Goal Glory

Bhoomika Madhu Kashyap
bkashyap (50544979)
bkashyap@buffalo.edu

Srinwanti sarkar
ssarkar9 (50541912)
ssarkar9@buffalo.edu

Sriman Reddy Pingili
srimanre (50539997)
srimanre@buffalo.edu

Abstract

Fantasy football has surged in popularity, prompting the need for a comprehensive database to manage player statistics, market values, league data, and more. Current methods using spreadsheets are inefficient. This project aims to develop a centralized database tailored for fantasy football, offering structured storage, efficient querying, and scalability. The database will enhance the management of fantasy teams, player analysis, and match simulations, benefiting both enthusiasts and media outlets in the sports industry.

***Index Terms*—Fantasy football database, player statistics,league data, BCNF, efficient querying, scalability, player analysis, sports industry.**

I. Introduction

Fantasy football has become immensely popular, attracting millions of enthusiasts who engage in managing virtual teams based on real players' performances. To facilitate this, a comprehensive database is being developed to provide detailed player statistics, performance metrics, market values, league data, and national team details. This database will serve as a centralized repository for managing fantasy football teams, analyzing player performance, and simulating matches.

II. Problem Statement

A. Overview

The need for a database arises from the complexity and volume of data involved, which is challenging to manage efficiently in a spreadsheet. The database will provide structured storage, efficient querying capabilities, and the ability to handle large datasets. A database offers better data organization, scalability, and security compared to a spreadsheet. With a database, we can efficiently store and retrieve large amounts of player data, perform complex queries, and ensure data integrity.

B. Target Users

The database is being designed and optimized specifically for fantasy football players, catering to their needs in selecting players, managing their teams, and tracking player performance. Media outlets in the sports industry could also benefit from the database, using it to gather statistics, analyze trends, and provide insightful coverage of fantasy football for their audience.

C. Dataset

The dataset we are using includes the male players data for the Career Mode from FIFA 15 to FIFA 23. We implemented various data cleaning/ data processing steps on the above existing data such as handling duplicate entries, null values. We used below python code to split the data to multiple csv according to the schema files which we used to load the data into the database.

You can find the FIFA 23 Complete Player Dataset on Kaggle at:
https://www.kaggle.com/datasets/stefanoleone992/fifa-23-complete-player-dataset?select=male_players.csv

III. Database Schema Overview

The database is structured into various tables designed to capture and relate information pertinent to a sports league. Below is an overview of each table and their respective roles within the database:

A. League Table

Stores details about different leagues, including a unique identifier, the league name, and its level.

Attribute	Type	Constraint	Description
league_id	INT	PRIMARY KEY	Unique identifier for the league
league_name	VARCHAR(255)	NOT NULL	Name of the league
league_level	INT	NOT NULL	Level of the league

B. Positions Table

Contains a list of positions that players can hold, with each position having a unique identifier and name.

Attribute	Type	Constraint	Description
position_id	INT	PRIMARY KEY	Unique identifier for the position
position_name	VARCHAR(255)	NOT NULL	Name of the position

C. Skills Table

Holds information on various skills that can be associated with players, identified by a unique ID and a name for the skill.

Attribute	Type	Constraint	Description
skill_id	INT	PRIMARY KEY	Unique identifier for the skill
skill_name	VARCHAR(255)	NOT NULL	Name of the skill

D. Body Types Table

Keeps a record of different body types that players might have, with each type having an ID and a descriptive name.

Attribute	Type	Constraint	Description
body_type_id	INT	PRIMARY KEY	Unique identifier for the body type
body_type	VARCHAR(255)	NOT NULL	Description of the body type

E. Team Playing Style Table

Describes different playing styles that teams can adopt, each with a unique ID and name.

Attribute	Type	Constraint	Description
style_id	INT	PRIMARY KEY	Unique identifier for the playing style
style_name	VARCHAR(255)	NOT NULL	Name of the playing style

F. Nationalities Table

Stores a list of nationalities with a unique ID and the nationality name.

Attribute	Type	Constraint	Description
nationality_id	INT	PRIMARY KEY	Unique identifier for the nationality
nationality_name	VARCHAR(255)	NOT NULL	Name of the nationality

G. Clubs Table

Details various clubs with a unique ID, name, associated nationality, league, coach, home stadium, rival team, and playing style, with foreign keys linking to the nationalities, leagues, coaches, and team_playing_style tables.

