). Users can then begin analyzing various aspects of the tournaments.

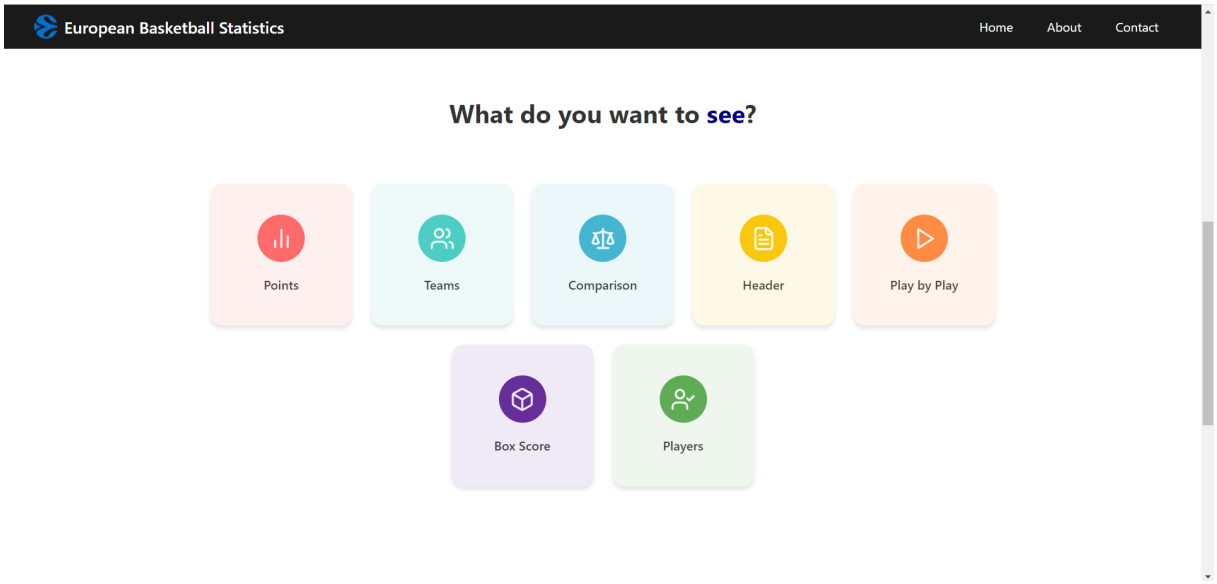


Figure 4: Seven Analysis Options for Users

In addition to the home screen, users can navigate to other pages for both tournaments through the footer (see Figure 5). Clicking on the website's logo also redirects users to the main page.

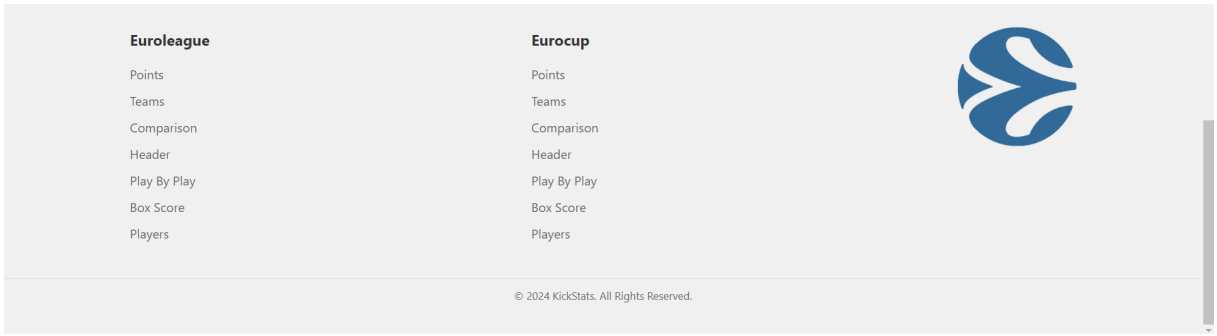


Figure 5: Footer Navigation

The header (see Figure 6) also provides quick access to the main page by clicking on the logo, the website title, or the "Home" button. Additionally, the header contains links to the "About" (Figure 7) and "Contact" (Figure 8) pages for user convenience.

Figure 6: Header Navigation

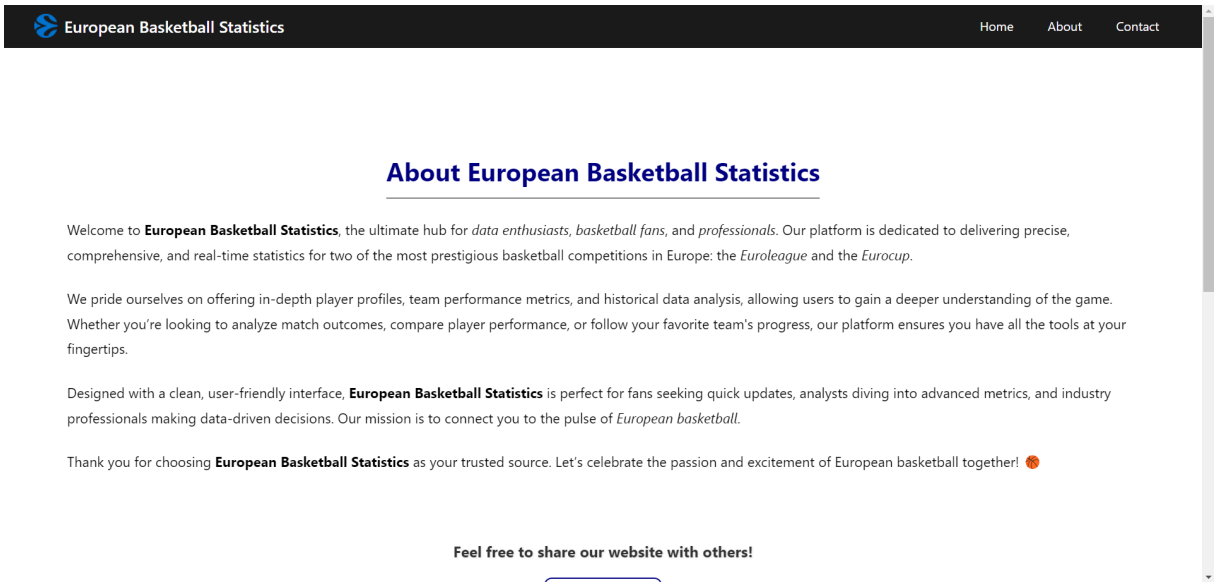


Figure 7: About Page

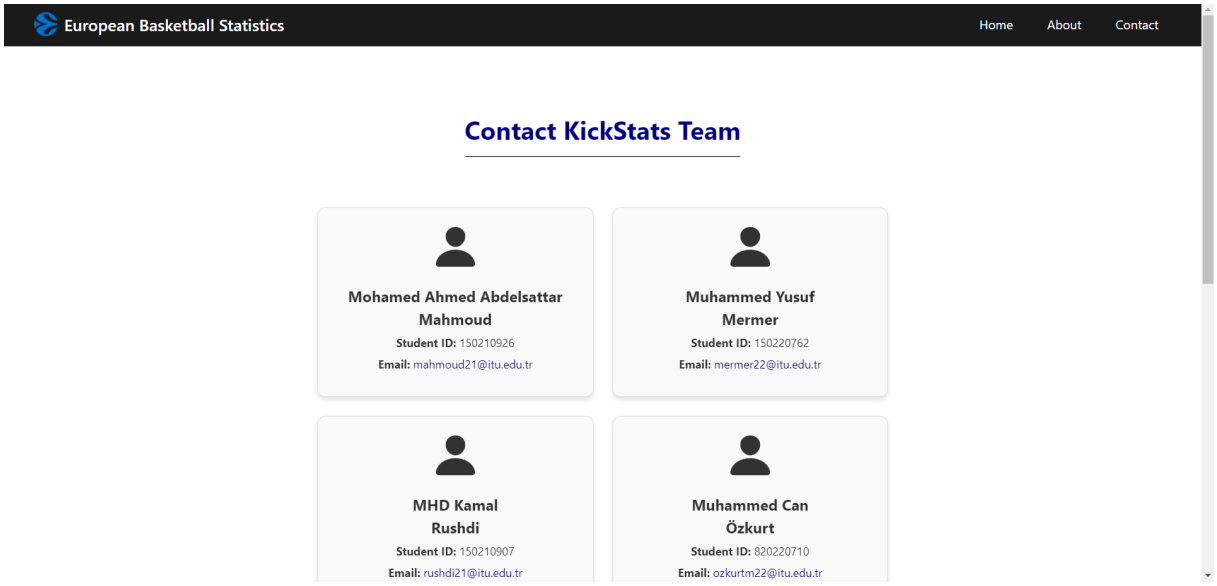


Figure 8: Contact Page

6.2 Admin View

The Admin View page is a core feature of the European Basketball Statistics platform, designed to perform **CRUD** *(Create, Read, Update, Delete)* operations on various datasets. The page allows administrators to efficiently manage data for different tables, such as **Players**, **Points**, **Teams**, and **Header**. The design of the **Admin View** page is consistent across all datasets, ensuring a standardized interface for managing entities. The primary focus of the Admin View is to facilitate data management through CRUD operations while maintaining data integrity and preventing errors caused by incorrect

data types or formats. Additionally, pagination controls are provided to handle large datasets, allowing administrators to navigate data efficiently by specifying rows per page and navigating between pages.

