Football Performance and Analytics System with MongoDB, MySQL, and Python

Alexandre Baptista
fc64506@alunos.fc.ul.pt

Matei-Alexandru Lupașcu
fc64471@alunos.fc.ul.pt

Lloyd DSilva
fc64858@alunos.fc.ul.pt

Vram Davtyan
fc64691@alunos.fc.ul.pt

Phase 1: Project Overview

Project Description

The Football Performance and Analytics System is designed to track and analyze various aspects of football matches, player performances, and team statistics. This system utilizes a combination of MongoDB, MySQL, and Python to extract valuable insights from football data. These insights help teams, analysts, and enthusiasts make data-driven decisions related to player performances, match outcomes, and team rankings.

Technical Implementation

- **Databases**: MongoDB (NoSQL) and MySQL (SQL) are used to store and manage football data.
- **Programming**: Python is used to query the databases, process the results, and visualize data, providing an intuitive interface for interacting with the backend.

Code Maintenance in GitHub

The project code is maintained on GitHub with two branches:

- **FirstPhase**: Contains the initial phase of development.
- **SecondPhase**: Includes optimizations and updates from Phase 2.

Key Features and Data Components

1. Player Statistics

- Tracks individual player performances such as goals scored (by head or foot), shot outcomes, and match details.
- **Columns**: Player, Outcome, Distance, Body Part, match_id, etc.

2. Match Details

- Contains information about individual football matches including date, league, attendance, and scores for home and away teams.
- **Columns**: Date, league, Round, Day, Attendance, home_id, away_id, score_away, score_home.
- **Example Query**: "How many matches had attendance greater than 12,000?"

3. Club Information

- Tracks statistics for football clubs such as league position, matches played, wins, losses, goals scored, and points earned.
- **Columns**: id, Pos, Matches, club_id, MP, year, name.
- **Example Query**: "Rank teams based on their average attendance at home games."

4. Player Rankings

- **Columns**: id, name, league, etc.
- **Example Query**: "List shots where a goal was scored outside the 16m box, sorted by descending order."

Data Cleaning & Columns Used in Queries

all_players.csv

- **Columns used**: `id`
  - **Data format**: `/en/players/player_id(*)/player_name`
  - As we used only `player_id` and `player_name` as separate data within the queries, in the preprocessing and data cleaning part we split the data by “/”.

matches.csv

- **Columns used**: `Attendance`, `home_id(**)`
  - **Data format**: numeric, string

- `Attendance` consists of the number of people that attended each match. `home_id` represents the id of the team that played home in the specific match.

shots.csv

- **Columns used**: `Outcome`, `Distance`, `Player (*)`, `Body Part`
  - **Data format**: string, numeric, string, string
  - `Outcome` contains all variations of shots' outcomes, while the values can be Goal, Saved, Off Target, Woodwork, Blocked, or Saved off Target. `Distance` marks the distance (in meters) from which every shot was kicked. `Body Part` values indicate which body part was used for each shot in the dataset.

teams.csv

- **Columns used**: `id(**)`, `name`
  - **Data format**: string, string
  - `id` describes the team identifier, while `name` lists the names of each team in the dataset.

Comparative Analysis of SQL and NoSQL on Football Dataset Queries

Upon analyzing the dataset, several questions naturally arose, which we addressed by querying both relational (SQL) and NoSQL databases. The answers to these questions, derived using both database models, are detailed below.

1.1 Number of Goals Scored with the Head

The first question posed was: How many goals were scored using the head?

To answer this, we utilized the `shots` dataset, which included two key attributes: `Outcome` and `Body Part`. The `Outcome` attribute included categories such as Goal, Saved, and Off-Target, while `Body Part` categorized attempts by Head, Left Foot, and Right Foot. A filter was applied using an AND condition to select records where the `Outcome` was Goal and the `Body Part` was Head.

- **Results**: Out of a total of 24,570 goals, 3,963 were scored using the head. The remaining goals were achieved using either the left foot, right foot, or other unspecified body parts.

1.2 Number of Matches with Attendance Greater than 12,000

The second question addressed was: How many matches had attendance exceeding 12,000?

This query was executed on the `matches` dataset, which contained attendance records. The condition applied used the > (greater than) operator to filter matches with an attendance count exceeding 12,000.