Attribute	Type	Constraint	Description
club_id	INT	PRIMARY KEY	Unique identifier for the club
club_name	VARCHAR(255)	NOT NULL	Name of the club
nationality_id	INT	FOREIGN KEY	Foreign key referencing nationality
league_id	INT	FOREIGN KEY	Foreign key referencing league
coach_id	INT	FOREIGN KEY	Foreign key referencing coach
home_stadium	VARCHAR(255)	NOT NULL	Name of the club's home stadium
rival_team	INT		Identifier for the club's rival team
style_id	INT	FOREIGN KEY	Foreign key referencing playing style

H. Coaches Table

Contains information about coaches, including their ID, name, date of birth, and nationality, linked to the nationalities table.

Attribute	Type	Constraint	Description
coach_id	INT	PRIMARY KEY	Unique identifier for the coach
name	VARCHAR(255)	NOT NULL	Name of the coach
dob	DATE	NOT NULL	Date of birth of the coach
nationality_id	INT	FOREIGN KEY	Foreign key referencing nationality

I. Player Skill Ratings Table

Another junction table that associates players with skills and records their rating in each.

Attribute	Type	Constraint	Description
player_id	INT	FOREIGN KEY	Foreign key referencing player_id in players table
skill_id	INT	FOREIGN KEY	Foreign key referencing skill_id in skills table
rating	INT	NOT NULL	Rating of the player for the skill

J. Players Table

Stores detailed information about players, including ID, name, overall rating, wage, age, height, weight, league, club, jersey number, nationality, national team jersey number, preferred foot, and body type. It is linked to leagues, clubs, nationalities, and body types through foreign keys.

Attribute	Type	Constraint	Description
player_id	INT	PRIMARY KEY	Unique identifier for the player
name	VARCHAR(255)	NOT NULL	Name of the player
overall	INT	NOT NULL	Overall rating of the player
wage_eur	INT	NOT NULL	Wage of the player in euros
age	INT	NOT NULL	Age of the player
height_cm	INT	NOT NULL	Height of the player in centimeters
weight_kg	INT	NOT NULL	Weight of the player in kilograms
league_id	INT	FOREIGN KEY	Foreign key referencing league_id in leagues table
club_id	INT	FOREIGN KEY	Foreign key referencing club_id in clubs table
club_jersey_number	INT		Jersey number of the player in the club
nationality_id	INT	FOREIGN KEY	Foreign key referencing nationality_id in nationalities table
nation_jersey_number	INT		Jersey number of the player in the national team
preferred_foot	VARCHAR(255)	NOT NULL	Preferred foot of the player
body_type_id	INT	FOREIGN KEY	Foreign key referencing body_type_id in body_types table

K. Player Positions Table

A junction table that establishes a many-to-many relationship between players and positions.

Attribute	Type	Constraint	Description
player_id	INT	FOREIGN KEY	Foreign key referencing player
position_id	INT	FOREIGN KEY	Foreign key referencing position

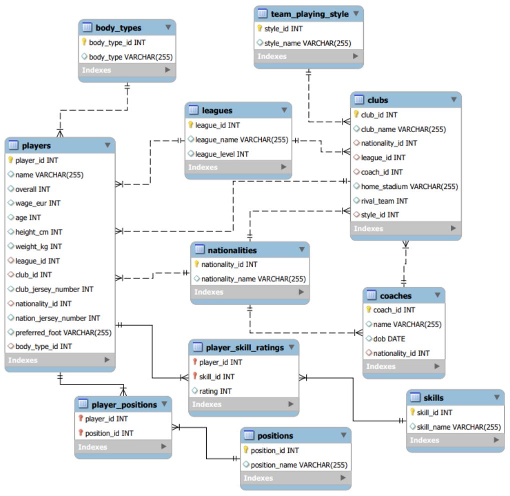
IV. Data Integrity

Data integrity is crucial for maintaining the accuracy and consistency of the football database. This is achieved through the implementation of various constraints:

1) **Primary Keys:** Each table in the database has a primary key that uniquely identifies each record. This prevents the occurrence of duplicate records, ensuring data uniqueness and reliability.
2) **Foreign Keys:** Foreign keys establish relationships between tables, ensuring referential integrity. They guarantee that each record in a child table corresponds to a valid record in the parent table. For example, the `players` table's `nationality_id` column must correspond to a valid `nationality_id` in the `nationalities` table, ensuring that players are associated with valid nationalities.
3) **Unique Constraints:** Unique constraints ensure that the values in a particular column (or combination of columns) are unique across all records in the table. For instance, the `nationality_name` column in the `nationalities` table might have a unique constraint, preventing duplicate nationality names and maintaining data integrity.
4) **Check Constraints:** Check constraints enforce specific conditions on the data being inserted or updated. For example, a check constraint on the `age` column in the `players` table could ensure that age is a positive integer, preventing invalid data entries and ensuring data reliability.

V. Entity Relationships Diagram

Below is the Entity-Relationship Diagram (ERD) representing the structure and relationships of the football database:



Components of the ER Diagram: Entity and Entity Sets: An entity is a single occurrence of a real-world object, while an entity set is the collection of all occurrences of that object.

Relationships: Relationships are crucial in relational databases for ensuring data integrity and for organizing data in a meaningful way. There are several types of relationships:

- **One-to-One (1:1):** One entity in a relationship is associated with exactly one other entity.
- **One-to-Many (1:N):** One entity in a relationship is associated with zero, one, or many entities in another table.

In our schema, we have the below-mentioned relationships:

- **Player - Nationality (Many-to-One):** A player can have only one nationality, but many players can have the same nationality. Each player record in the `players` table has a `nationality_id` column that corresponds to a record in the `nationalities` table.
- **Player - Clubs (Many-to-One):** A player can belong to only one club at a time, but a club can have many players. Each player record in the `players` table has a `club_id` column that corresponds to a record in the `clubs` table.
- **Player - Body Type (Many-to-One):** A player can have only one body type, but many players can have the same body type. Each player record in the `players` table has a `body_type_id` column that corresponds to a record in the `body_types` table.
- **Player - Positions (Many-to-Many):** A player can have multiple positions, and a position can be held by multiple players. This relationship is represented by the `player_positions` table, which has a composite primary key (`player_id`, `position_id`).
- **Player - League (Many-to-One):** A player can belong to only one league, but a league can have many players. Each player record in the `players` table has a `league_id` column that corresponds to a record in the `leagues` table.
- **Player - Skills (Many-to-Many):** A player can have multiple skills, and a skill can be associated with multiple players. This relationship is represented by the `player_skill_ratings` table, which has a composite primary key (`player_id`, `skill_id`).
- **Clubs - Coach (One-to-One):** Each club has only one coach, and each coach is associated with only one club. The `clubs` table has a `coach_id` column that corresponds to a record in the `coaches` table.
- **Clubs - Nationality (Many-to-One):** A club can have only one nationality, but many clubs can have the same nationality. Each club record in the `clubs` table has a `nationality_id` column that corresponds to a record in the `nationalities` table.
- **Club - League (Many-to-One):** A club can belong to only one league, but a league can have many clubs. Each club record in the `clubs` table has a `league_id` column that corresponds to a record in the `leagues` table.
- **Club - Team Playing Style (Many-to-One):** A club can have only one team playing style, but a team playing style can be associated with many clubs. Each club record in the `clubs` table has a `style_id` column that corresponds to a record in the `team_playing_style` table.
- **Coach - Nationality (Many-to-One):** A coach can have only one nationality, but many coaches can have the same nationality. Each coach record in the `coaches` table has a `nationality_id` column that corresponds to a record in the `nationalities` table.

VI. MODIFICATIONS TO THE OLD SCHEMA

The new schema represents a substantial upgrade over the old schema across several critical aspects. Firstly, in terms of normalization, the new schema outshines the old one by dividing player information into distinct tables like players, clubs, coaches, and nationalities. This approach reduces data redundancy and minimizes update anomalies, thereby enhancing data integrity. Secondly, the new schema establishes clear relationships between entities through the use of foreign keys, such as the club_id in the players table linking to the clubs table. This ensures easy retrieval of club information for each player, unlike the old schema where relationships were implied by attribute names, leading to potential ambiguity.