6.2.1 Functionalities and Design

The Admin View page offers essential CRUD operations. The **Add** button allows administrators to create new entries in the dataset. This operation includes a form for entering the required attributes while ensuring the correctness of data types. The **Delete** button removes records based on their primary key. The **Update** button allows modifications to existing records by selecting an entity using its primary key and updating specific attributes. Additionally, error messages are displayed if invalid data types or unexpected values are entered, improving usability and error handling.

By default, the Admin View displays only a subset of columns for simplicity. Administrators can use the **More Columns** button to include additional columns in the view, tailoring the table display to their needs. The **Add Filter** button enables users to filter data based on specific attributes, streamlining the search for relevant records, and even they have the ability to do multiple filtering. Pagination controls further enhance the usability by allowing users to manage datasets with ease, offering options to navigate between pages and adjust the number of rows displayed per page. These features ensure that the interface is both powerful and user-friendly.

6.2.2 Backend Implementation

The backend is designed with a flexible architecture to support all CRUD operations, dynamically interacting with the database.

The **Create (POST)** functionality is implemented through dynamically constructed SQL queries. For instance, when adding a team's season statistics, the system converts input data into a dictionary, builds a query using column names and placeholders, and executes the query with the provided values. This approach ensures flexibility, allowing the addition of records across various datasets without hardcoding specific attributes.

The **Read (GET)** operation supports multiple modes of data retrieval to accommodate diverse use cases. It enables fetching a single record by its primary key, such as retrieving the detailed statistics of a specific team for a particular season. Additionally, it allows retrieving all records from a table, for example, fetching the performance metrics of all players in a league. To manage large datasets effectively, the Read operation includes pagination, which limits the number of rows retrieved per request and specifies an offset for navigating through datasets. This is particularly useful for extensive data, such as play-by-play statistics for an entire season, ensuring efficient and organized data handling.

The **Update (PUT)** functionality is designed to dynamically construct SQL queries to modify only the fields provided by the user. For example, when updating a team's performance metrics, the system identifies the attributes to be updated, generates a query for these specific fields, and ensures that other data remains unaffected. This design minimizes the risk of unintended overwrites, improves efficiency, and ensures data integrity.

The **Delete (DELETE)** operation enables administrators to safely remove records based on their primary key. For instance, deleting a record for a specific player involves executing a delete query with the player's unique identifier. Robust error handling is implemented to manage database constraints, ensuring that dependent records, such as those linked to teams or games, are not inadvertently affected. This ensures that the database's relational integrity is maintained.

Backend endpoints also include advanced pagination and filtering capabilities to enhance data navigation and exploration. Pagination is implemented by dynamically adding `LIMIT` and `OFFSET` clauses to SQL queries, enabling seamless browsing through large datasets like player statistics across multiple seasons. Filtering is achieved using dynamic `WHERE` clauses that adapt to user-defined parameters, allowing for targeted data retrieval. For example, administrators can filter results to display teams with a specified number of wins or players exceeding a certain valuation score. Sorting functionality enables users to organize records based on specified columns, either in ascending or descending order, providing an intuitive and customizable data exploration experience.

This backend implementation provides a comprehensive and efficient foundation for managing the basketball statistics database. Its dynamic query construction ensures adaptability to varying data structures, while robust pagination, filtering, and error handling mechanisms enhance performance and user experience.

6.2.3 Frontend Implementation

The frontend is implemented with React and integrates seamlessly with the backend, utilizing its dynamic CRUD operations and robust capabilities. The Admin View serves as the main interface for data management, offering features like pagination, filtering, sorting, and column selection, all closely aligned with the backend.

The **Add** operation uses a form that validates required fields before sending data to the backend's **Create (POST)** functionality. The backend dynamically constructs and executes the SQL query to add the new record.

The **Read (GET)** functionality is leveraged to fetch single records, retrieve all records, or use pagination to navigate large datasets. Pagination controls interact with the backend's `LIMIT` and `OFFSET` clauses, ensuring efficient data browsing.

The **Update** operation enables users to modify specific attributes of a selected record

through an interface that aligns with the backend's **Update (PUT)** functionality. Only the fields provided by the user are updated, ensuring efficiency and data integrity.

The **Delete** operation allows users to remove records by specifying the primary key. This is integrated with the backend's **Delete (DELETE)** functionality, with error handling to manage database constraints and confirmation dialogs to prevent accidental deletions.

Advanced filtering and sorting in the frontend utilize the backend's dynamic `WHERE` clauses and sorting capabilities, allowing users to retrieve targeted results and organize data based on specified columns. Pagination controls further enhance usability, enabling users to navigate datasets efficiently.

Error handling is integrated into all operations, ensuring invalid inputs or failed requests are clearly communicated to users. The synergy between the backend's flexible architecture and the frontend's intuitive design provides a reliable and user-friendly platform for managing basketball statistics.

6.2.4 Key Features

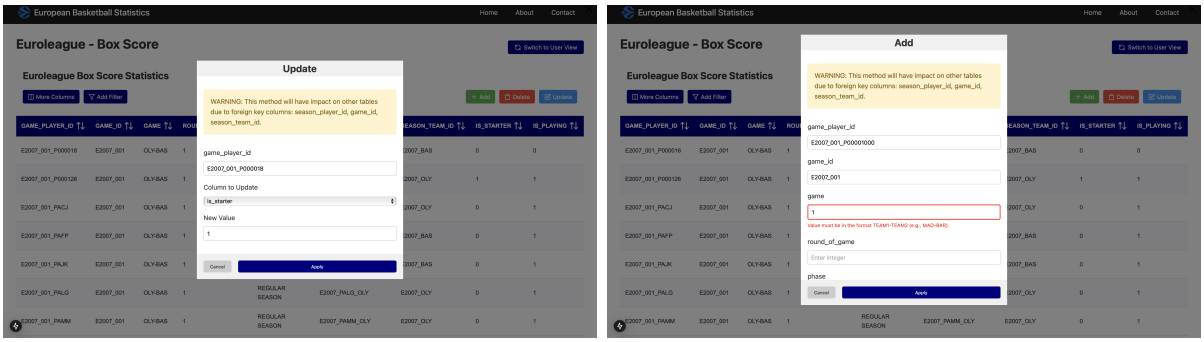
The Admin View ensures data integrity and user-friendly interaction by implementing the following features:

- Validation of data types to prevent errors during CRUD operations.
- Protection against modifying foreign key attributes to not present values to maintain relational integrity.
- Pagination and sorting to efficiently navigate large datasets.
- Dynamic column selection and filtering to tailor the view to user requirements.
- Clear error messages to guide users when invalid inputs are detected.

6.2.5 Usage Workflow

The workflow begins with administrators selecting the table they wish to manage. By default, a subset of columns is displayed, but additional columns can be added using the **More Columns** button. To locate specific records, filters can be applied through the **Add Filter** button. New records are added via the **Add** button, which opens a form for data entry. Records can be updated by selecting the primary key and modifying specific attributes through the **Update** button. Records are deleted by specifying the primary key and confirming the action. Pagination controls enable users to navigate efficiently through large datasets, ensuring a seamless data management experience. All operations

are validated for data type and format correctness, and errors are displayed if any issues arise. Example operations like the Add and Update are shown in Figure 9.



(a) Update Operation (b) Invalid Add Attempt

Figure 9: Example operations in the Admin View

The Admin View has a powerful and intuitive interface for managing the dataset. By integrating advanced backend logic with a responsive frontend, it ensures that administrators can efficiently perform CRUD operations while maintaining data integrity. Features like dynamic column selection, filtering, and pagination enhance usability, making the Admin View a critical tool for data management in the European Basketball Statistics platform.