- **Results**: Out of 14,148 matches, 9,166 matches had attendance surpassing 12,000 attendees.

Summary of Simple Queries

The analysis highlights the efficiency and accuracy of both SQL and NoSQL databases in answering specific queries. While the results were identical for both database types, there were slight differences in execution times. This analysis depicts the suitability of SQL and NoSQL databases for handling structured and semi-structured datasets, respectively, depending on the nature of the query and dataset.

1.3 Goals Scored Outside the 16m Box: Goals by Distance

This analysis aimed to determine the total goals scored by players, distinguishing between those scored from distances greater than 16 meters and those within 16 meters. The query was executed in both MongoDB and SQL databases, and the query is summarized below.

MongoDB Implementation

- **Aggregation Pipeline**:
  - **Filtering**: The `Outcome` field was matched to retain only records where `Outcome` was "Goal."
  - **Grouping**: Data was grouped by `Player` (ID of the players from the `shots` dataset) and `Outcome` (shots dataset) to calculate:
    - `gt_16`: Goals scored from distances > 16 meters using `$sum` and `$cond` operators.
    - `total_goals`: Total goals scored by each player.

        - `lt_16`: Goals scored from distances ≤ 16 meters (calculated as the difference between total_goals and gt_16).
    - **Sorting**: Results were sorted in descending order of total_goals.
    - **Player Name Mapping**: Player IDs from the `shots_collection` were cross-referenced with the `players_collection` to retrieve player names.
    - **Lookup**: Perform a `$lookup` to join the `shots` collection with the `players` collection based on the player ID, adding player names.
    - **Unwind**: Consolidate the joined results to ensure one document per player.
    - **Projection**: Exclude the `_id` field and retain relevant fields like Player_Name, total_goals, gt_16, and lt_16.
    - **Sorting**: Order results by `total_goals` in descending order.

SQL Implementation

- **Query Structure**:
    - **Subquery**: A Common Table Expression (CTE) `unique_players` was created to retrieve distinct player IDs and names from the `all_players` table.
    - **Main Query**: Data from the `shots` table was joined with the `unique_players` CTE on the player ID, with the following calculations:
        - `goals_gt_16`: Count of goals scored from distances > 16 meters using a `CASE` statement.
        - `goals_lt_16`: Count of goals scored from distances ≤ 16 meters using a `CASE` statement.
        - `total_goals`: Total number of goals per player.
    - **Filtering**: Only records where `Outcome` was "Goal" were included.
    - **Sorting**: Results were ordered by `total_goals` in descending order.

Both approaches showed identical results, with slight differences in execution time. MongoDB's pipeline excels in handling document-oriented data, while SQL's structured approach leverages relational schema for efficient joins and aggregations.

2.2 Rank Teams Based on Average Attendance at Home Games

The query aims to rank teams based on the average attendance at their home games. By calculating the average attendance and the total number of home matches, this analysis provides insights into the teams' popularity and fan engagement at their home stadiums. The data is then grouped by each team's home games, allowing for a ranking of teams based on the average attendance per match. The query is implemented in both MongoDB and SQL, each using its respective querying capabilities to achieve the same result: a list of teams ordered by their average home match attendance.

MongoDB Implementation

- **Aggregation Pipeline**:
    - **Grouping**: Data is grouped by `home_id` to calculate:
        - `average_attendance`: Average attendance using `$avg`.
        - `matches_count`: Total number of home matches using `$sum`.
    - **Lookup**: A `$lookup` is performed to join the `teams` collection based on `home_id` to get team names.
    - **Unwind**: Ensures that each home team record appears only once after the join.
    - **Projection**: Selects relevant fields like `home_id`, `average_attendance`, `matches_count`, and `team_name`.
    - **Sorting**: Results are ordered by `average_attendance` in descending order.

SQL Implementation

- **Query Structure**:
    - **Join**: Data is retrieved from the `matches` table, joined with the `teams` table on `home_id` to get team names.
    - **Aggregation**: Uses `AVG()` to calculate `average_attendance` and `COUNT()` for `matches_count`.
    - **Grouping**: Data is grouped by `home_id` and `team_name`.
    - **Sorting**: The results are sorted by `average_attendance` in descending order.

Both implementations return identical results, with minor differences in the underlying query structure. MongoDB's aggregation pipeline efficiently handles document-based data, while SQL's relational approach excels in structured data handling through joins and groupings.