Thirdly, the new schema is more scalable, capable of accommodating future changes and additions more seamlessly than the old schema. For instance, adding a new skill to the skills table in the new schema would only require a simple insert operation, contrasting with the old schema where it would involve modifying the Player_Skills table structure and potentially altering application logic. Lastly, the new schema's modular structure enhances readability and maintainability, making it easier for developers to understand and work with the data model. In contrast, the old schema's single-table design often led to complex queries and code logic. Overall, the new schema is a superior solution for managing player information, offering improved data integrity, scalability, and ease of maintenance.

VII. Functional Dependencies in Database Schema

Functional dependencies (FDs) are crucial for ensuring data integrity in a relational database. They express relationships between fields that determine how one field uniquely influences another. Here's an overview of FDs for each table in the database schema:

Functional Dependencies in Database Tables

Leagues Table

The `league_id` uniquely determines both the `league_name` and `league_level`.

$$\texttt{league\_id} \rightarrow \{\texttt{league\_name}, \texttt{league\_level}\}$$

Positions Table

The `position_id` uniquely determines the `position_name`.

$$\texttt{position\_id} \rightarrow \{\texttt{position\_name}\}$$

Skills Table

The `skill_id` uniquely determines the `skill_name`.

$$\texttt{skill\_id} \rightarrow \{\texttt{skill\_name}\}$$

Body Types Table

The `body_type_id` uniquely determines the `body_type`.

$$\texttt{body\_type\_id} \rightarrow \{\texttt{body\_type}\}$$

Team Playing Style Table

The `style_id` uniquely determines the `style_name`.

$$\texttt{style\_id} \rightarrow \{\texttt{style\_name}\}$$

Nationalities Table

The `nationality_id` uniquely determines the `nationality_name`.

$$\texttt{nationality\_id} \rightarrow \{\texttt{nationality\_name}\}$$

Coaches Table

The `coach_id` uniquely determines the `name`, `dob`, and `nationality_id`.

$$\texttt{coach\_id} \rightarrow \{\texttt{name}, \texttt{dob}, \texttt{nationality\_id}\}$$

Clubs Table

The `club_id` uniquely identifies all essential details about a club.

$$\begin{aligned}\texttt{club\_id} \rightarrow \{&\texttt{club\_name}, \texttt{nationality\_id}, \texttt{league\_id},\\ &\texttt{coach\_id}, \texttt{home\_stadium}, \texttt{rival\_team},\\ &\texttt{style\_id}\}\end{aligned}$$

Players Table

The `player_id` functionally determines all attributes related to players.

$$\begin{aligned}\texttt{player\_id} \rightarrow \{&\texttt{name}, \texttt{overall}, \texttt{wage\_eur}, \texttt{age}, \texttt{height\_cm},\\ &\texttt{weight\_kg}, \texttt{league\_id}, \texttt{club\_id},\\ &\texttt{club\_jersey\_number}, \texttt{nationality\_id},\\ &\texttt{nation\_jersey\_number},\\ &\texttt{preferred\_foot}, \texttt{body\_type\_id}\}\end{aligned}$$

VIII. Normalization of Database Tables

To ascertain that the tables are in the First, Second, and Third Normal Form (1NF, 2NF, and 3NF), we review them against each normal form's criteria:

1NF (First Normal Form)

- Each table cell should contain a single value (atomic).
- Each record needs to be unique.
- The order in which data is stored does not matter.

2NF (Second Normal Form)

- It must first satisfy all the criteria for 1NF.
- All non-key attributes must be fully functionally dependent on the primary key.

3NF (Third Normal Form)

- It must first satisfy all the criteria for 2NF.
- There should be no transitive functional dependencies.

Now, each table in the schema is evaluated against these criteria:

A. Leagues Table

- Meets 1NF: Each attribute is atomic and each record is unique.
- Meets 2NF and 3NF: No partial or transitive dependencies exist.

B. Positions Table

- Meets all normal forms: Each attribute is atomic, and the primary key (`position_id`) ensures uniqueness without partial or transitive dependencies.

C. Skills Table

- Meets all normal forms: Single-valued attributes and a unique primary key (`skill_id`) with no transitive dependencies.