6.3 User Views

6.3.1 Team User View

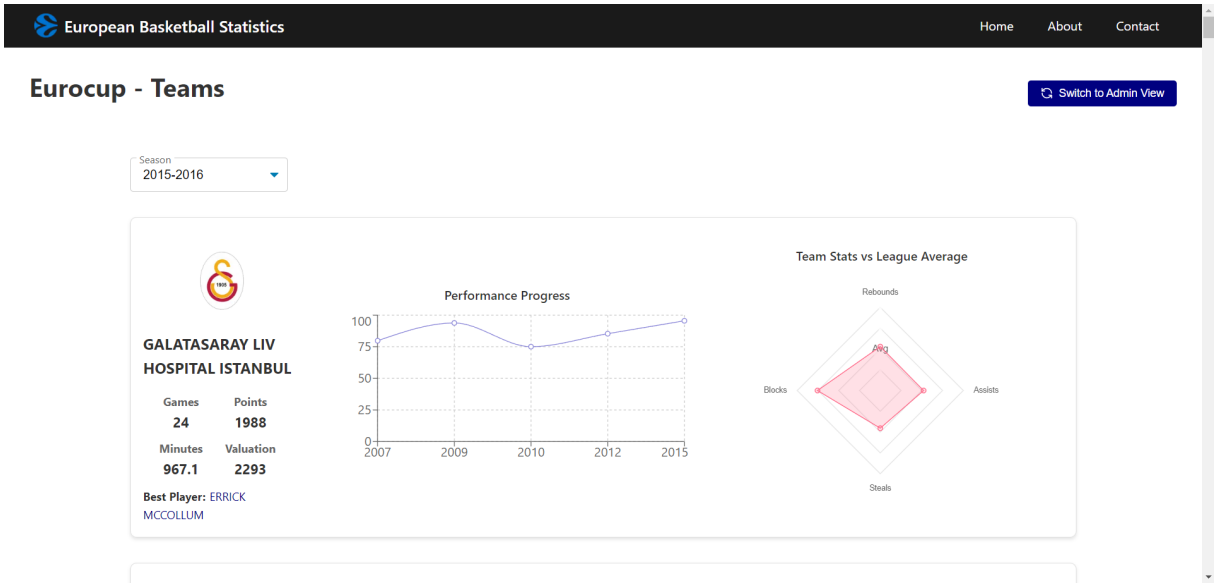


Figure 10: Team User View

The Team User View provides an overview of how teams performed during a specific season. Users first select the season from a dropdown menu, and the system displays team statistics, sorted from the highest-scoring team to the lowest.

Since the number of teams in a season is relatively small, we opted to load all teams without implementing pagination. As a result, there is no need for `LIMIT` or `OFFSET` in our SQL queries.

- **get_teams_with_like Function:**

  This function is responsible for retrieving all relevant team data for a given season. The `like_pattern` parameter is passed from the API route function to this model function. Using this pattern, the function filters data based on the season by applying a `LIKE` clause on the `season_team_id` column. The `season_team_id` contains the season number in its first five characters, where the first character represents the tournament type, and the remaining four characters represent the year.

  In addition to retrieving statistics for the selected season, the function also identifies the best player for each team. To achieve this, we use a `LEFT JOIN` with the `PLAYERS` table. The `LEFT JOIN` ensures that if there is corrupted data in the player's table—such as missing player records for a season—the function will still return the team's statistics. The join condition is based `ON` the `season_team_id`, ensuring that team and player data correspond correctly.

  To determine the best player, we extend the `WHERE` clause with a nested query. The nested query identifies the maximum points scored (`MAX(PLAYERS.points)`) by a player for a given `season_team_id`. The outer `WHERE` clause then filters rows to include only the player corresponding to this maximum value.

  Finally, we optimize the query by selecting only the columns required for the user interface, minimizing data retrieval overhead and ensuring efficient response times.

- **get_teams_by_abbrs Function:**

  The `get_teams_by_abbrs` function is designed to retrieve the progress graphs displayed for each team. Instead of sending multiple queries for individual teams, we pass a list of all team abbreviations as parameters to the SQL query, optimizing performance and reducing delays.

  To ensure the progress shown is relevant to a specific season, the function also accepts a `max_year` parameter. From the React frontend, we always send the current season as `max_year`, allowing the progress graphs to display data only up to the selected season.

Within the SQL query, the filtering logic uses the `SUBSTRING(season_team_id, 2, 4)<= max_year` condition in the `WHERE` clause. This ensures that only data up to the specified `max_year` is retrieved. Additionally, the function includes a `LIKE` filter for each team's abbreviation to match the relevant records.

The query is further refined to return only the necessary columns, as specified by the frontend. Currently, the React frontend determines the required column as `value_per_game`, which is included in the query's `SELECT` statement. This streamlined approach makes sure efficient data retrieval and a responsive user experience.

- **get_average_values_by_season Function:**

  The `get_average_values_by_season` function calculates average values for key columns, `total_rebounds_per_game`, `assists_per_game`, `steals_per_game`, and `blocks_favour_per_game`, for a given season. It also uses a `LIKE` statement to filter data for the specified season.

  Why is this function necessary? It is essential for generating the spider chart. The spider chart visualizes the specified four metrics for each team. However, raw values alone do not provide meaningful insights. For instance, a high value in one column does not necessarily indicate team success; it could simply mean that the metric is generally high across all teams. Without scaling these values against their average, the spider chart would lack context and fail to highlight meaningful differences.

  By normalizing the column values relative to their seasonal averages, the chart effectively reveals each team's unique strengths and strategies. With the current implementation, users can clearly observe how a team performed in these four essential metrics and better understand their playing style or strategy during that season.

6.3.2 Points User View

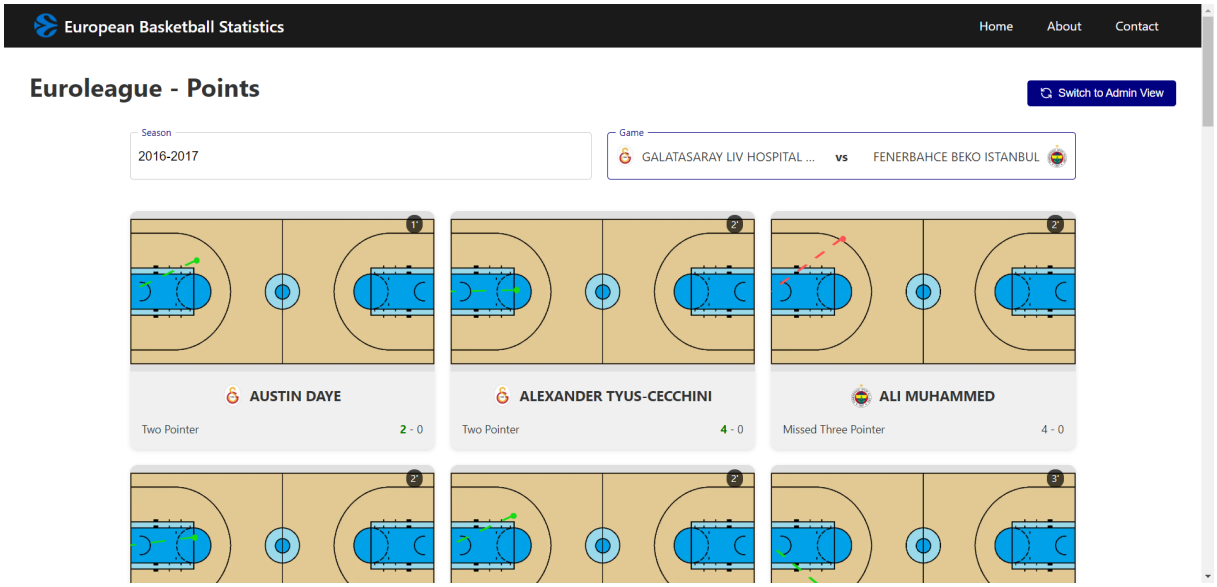


Figure 11: Points User View

The Points User View provides a detailed breakdown of scoring attempts within a match. On the initial screen, users are prompted to select a season and then a specific competition between two teams. Once selected, the view displays blocks of information for each scoring attempt, including:

- The location on the court where the player attempted the shot.
- The team making the attempt.
- The current score at the time of the attempt.
- The type of score being attempted (e.g., two-pointer, three-pointer).
- The player's name and the minute the attempt occurred.