Query Comparison: Using `$lookup` vs Not Using `$lookup`

Context of Comparison

The comparison was conducted for the query "Goals scored outside the 16m Box: Goals by Distance", where the two approaches (with and without `$lookup`) were evaluated based on execution time and efficiency. Additionally, we analyzed a different query, "Rank teams based on average attendance at home games", using the same approaches. In the second query, the execution times for both methods were nearly identical, unlike the significant differences observed in the first query.

Query 1: "Goals Scored Outside the 16m Box: Goals by Distance"

Without `$lookup`

- **Execution Time**: Faster.
- **Process**:
  - Aggregates data on goals by distance in one query.
  - Runs a second query to fetch player names from the players collection.
  - Merges player IDs with names programmatically in Python.
- **Why Faster**:
  - Avoids the computational overhead of performing a `$lookup` join in MongoDB.
  - The second query fetches only specific fields (id and name), which is less intensive than a join operation during aggregation.
- **Why Less Efficient**:
  - Increases code complexity with additional steps and external mapping.
  - The multi-query approach may become less manageable and prone to errors as datasets grow larger.

With `$lookup`

- **Execution Time**: Slower.
- **Process**:
  - Uses `$lookup` to join the `shots` and `players` collections directly within the aggregation pipeline.
  - Produces a single consolidated output that includes player names.
- **Why Slower**:
  - `$lookup` introduces significant overhead by performing a join operation during the aggregation pipeline, which is resource-intensive for large collections.
- **Why More Efficient**:
  - Reduces code complexity by keeping the entire operation within MongoDB.
  - Ensures consistency and scalability, avoiding manual merging or handling mismatched data.

Query 2: "Rank Teams Based on Average Attendance at Home Games"

Key Observation

For this query, the execution times for both with and without `$lookup` were nearly identical.

- Both methods performed well because the underlying data structure and access patterns likely required less computational effort for the `$lookup` join.

Conclusion

- For the query "Goals scored outside the 16m Box: Goals by Distance", the **without `$lookup`** method executed faster but was less efficient in terms of code complexity. The **with `$lookup`** approach took longer but offered better design, scalability, and maintainability.
- For the query "Rank teams based on average attendance at home games", the execution time was almost equal for both methods. This suggests that `$lookup` can perform efficiently when the collections are optimized for joins or involve fewer complex relationships.
- **Key Trade-off**: While `$lookup` adds computational overhead in some cases, it simplifies the code, reduces potential errors, and is more suitable for scalable solutions. For smaller

Known Errors

Data Type Mismatch and Large Number of Columns

- **Problem**: Difficulty in loading datasets into MongoDB and SQL due to inconsistent data types and too many columns, which impacted performance.
- **Solution**:
  - **Data Transformation**: Standardizing formats.
  - **Schema Mapping**: Mapping data to the correct schema.
  - **Column Selection**: Removing unnecessary

Execution Time Comparison

Phase 1

Query	MySQL Execution Time (s)	MongoDB Execution Time (s)
Q1.1	0.1290	0.1630
Q1.2	0.0330	0.0370
Q2.1	0.1870	0.3361
Q2.2	0.0280	0.0430

Phase 2 With Lookup in MongoDB

Query	MySQL (Without Index)	MySQL (With Index)	MongoDB (Without Index)	MongoDB (With Index)
Q1.1	0.1380	1.0844	0.1470	0.6497
Q1.2	0.0480	0.0979	0.0100	0.0740
Q2.1	302.1917	1.9367	35.2755	35.6576
Q2.2	0.0270	0.2027	0.2791	0.3297

Speedup Comparisons (With Index)

In the MongoDB queries, the execution times vary based on the indexed columns. The most noticeable speedups occur when the `PlayerID` column is used because it contains unique values, making queries faster.

- Queries using `PlayerID` (e.g., `{'Player': 'PlayerID'}`) show significantly lower execution times due to the unique nature of the column.
  - Example: Query `{'Player': 'PlayerID'}` took **0.6003 seconds**.
- Queries with combinations of `Outcome`, `Distance`, and `PlayerID` benefit from these indexes, yielding faster results.
  - Example: Query `{'Outcome': 'Goal', 'Player': 'PlayerID'}` took **0.6160 seconds**.
- The queries involving `Outcome` and `Distance` without `PlayerID` are slower, highlighting the impact of index optimization on queries with more unique fields.
  - Example: Query `{'Outcome': 'Goal'}` took **35.4424 seconds**.