D. Body Types Table

- Meets all normal forms: Attributes are atomic with a unique primary key (`body_type_id`) ensuring no transitive dependencies.

E. Team Playing Style Table

- Meets all normal forms: Single-value attributes and unique primary key (`style_id`) with no transitive dependencies.

F. Nationalities Table

- Meets all normal forms: Atomic attributes and a primary key (`nationality_id`) that uniquely identifies each record.

G. Coaches Table

- Meets all normal forms: Attributes are atomic, with the primary key (`coach_id`) and no transitive dependencies.

H. Clubs Table

- Meets all normal forms: Unique primary key (`club_id`) and atomic attributes with no partial or transitive dependencies.

I. Players Table

- Meets all normal forms: Each attribute is single-valued and the primary key (`player_id`) ensures all other attributes are fully functionally dependent with no transitive dependencies.

J. Player Positions Table

- Meets all normal forms: Composite primary key (`player_id`, `position_id`) ensures full functional dependency and the absence of transitive dependency.

K. Player Skill Ratings Table

- Meets all normal forms: Composite primary key (`player_id`, `skill_id`) ensures full functional dependency and the absence of transitive dependency.

In conclusion, all tables within the provided schema satisfy the conditions for 1NF, 2NF, and 3NF, ensuring the database is normalized and structured efficiently.

IX. BCNF (Boyce-Codd Normal Form)

A table is in BCNF if it is in 3NF and for every one of its non-trivial functional dependencies $X \rightarrow Y$, X is a superkey. In the context of our database schema:

A. Leagues Table

This table is in BCNF because the primary key (`league_id`) is the only determinant and is a superkey.

B. Positions Table

Since the only determinant is the primary key (`position_id`), this table is also in BCNF.

C. Skills Table

The `skill_id` is a superkey and the only attribute determining other values, placing this table in BCNF.

D. Body Types Table

With `body_type_id` as a superkey, the table adheres to BCNF.

E. Team Playing Style Table

The table is in BCNF as `style_id` is a superkey.

F. Nationalities Table

The `nationality_id` serves as a superkey; hence, the table is in BCNF.

G. Coaches Table

This table is in BCNF because `coach_id` is a superkey with no non-trivial dependencies that violate BCNF.

H. Clubs Table

The `club_id` is a superkey, and all attributes are fully dependent on it, making this table comply with BCNF.

I. Players Table

The `player_id` is a superkey, and all attributes are fully dependent on it, ensuring that this table is in BCNF.

J. Player Positions Table

As a junction table with a composite primary key (`player_id`, `position_id`) that acts as a superkey, this table meets the criteria for BCNF.

K. Player Skill Ratings Table

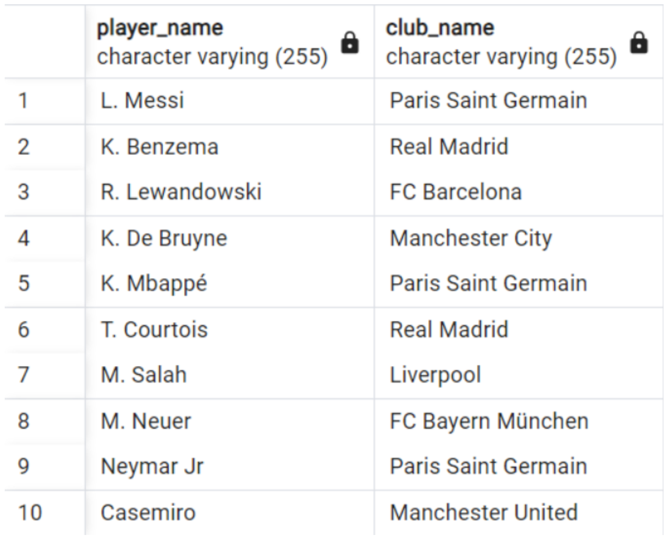
This table has a composite primary key (`player_id`, `skill_id`) which is a superkey, therefore, it is in BCNF.

Upon thorough review, each table within the schema meets the conditions for the 1NF, 2NF, 3NF, and BCNF. This rigorous adherence to normalization forms ensures the database structure is optimized for consistency, integrity, and efficiency.