Below is a detailed explanation of the queries used and how React processes them:

- **get_distinct_games_with_like Function:**

  This function is a critical part of our implementation. When a season is selected, the React frontend sends a request to the backend to retrieve all `DISTINCT games` from the `POINTS` table. Why is this necessary? Because the number of scoring attempts significantly exceeds the number of unique games, and we need to filter the points data to display information for a specific match.

  Once all distinct games are retrieved, the backend separates the competing teams and extracts their abbreviations. These abbreviations are passed to the `get_teams`

function via API calls, which returns the full names and logos of the teams. By combining this data, the application constructs a well-organized dropdown menu that allows users to select a specific match easily.

- **get_paginated_cup_points_with_like Function:**

  This function leverages the `LIKE` functionality, similar to the implementation in the Team User View, combined with **pagination** used in the Admin View. By using this approach, we retrieve the necessary information from the database to display to the user efficiently.

  Based on the value in the `points` column (representing point changes), the visual elements are updated dynamically:

  – The trajectory and shot position are displayed in green if `points` is greater than 0; otherwise, they remain red.

  – A similar logic is applied to the score color on the team indicator located in the bottom-right corner.

  The `player` field provides the shooter's name in the format `SURNAME, NAME`. To improve readability, it is converted and reformatted accordingly.

  A significant challenge in this implementation was scaling points on the visual representation. Since no reliable information was available online about the scales, we developed a custom script, `scripts/getmaxmin.py`. This script calculates the maximum and minimum values for the specified file. Using this data, we estimated the average as the midpoint of the max and min coordinates and used the difference (`max - min`) as the scaling factor. This solution worked effectively in the end.

6.3.3 Comparison User View

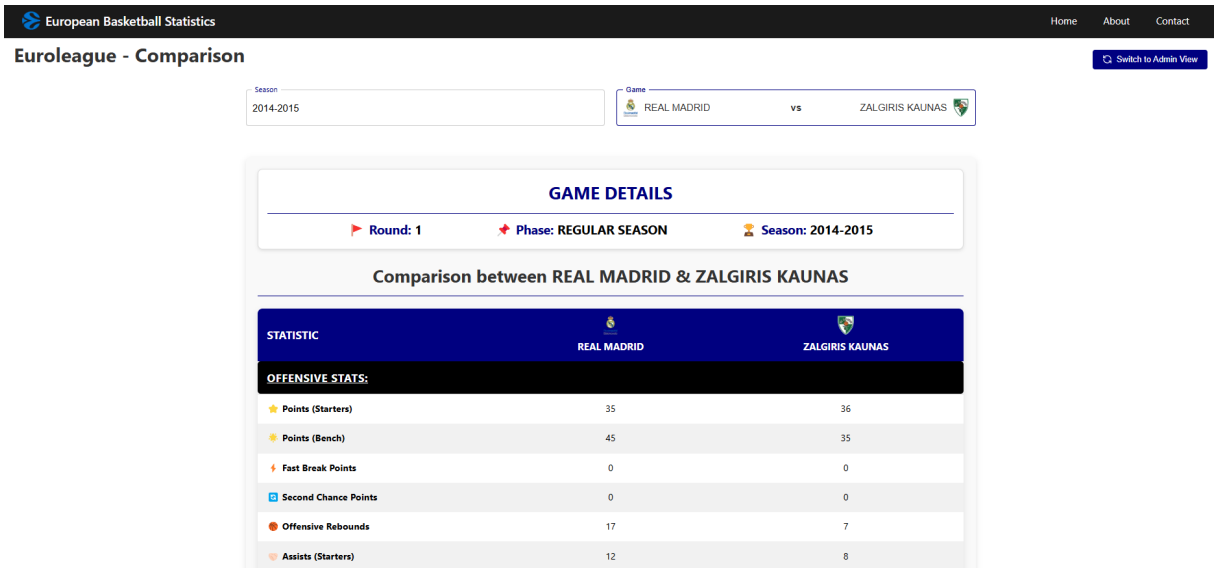


Figure 12: Comparison User View

The Comparison User View enables users to analyze detailed comparative statistics for specific matches. Users can filter the matches by season and select a particular match to view a breakdown of the 2 teams performances. The interface presents various statistics such as points scored, rebounds, assists, turnovers, and more, organized into offensive, defensive, and performance categories.

To ensure efficient data retrieval, we implemented backend queries that fetch the required match statistics while filtering and sorting the results based on the user's selections. Below are the key queries used to power this user view:

- **get_distinct_games_with_like Function:**

  This function plays a pivotal role in the system by retrieving a list of distinct matches from the `CUP_COMPARISON` or `LIG_COMPARISON` table. The purpose is to populate a dropdown menu in the user interface, enabling users to easily select a match. Below are the key components and functionalities of this function:

  - **Filtering with the `LIKE` Clause:**

    The function takes a `like_pattern` as an input parameter, which is used to filter matches by season or any other user-defined criteria. For example, if the user selects the 2023 season, the `like_pattern` is set as `U2023%` or `E2023%`, depending on whether it is the *euroleage* or *eurocup*. The `%` wildcard allows the query to match all game IDs starting with `U2023` or `E2023`.

– **Distinct Matches:**

By using the SQL `SELECT DISTINCT` statement, the function ensures that the results contain only unique matches. This prevents redundancy in the dropdown menu, enhancing user experience by displaying each match only once.

– **Dynamic Column Selection:**

The function supports dynamic column selection through the `columns` parameter. If the parameter is provided, the query retrieves only the specified columns. For instance, if the `columns` parameter includes `game` and `game_id`, the query fetches both columns. If no columns are specified, the default column `game` is used.

– **Mapping Results:**

The function maps the fetched rows to dictionaries or raw values based on the columns selected:

* If multiple columns are specified, the results are returned as a list of dictionaries, where each dictionary maps column names to their respective values.
* If only the default column `game` is selected, the results are returned as a simple list of values.

– **Error Handling and Resource Management:**

The function includes robust error handling to catch any database errors and ensure that connections and cursors are properly closed in the `finally` block. This prevents resource leaks and ensures the stability of the application.

How It Works in Practice: When the user selects a season in the frontend, the `like_pattern` parameter is generated and sent to this function through an API call. The function filters matches using the season prefix (e.g., `U2023`) and returns distinct matches. These matches are displayed in the dropdown menu, allowing the user to select a specific match for further analysis.

By combining efficient filtering, distinct selection, and dynamic column handling, the `get_distinct_games_with_like` function ensures that the dropdown menu remains organized, responsive, and user-friendly. And the output (the games dropdown list) is shown in Figure 13.

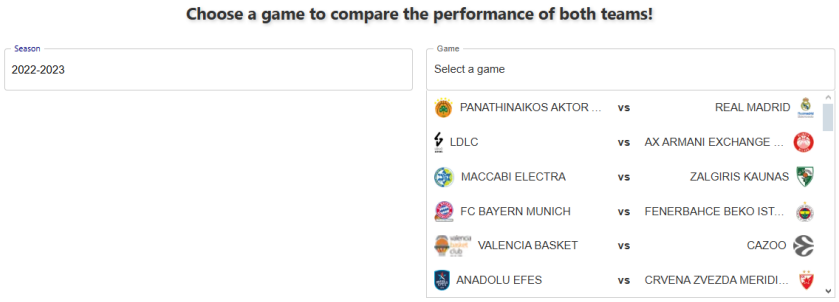


Figure 13: Games dropdown list filtering according to the season

- **get_win_loss_history Function:**

  The `get_win_loss_history` function retrieves a comprehensive win-loss history between two specified teams by joining (`JOIN`) data from the `CUP_HEADER` and `CUP_COMPARISON` or `LIG_HEADER` and `LIG_COMPARISON` tables. This function plays a critical role in enabling users to analyze past matchups between teams, highlighting their performance over time.

  - **Fetching Team Abbreviations:**

    The function begins by fetching the abbreviations of the two teams provided as input. This is achieved through a query on the `teams` table, where the `full_name` of the teams is used to retrieve their corresponding `abbreviation`. If the abbreviations for one or both teams are not found, the function gracefully handles the error and returns an appropriate response, ensuring robustness.

  - **Mapping Abbreviations to Team Names:**

    After retrieving the abbreviations, a mapping is created between each team's full name and its abbreviation. This mapping facilitates the identification of games involving the two teams in subsequent queries.