Summary of Times (With Index):

- **Outcome**: 35.4424s
- **PlayerID**: 0.6003s
- **Distance**: 31.3678s
- **Outcome & PlayerID**: 0.6160s
- **Outcome & Distance**: 7.8821s
- **PlayerID & Distance**: 0.6153s
- **Outcome, PlayerID & Distance**: 0.6585s

The queries with `PlayerID` as a key column run faster due to its uniqueness, significantly speeding up performance for those combinations.

Key Point:

Queries involving `PlayerID` (a unique field) execute significantly faster, with execution times around **0.6 seconds**, while queries with non-unique fields take much longer, especially without indexing.

Phase 2: Relational vs. Non-Relational Databases

Relational Data Structures

- **MySQL**: Ideal for structured data with fixed schema, such as player statistics, match details, and team performance.
- **MongoDB**: More suitable for semi-structured or unstructured data like player shots, which can have varying attributes (e.g., outcome, body part, distance).

Comparative Analysis: SQL vs. NoSQL

1. Goals Scored with the Head

- **Query**: How many goals were scored using the head?
- **Execution Time**: Both MySQL and MongoDB returned the same result (3963 goals scored by head), with MySQL executing slightly faster (0.1290s vs. 0.1630s).

2. Matches with Attendance Greater Than 12,000

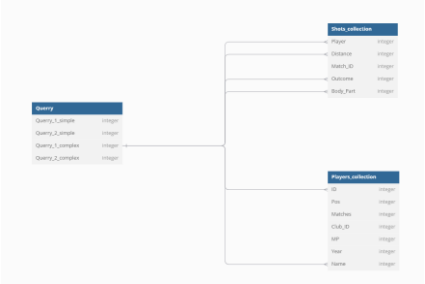
- **Query**: How many matches had attendance exceeding 12,000?
- **Execution Time**: Both databases returned the same result (9166 matches), with MySQL executing slightly faster (0.0330s vs. 0.0370s).

Relational Diagram

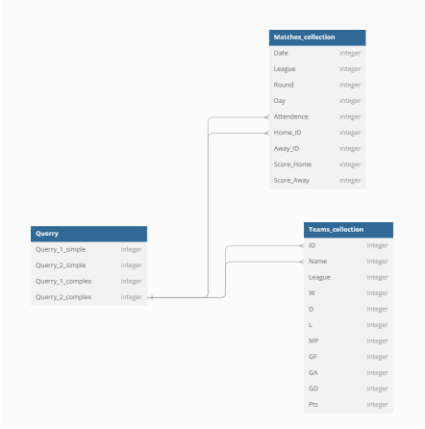
1.1

1.2

2.1



2.2



Contributions

Throughout the project, the team collaborated extensively and contributed in different areas:

1. **Database Design**: Some team members focused on structuring the data in MongoDB and MySQL for efficient querying, ensuring the database schema was optimized for performance.
2. **Query Development**: Team members developed and optimized complex aggregation queries to extract meaningful football statistics.
3. **Python Code**: Two team members wrote Python scripts to handle data processing, query the databases, and generate visual insights.
4. **Version Control and Collaboration**: One team member managed version control using GitHub to ensure smooth collaboration.
5. **Performance Optimization**: All team members focused on optimizing query performance, ensuring the system could handle large datasets efficiently.

The team also held one physical meeting to discuss requirements, set milestones, and participated in several online calls to maintain collaboration and resolve challenges.

Conclusion

The Football Performance and Analytics System successfully combines both relational and non-relational databases to handle structured and semi-structured football data efficiently. The use of MySQL ensures fast querying of structured data like team and match statistics, while MongoDB offers flexibility in handling complex, unstructured player shot data.

Key insights from the project:

- **SQL (MySQL)** is ideal for handling structured data with defined relationships.
- **NoSQL (MongoDB)** provides flexibility for unstructured data, such as shot outcomes, where attributes can vary.

Through advanced data cleaning, query optimization, and the use of both databases, the system is able to offer insightful analytics for football performance, match predictions, and team rankings. This system supports data-driven decisions for analysts, teams, and fans, enhancing the understanding of football performance.