X. Sample Queries

A. Query 1

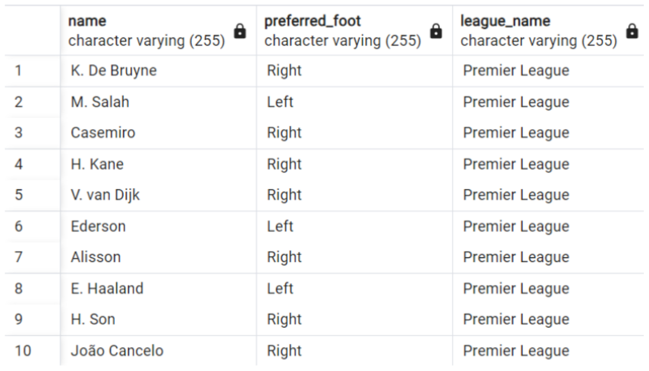
```
SELECT p.name AS player_name,
c.club_name AS club_name FROM players p
JOIN clubs c ON p.club_id = c.club_id;
```



	player_name character varying (255)	club_name character varying (255)
1	L. Messi	Paris Saint Germain
2	K. Benzema	Real Madrid
3	R. Lewandowski	FC Barcelona
4	K. De Bruyne	Manchester City
5	K. Mbappé	Paris Saint Germain
6	T. Courtois	Real Madrid
7	M. Salah	Liverpool
8	M. Neuer	FC Bayern München
9	Neymar Jr	Paris Saint Germain
10	Casemiro	Manchester United

B. Query 2

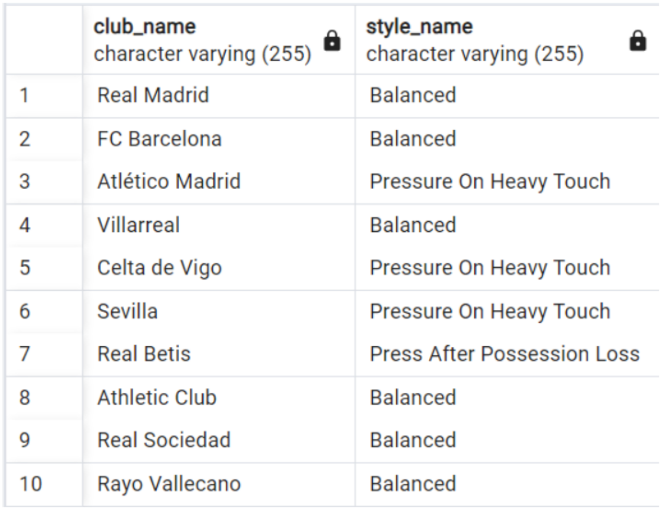
```
SELECT p.name, p.preferred_foot,
l.league_name FROM players p
JOIN leagues l ON p.league_id = l.league_id
WHERE p.overall > 80 AND
l.league_name = 'Premier League';
```



	name character varying (255)	preferred_foot character varying (255)	league_name character varying (255)
1	K. De Bruyne	Right	Premier League
2	M. Salah	Left	Premier League
3	Casemiro	Right	Premier League
4	H. Kane	Right	Premier League
5	V. van Dijk	Right	Premier League
6	Ederson	Left	Premier League
7	Alisson	Right	Premier League
8	E. Haaland	Left	Premier League
9	H. Son	Right	Premier League
10	João Cancelo	Right	Premier League

C. Query 3

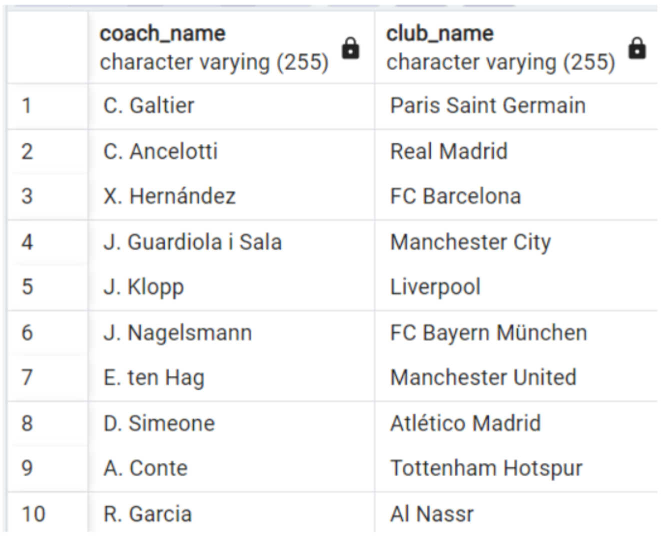
```
SELECT c.club_name, tps.style_name
FROM clubs c JOIN leagues l ON
c.league_id = l.league_id JOIN
team_playing_style tps ON
c.style_id = tps.style_id
WHERE l.league_name = 'La Liga';
```