  - **Constructing Game Identifiers:**

    Using the team abbreviations, two possible game strings (`game1` and `game2`) are generated to account for the order in which the teams might appear in the `game` column of the `CUP_COMPARISON` or `LIG_COMPARISON` table. For example, if the teams are `ABC` and `XYZ`, the generated strings would be `ABC-XYZ` and `XYZ-ABC`.

  - **Joining Tables to Fetch Historical Data:**

    The function performs a `JOIN` operation between the `CUP_COMPARISON` and `CUP_HEADER` or `LIG_COMPARISON` and `LIG_HEADER` tables to retrieve game details. This includes:

* `game_id`, `game`, `date_of_game`, and `time_of_game`.
* Scores for both teams (`score_a` and `score_b`).
* The winner of the match.

The query applies a `WHERE` clause to filter games matching `game1` or `game2`, and results are sorted by the `date_of_game` in ascending order.

– **Win-Loss and Draw Calculation:**

The function iterates over the retrieved games to calculate the number of wins for each team and the number of draws. The logic is as follows:

* If `score_a` is greater than `score_b`, the winner is determined based on whether `game` starts with `abbr_team1` or `abbr_team2`.
* If `score_b` is greater than `score_a`, the same logic applies but in reverse.
* If the scores are equal, the game is classified as a draw.

The total number of games is also calculated as the length of the result set.

– **Error Handling and Resource Management:**

The function includes error handling to catch database-related exceptions. Additionally, it ensures that the database connection and cursor are properly closed in the `finally` block, preventing resource leaks.

– **Output:**

The function returns a dictionary containing:

* `total_games`: The total number of games between the two teams.
* `team1_wins` and `team2_wins`: The win counts for each team.
* `draws`: The number of draws.
* `history`: The detailed game records, including dates, times, scores, and winners.

How It Works in Practice: When users select two teams in the frontend, the function is invoked through an API call. The returned data is displayed in a sortable table, allowing users to analyze historical matchups and sort games by date or other attributes. This feature enhances user experience by providing clear insights into team performance over time. And the output (the win-loss history) is shown in Figure 14.

Figure 14: Win-Loss History output

This was 2 of the most important functions used to achieve compatibility for the **Comparison User View**, this all worked together to achieve the final outcome we can see (part of it) in Figure 12.
There were also other functions like **get_cup/lig_comparison** that worked on fetching the data, but for the space constraint, we will explain a similar function in the header user view: **get_cup/lig_header**

6.3.4 Header User View

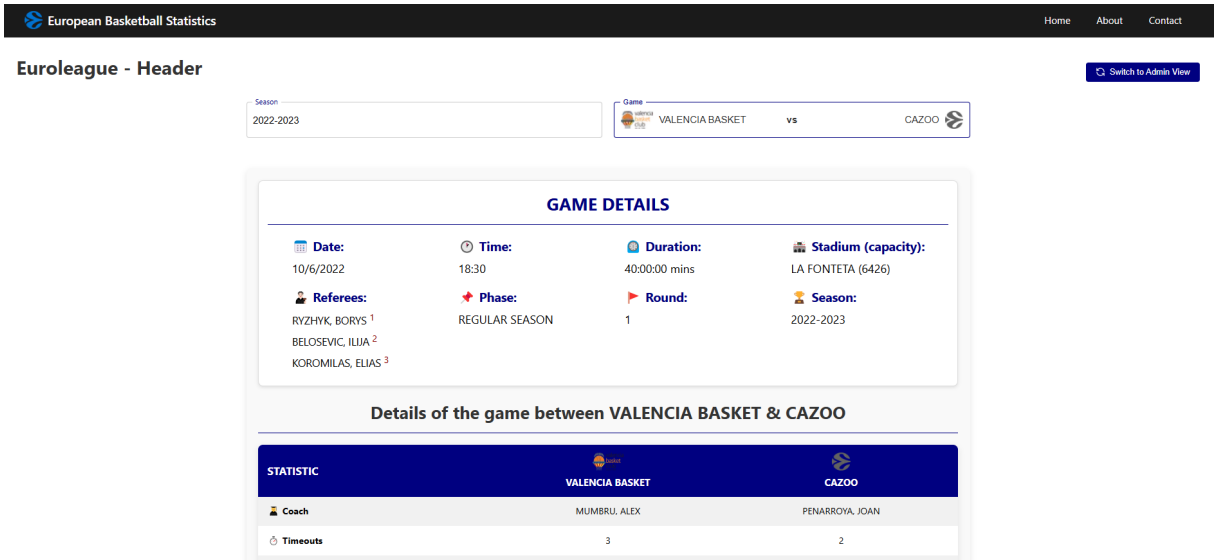


Figure 15: Header User View

The **Header User View** provides users with detailed information about a specific match, including game details such as scores, fouls, timeouts, stadium information, and referees. Users can filter matches by season and select a specific match to analyze. Additionally, the system dynamically displays a winner section with visual effects, such as

fireworks, based on the match result. Below are the key queries used to power this user view:

- **get_distinct_games_with_like Function:**

  This function is responsible for retrieving a list of distinct matches from the `CUP_HEADER` or `LIG_HEADER`table. It populates a dropdown menu in the user interface, allowing users to select a match for further analysis. This function works exactly like the one in the Comparison User View, below are the key functionalities of this function:

  - **Filtering with the `LIKE` Clause:**

    The function takes a `like_pattern` parameter to filter matches by season. For example, selecting the 2023 season sets the `like_pattern` as `U2023%` or `E2023%`, depending on whether the tournament is *EuroLeague* or *EuroCup*.

  - **Retrieving Distinct Matches:**

    By using the SQL `SELECT DISTINCT` statement, the function ensures that only unique matches are retrieved, eliminating redundancy and improving the user experience.

  - **Dynamic Column Selection:**

    The function supports dynamic column selection via the `columns` parameter. If specified, the query retrieves only the requested columns, such as `game` and `game_id`. By default, it selects the `game` column.

  - **Mapping Results:**

    The function maps the results to dictionaries or raw values, depending on the selected columns:

    * If multiple columns are selected, the results are returned as a list of dictionaries.
    * If only the default `game` column is selected, the results are returned as a simple list.

  - **Error Handling and Resource Management:**

    Robust error handling ensures stability by catching database-related errors and properly closing connections and cursors in the `finally` block.

  How It Works in Practice: When a season is selected in the frontend, the `like_pattern` is passed to this function via an API call. The function filters matches by the season prefix (e.g., `U2023`), retrieves distinct matches, and sends them to the frontend for display in a dropdown menu. The output is shown in Figure 16.

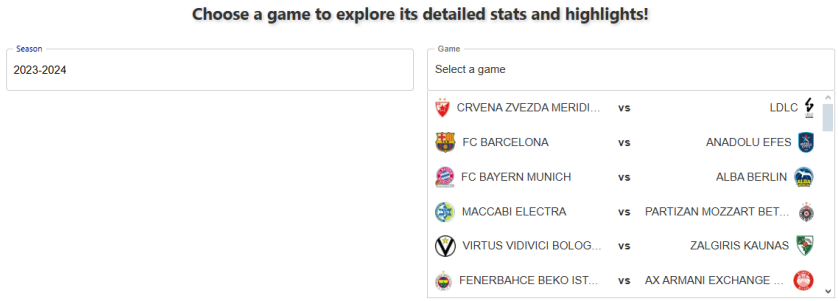


Figure 16: Games dropdown list filtering according to the season

- **get_(cup/lig)_header Function:**

  The `get_(cup/lig)_header` function retrieves detailed information about a specific match from the `CUP_HEADER` or `LIG_HEADER` table. This data is displayed in the user interface, providing insights into the game's critical aspects.

  - **Input Parameter:**

    The function takes a `game_id` as input to identify the specific match.

  - **Query Execution:**

    It executes a `SELECT *` query to retrieve all columns from the `HEADER` table where the `game_id` matches the input.

  - **Mapping Results:**

    The query result is mapped to a `Header` object, containing attributes such as:

    * `game_id`, `game`, `date_of_game`, `time_of_game`.
    * Team-specific details, including `coach`, `timeouts`, `fouls`, and `scores`.
    * Quarter-by-quarter scores and extra-time details.
    * Stadium information (`stadium`, `capacity`).
    * Referee details (`referee_1`, `referee_2`, `referee_3`).
    * The match winner (`winner`).

  - **Error Handling:**

    The function includes error handling for cases where the `game_id` does not exist, ensuring the application does not crash.

  - **Output:**

    The retrieved data is formatted and returned to the frontend for display. Users can view game-specific statistics and additional information, such as the winning team, referees, and stadium details.

How It Works in Practice: When a user selects a match in the frontend, the `game_id` is sent to this function through an API call. The function fetches detailed match information, which is displayed in a structured format on the user interface. The output (game details) is shown in Figure 15.

These two functions, `get_distinct_games_with_like` and `get_cup_header`, are the primary queries powering the **Header User View**, ensuring seamless and efficient data retrieval for user analysis.

6.3.5 Play By Play User View

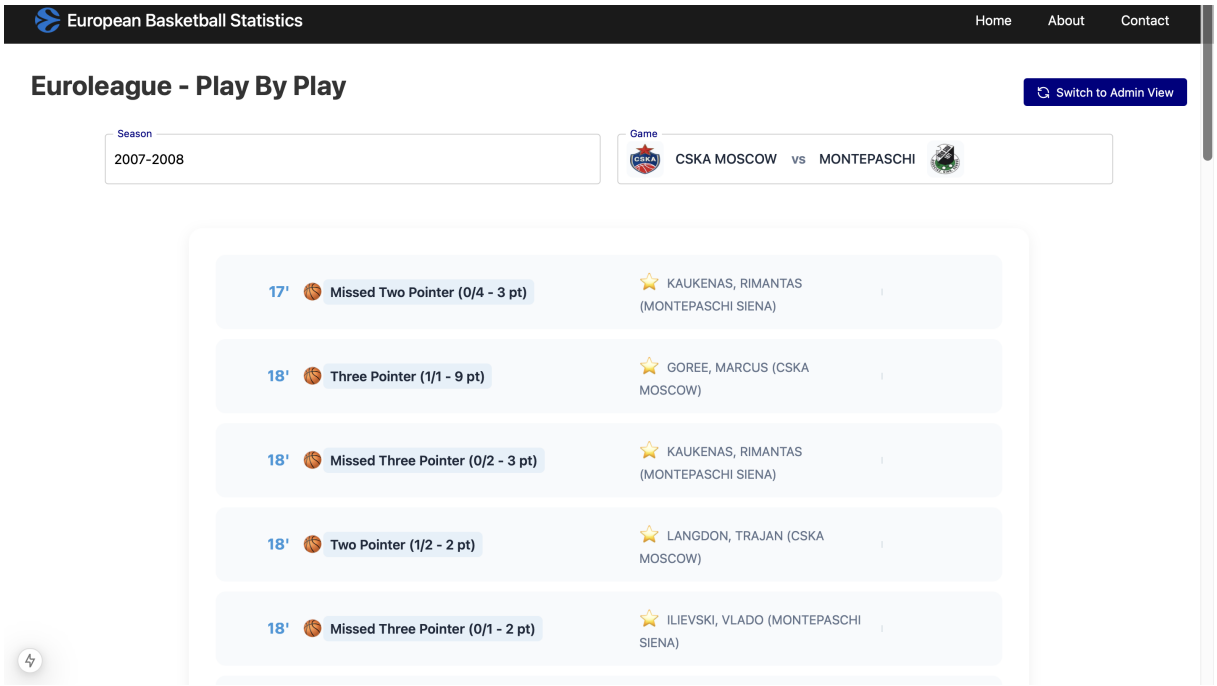


Figure 17: Play By Play User View Page

The **Play By Play User View** is designed for displaying detailed play-by-play basketball data, as shown in Figure 17. Its primary purpose is to present users with comprehensive statistics of a selected game, highlighting crucial game events such as the minute, player involved, and play description (`play_info`). These details are vital for understanding the flow and critical moments of the game, offering a granular view of basketball dynamics. This feature offers an interactive display of basketball events, allowing users to follow key game moments in a structured format. For each event, it provides details about the game minute, play description, player name, and the associated team, along with the current scores of the competing teams. The inclusion of these details helps users analyze the dynamics and turning points of the game. Pagination controls further enhance usability, enabling efficient navigation through large datasets with adjustable rows

per page, such as **25, 50, or 100** rows.

The component offers an interactive display for basketball events. For each play-by-play event, it shows the game minute, play description, player name, and the associated team, alongside the current scores of the competing teams. To enhance user engagement, the component emphasizes leading scores by applying dynamic **CSS styling**. **Pagination** controls are available, enabling users to navigate through large datasets efficiently. These controls include *"Previous"* and *"Next"* buttons as well as adjustable rows per page, such as 25, 50, or 100 rows. **Error handling** is robust, displaying messages if **API** calls fail and providing retry options. Additionally, the loading skeleton ensures a seamless experience by showing placeholder content during data fetching.

State management in the component is handled using **React's** `useState` and `useEffect` hooks. These hooks track key variables such as the selected season and game, fetched data like **games** and **play-by-play** details, and states for loading and errors. Variables like `offset` and `rowsPerPage` ensure accurate pagination. `teamInfo` and `currentGameTeams` store metadata about teams and current selections.

Play By Play user view integrates multiple API endpoints to fetch data dynamically. For example, the `/year_distinct_games` endpoint fetches the list of games for a selected season, while the `/with_year_like` endpoint retrieves play-by-play data with specified filters.

The component employs advanced dynamic rendering techniques to present content seamlessly. Dropdowns for selecting seasons and games are implemented with smooth transitions, offering a responsive user interface. Play-by-play rows are designed with a grid layout, aligning details like the minute, player info, play description, and points consistently. Dynamic hover effects enhance interactivity. Team logos and names are fetched and displayed dynamically, enriching the user experience. Pagination controls are styled buttons and dropdown selectors that provide easy navigation. Moreover, the component adapts to various screen sizes through responsive CSS breakpoints.

The styling is modular, utilizing CSS classes for scoped and organized design. Flexbox layouts ensure proper alignment and spacing, while grid-based designs align the play-by-play details effectively. Subtle animations, such as `fadeIn` keyframes, and hover effects on dropdowns and rows enhance usability. Utility classes like `.leadingPoints` visually emphasize critical information, such as leading scores, using green highlights. Key CSS components include the `.containerWrapper`, which centralizes content with responsive adjustments; `.playRow`, which structures play-by-play details in a grid layout; and `.paginationControls`, which consist of styled buttons and dropdowns for navigation. Media queries are used extensively to ensure responsiveness and usability across various

devices, adapting layouts and font sizes dynamically.

The workflow begins with season selection. Users select a season from the dropdown menu, prompting the component to fetch and display a list of available games. Upon selecting a specific game, team logos and names are displayed dynamically, and play-by-play data is fetched for that game. The data is displayed in a structured, minute-by-minute view, allowing users to review detailed game events. The ability to highlight key information, such as the `minute`, `player`, and `play_info`, ensures that users can easily understand the sequence of events and their importance. Pagination controls enable efficient navigation through larger datasets. In case of errors during data fetching, users are notified with an option to retry, ensuring a smooth experience.