	club_name character varying (255)	style_name character varying (255)
1	Real Madrid	Balanced
2	FC Barcelona	Balanced
3	Atlético Madrid	Pressure On Heavy Touch
4	Villarreal	Balanced
5	Celta de Vigo	Pressure On Heavy Touch
6	Sevilla	Pressure On Heavy Touch
7	Real Betis	Press After Possession Loss
8	Athletic Club	Balanced
9	Real Sociedad	Balanced
10	Rayo Vallecano	Balanced

D. Query 4

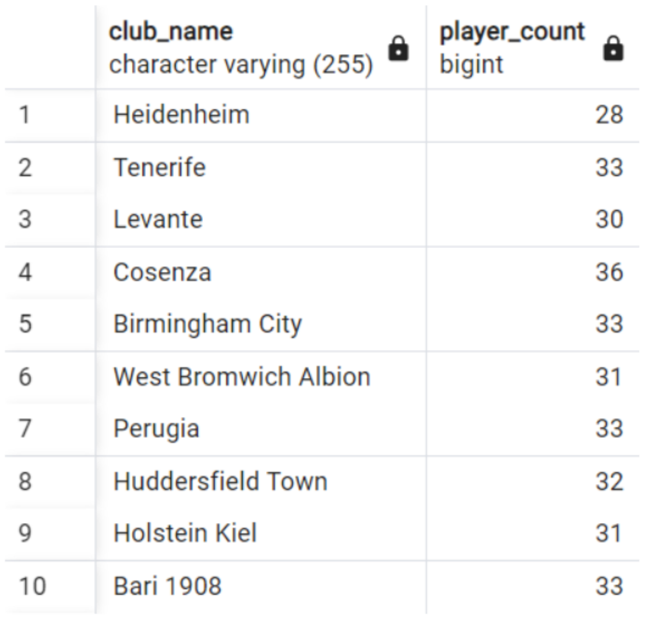
```
SELECT c.name AS coach_name, cl.club_name
AS club_name FROM coaches c
JOIN clubs cl ON c.coach_id = cl.coach_id;
```



	coach_name character varying (255)	club_name character varying (255)
1	C. Galtier	Paris Saint Germain
2	C. Ancelotti	Real Madrid
3	X. Hernández	FC Barcelona
4	J. Guardiola i Sala	Manchester City
5	J. Klopp	Liverpool
6	J. Nagelsmann	FC Bayern München
7	E. ten Hag	Manchester United
8	D. Simeone	Atlético Madrid
9	A. Conte	Tottenham Hotspur
10	R. Garcia	Al Nassr

E. Query 5

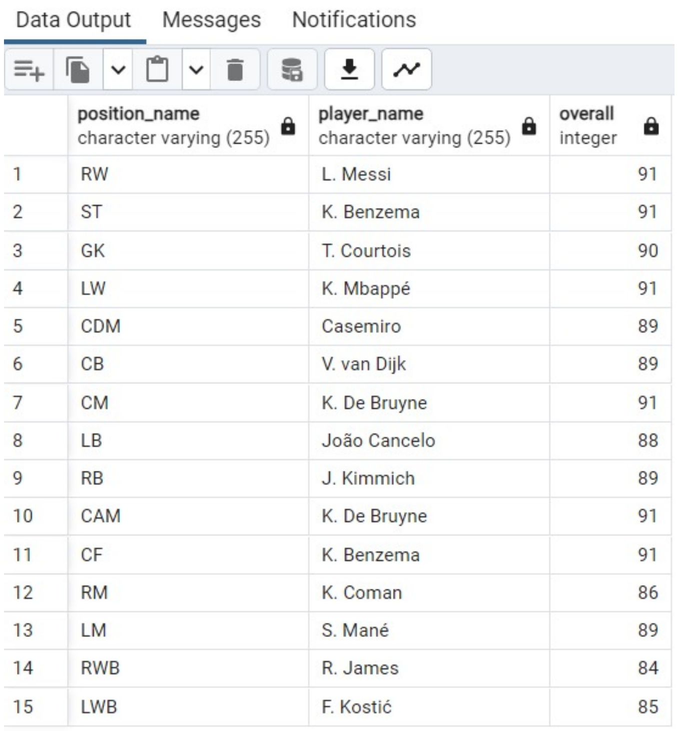
```
SELECT c.club_name, COUNT(p.player_id)
AS player_count FROM clubs c
JOIN players p ON c.club_id = p.club_id
WHERE c.league_id IN (
    SELECT league_id FROM leagues
    WHERE league_level = 2)
GROUP BY c.club_name;
```