6.3.6 Box Score User View

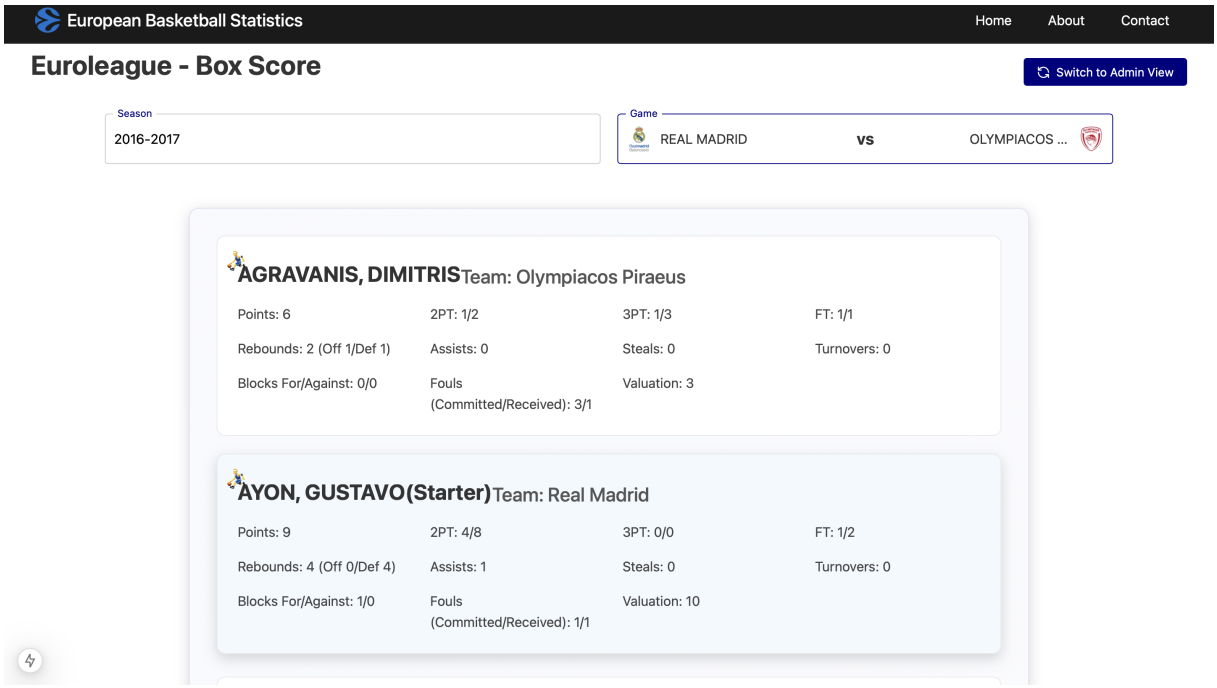


Figure 18: Box Score User View Page

The **Box Score User View** is designed to display detailed box score data for basketball games, as can be seen from Figure 18. Its purpose is to provide a comprehensive and granular view of individual player statistics for specific games. In sports, the concept of a *"box score"* is widely used to summarize a player's performance during a game, and in basketball, it holds particular significance due to the wide range of statistics tracked for each player. The box score includes critical metrics such as **Points**, **2PT** (two-point field goals), **3PT** (three-point field goals), **FT** (free throws), **Rebounds** (offensive, defensive, and total), **Assists**, **Steals**, **Turnovers**, **Blocks** (For/Against), **Fouls** (Committed/Re-

ceived), and **Valuation**. These statistics offer a thorough understanding of each player's contributions and performance during a game.

The Box Score User View allows users to select seasons and games, fetch **player-specific statistics**, and view detailed contributions of each player to the game. The dynamically rendered box score prominently displays whether a player is a *starter* or a *substitute*, along with their team information. The interface is interactive, utilizing hover effects and CSS styling to ensure clarity and enhance the user experience. Additionally, it supports pagination for efficient navigation through large datasets and incorporates robust error handling to provide retry options when API calls fail.

The query used to fetch detailed box score data is as follows:

```
SELECT DISTINCT {selected_columns}
FROM LIG_BOX_SCORE bs
LEFT JOIN LIG_PLAY_BY_PLAY pbp
ON bs.game_player_id = pbp.game_player_id
{where_clause}
{order_clause}
```

This query is designed to retrieve comprehensive player statistics for a specific game. The `selected_columns` variable determines the fields to be retrieved, such as player name, points, rebounds, and valuation. A critical operation in this query is the `LEFT JOIN` between the `LIG_BOX_SCORE` table and the `LIG_PLAY_BY_PLAY` table. This join operation connects the two tables through the `game_player_id` field, ensuring that relevant team information from the `LIG_PLAY_BY_PLAY` table is included in the result set of the box score data. The purpose of this **join** is to enrich the box score data with team-specific details, such as the name of the team a player belongs to. Since the `LIG_BOX_SCORE` table primarily contains player-specific performance metrics (e.g., points, rebounds, and assists), the `LEFT JOIN` enables the addition of contextual information from the `LIG_PLAY_BY_PLAY` table. This integration allows for a more complete representation of each player's contribution in relation to their team. This `LEFT JOIN` operation was implemented for the `LIG_BOX_SCORE` table but is also used in a similar way for the `CUP_BOX_SCORE` table. By replicating the join logic, both `lig` (league) and `cup` datasets are supported, ensuring consistency in how player statistics and team information are combined and displayed.

The `BoxScoreUserView` component manages its state using React's `useState` and `useEffect` hooks. Key state variables include `selectedSeason` and `selectedGame` for tracking user selections, `boxScores` for storing fetched player statistics, and `loading` and `error` for **handling data fetching states**. The `offset` and `rowsPerPage` variables manage pagination effectively. The `fetchBoxScores` function dynamically calls the **API**

to retrieve box score data using the SQL query, and deduplication of player data ensures accurate results. Additionally, the `fetchTeamInfo` function retrieves team metadata to enhance the display.

The workflow for using the `BoxScoreUserView` component begins with users selecting a season from a dropdown menu, which prompts the component to fetch and display a list of available games. After selecting a specific game, the component fetches box score data and displays individual player statistics in a structured layout. The inclusion of critical metrics like Points, 2PT, 3PT, FT, Rebounds, Assists, Steals, etc. ensures that users can fully analyze each player's performance. Pagination controls allow users to navigate through larger datasets efficiently, while error messages guide users in case of issues, ensuring a smooth and seamless user experience.

6.3.7 Player User View

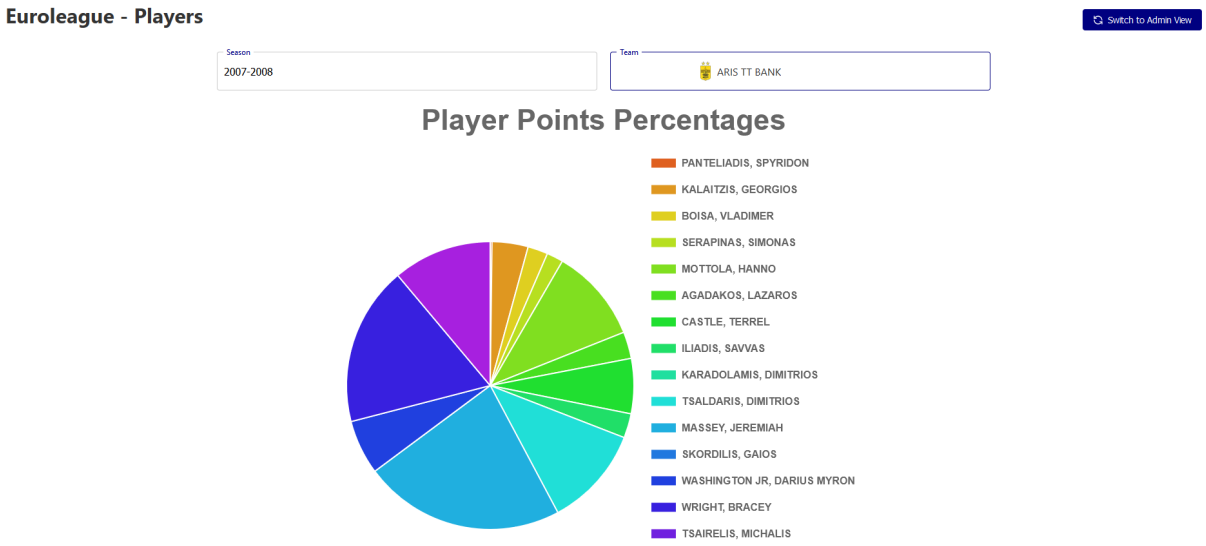


Figure 19: Player User View

The Player User View provides an overview of how Players performed during a specific season. Users first select the season from a dropdown menu, then select a team from that season, and the system displays a pie chart that shows the percentages of points of that team. This pie chart helps the user identify the team players' performance. On top of that, the user can remove some players from the pie chart by clicking on their names on the side, better highlighting the performance of the players remaining in the pie chart.

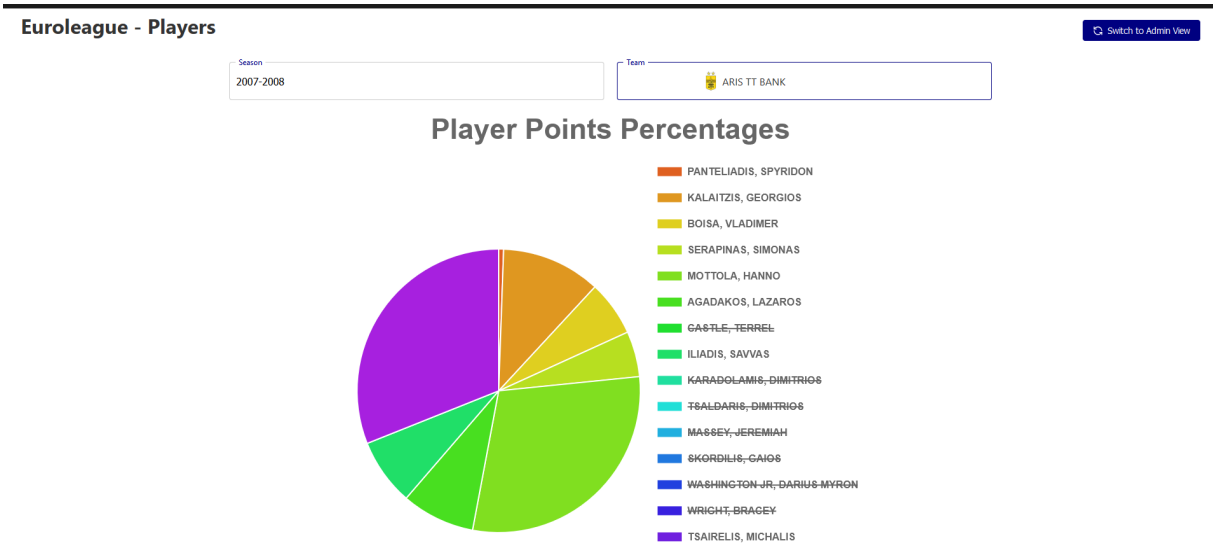


Figure 20: Removing player from the pie chart functionality

- **get_distinct_teams_by_year Function:**

  This function retrieves a list of distinct team identifiers for a given season. The `year` parameter specifies the season, prefixed with an identifier (e.g., `E2007`). Using this parameter, the function filters data by applying a `LIKE` clause to the `season_team_id` column in the `LIG_PLAYERS` table.

  The `SUBSTRING_INDEX` function is used to extract the portion of the `season_team_id` after the last underscore (_), which represents the team identifier (e.g., `PAM`). The result is a flat list of unique team identifiers for the specified season.

  Why it is needed: This function is essential for populating dropdown menus with team options for the selected season. It ensures that only relevant teams are displayed, streamlining the process for users to analyze specific teams.

- **get_players_point_percentage_by_team_and_season Function:**

  The `get_players_point_percentage_by_team_and_season` function calculates the percentage of points each player contributed to their team's total points during a specific season.

  This function joins the `LIG_PLAYERS` and `LIG_TEAMS` tables on the `season_team_id` column to retrieve both player and team statistics. It selects:

  - `player_points`: The total points scored by each player.
  - `player_name`: The player's name.
  - `team_name`: The team's name (extracted from `season_team_id`).
  - `team_points`: The total points scored by the team.

– `point_percentage`: The player's points as a fraction of the team's points.

The results are returned as a list of dictionaries, with each entry containing the statistics for a single player.

Why it is needed: This function provides detailed insights into individual player contributions, helping users identify key players and understand their roles within the team. By presenting point percentages, the data highlights the relative importance of each player's performance.

The Player User View enables users to explore individual performances within the context of their team's overall results. By combining these functions, the system ensures accurate and meaningful data retrieval, allowing users to analyze player contributions effectively.

This approach ensures users can quickly access detailed and meaningful player statistics for any team and season.

The Player User View provides detailed insights into individual player performances during a specific season. Users can filter data by season and team, and the system displays statistics such as player points, team contributions, and performance percentages. This view allows users to analyze player contributions within the context of their teams, offering a clear understanding of their impact.

7 DISCUSSION AND CHALLENGES

- **Dataset Selection Process:** Our initial plan was to work on a dataset related to football, as this topic was of interest to us. However, due to the strict constraints on the relations required for the project, we had to explore multiple tables across various topics. Finding a suitable dataset that met all the necessary relational conditions proved to be a challenging task. Moreover, since our selected dataset denied from the very beginning, most of the high-quality datasets had already been taken by other groups.

  To overcome these difficulties, we consulted the assistant several times to seek guidance and clarify our understanding of the requirements. After considerable effort and exploration, we finally identified a dataset during the final days before submission that could meet the necessary conditions with some modifications. These modifications and the steps we followed to prepare the dataset are explained in detail in section 5.

- **Integration of Backend and Frontend** Integrating the backend API with the React frontend presented challenges in maintaining consistency in data representation

and avoiding mismatches in data types and structures. During development, mismatched data types between the API responses and the frontend caused unexpected behavior in certain user views. This challenge was addressed by standardizing the API output format and implementing robust error handling on both ends.

- **Database Optimization for Performance** The large size of the dataset and complex queries caused delays in data retrieval and required optimization. Performance issues arose when handling large datasets, especially in views requiring frequent filtering and sorting. To address this, indexes were added to frequently queried columns, and queries were optimized using efficient joins and pagination techniques. These improvements significantly reduced response times and enhanced user experience. An example of that is the **Admin View** implementation in subsection 6.2.

- **Data Cleaning and Preprocessing** The raw dataset contained inconsistencies such as missing values, duplicate records, and corrupted data. Preprocessing the dataset was a critical step to ensure data quality. Challenges included handling missing values, removing duplicates, and resolving discrepancies in player statistics and team records. Custom Python scripts were written to automate this process, ensuring consistency and reliability across all tables. And this can been seen in section 5.

- **Error Handling and Debugging** Debugging errors in complex SQL queries and ensuring seamless API functionality were time-consuming. During the project, debugging SQL queries and API callings involving multiple joins and filters proved challenging, as minor errors led to unexpected results. We addressed this by using tools like **POSTMAN** [2] and systematically breaking down complex queries into simpler parts for testing. Additionally, comprehensive logging was implemented in the backend to quickly identify and resolve API-related issues. The final errorless result of this is shown in the functionalities of both the **Admin View** and **User Views** parts in subsection 6.2 and subsection 6.3.

- **Time Management and Collaboration** Balancing individual responsibilities with collaborative tasks was a key challenge given the scope of the project. Managing individual contributions while maintaining team coordination was challenging, especially when integrating different components. Weekly meetings and a shared task-tracking system ensured progress alignment and timely completion of tasks. Tools like GitHub were instrumental in managing code collaboration effectively. All this can be obviously seen in section 4

- **Dynamic User Interface Development** Designing and implementing an intuitive yet feature-rich user interface required balancing aesthetics and functionality. Creating a dynamic and visually appealing user interface involved significant effort in both design and implementation. The use of React allowed us to build reusable components, but challenges arose in synchronizing frontend states with API responses. Iterative testing and user feedback helped refine the UI to meet user expectations. All of which can be touched through all the user views in our web-app.

- **Dataset Limitations and Adaptations** The dataset lacked some desired information, which required creative solutions to extract meaningful insights. Certain key statistics, such as team logos and full names, were missing from the dataset. To address this, we sourced external data and created additional scripts to merge and preprocess the dataset. This enhanced the overall user experience by providing visually enriched data. An example of that is shown in the teams table creation in subsection 5.4

- **Learning Curve for New Technologies** At the beginning, we had a limited experience with tools like React, Flask, and especially MySQL and database systems. Attending the class lectures was very helpful, and at the same time for the other used technologies, we have conducted a lot of online search and knowledge-sharing sessions until we mastered them to get to the current output. This collaborative learning approach ensured all members could contribute effectively to their assigned tasks.

- **Handling Complex Relationships in the Database** Establishing relationships between tables and maintaining referential integrity required significant effort. Establishing primary and foreign key relationships in the database was a complex task due to the interdependence of tables. Challenges included resolving orphaned records and ensuring data consistency during updates. This was achieved by carefully designing the schema and implementing robust constraints in the database as you can see from our ER Diagram in section 3.

- **Testing and Validation** Ensuring all components worked seamlessly together required extensive testing and validation. Testing the system as a whole was a crucial yet challenging task. Bugs were identified during integration testing, particularly in edge cases involving rare scenarios like missing data or invalid inputs. Comprehensive unit and integration tests were implemented to ensure the reliability of all components.

8 CONCLUSION

In conclusion, the European Basketball Statistics Project successfully reached its goal of creating an interactive platform for analyzing and managing basketball data. By carefully preparing the dataset and using technologies like Flask, React, and MySQL, the project delivered a system that can handle complex relationships and large amounts of data. The team faced challenges, such as fixing issues in the dataset, integrating different parts of the project, and learning new tools, but these were solved through teamwork, research, and problem-solving. The platform provides a user-friendly way to explore basketball data and supports better decision-making for both admin and users.

REFERENCES

[1] Lucidchart. Lucidchart: Online diagram software and visual solution, 2025. Accessed: 2024-10-11.

[2] Inc. Postman. Postman: Api platform for building and using apis, 2025. Accessed: 2024-11-06.

[3] Octopus.do. Octopus.do: Visual sitemap builder and website planning tool, 2025. Accessed: 2025-01-03.