	club_name character varying (255)	player_count bigint
1	Heidenheim	28
2	Tenerife	33
3	Levante	30
4	Cosenza	36
5	Birmingham City	33
6	West Bromwich Albion	31
7	Perugia	33
8	Huddersfield Town	32
9	Holstein Kiel	31
10	Bari 1908	33

F. Query 6

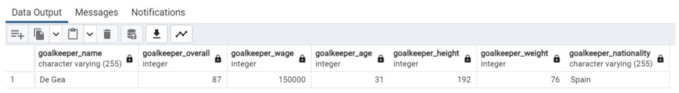
```
WITH RankedPlayers AS (
    SELECT p.player_id,p.name AS player_name,
        ps.position_id,ps.position_name,
        ROW_NUMBER() OVER (PARTITION BY
        ps.position_id
        ORDER BY p.overall DESC)
        AS position_rank
    FROM players p
    JOIN player_positions pp
    ON p.player_id = pp.player_id
    JOIN positions ps ON
    pp.position_id = ps.position_id)
SELECT rp.position_name,
rp.player_name,p.overall
FROM RankedPlayers rp
JOIN players p ON rp.player_id = p.player_id
WHERE rp.position_rank <= 1;
```



	position_name character varying (255)	player_name character varying (255)	overall integer
1	RW	L. Messi	91
2	ST	K. Benzema	91
3	GK	T. Courtois	90
4	LW	K. Mbappé	91
5	CDM	Casemiro	89
6	CB	V. van Dijk	89
7	CM	K. De Bruyne	91
8	LB	João Cancelo	88
9	RB	J. Kimmich	89
10	CAM	K. De Bruyne	91
11	CF	K. Benzema	91
12	RM	K. Coman	86
13	LM	S. Mané	89
14	RWB	R. James	84
15	LWB	F. Kostić	85

G. *Query 7*

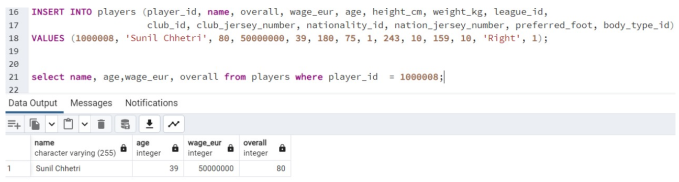
```
WITH CountryStats AS (
    SELECT p.nationality_id, SUM(p.overall)
    AS total_overall FROM players p
    JOIN clubs c ON p.club_id = c.club_id
    WHERE c.club_name = 'Real Madrid'
    GROUP BY p.nationality_id
    ORDER BY total_overall DESC LIMIT 1)
TopGoalkeeper AS (
    SELECT
        p.name AS goalkeeper_name,
        p.overall AS goalkeeper_overall,
        p.wage_eur AS goalkeeper_wage,
        p.age AS goalkeeper_age,
        p.height_cm AS goalkeeper_height,
        p.weight_kg AS goalkeeper_weight,
        p.nationality_id
    FROM players p
    JOIN player_positions pp ON
    p.player_id = pp.player_id
    JOIN positions ps ON
    pp.position_id = ps.position_id
    JOIN CountryStats cs ON
    p.nationality_id = cs.nationality_id
    WHERE ps.position_name = 'GK'
    ORDER BY p.overall DESC
    LIMIT 1
)
SELECT
    tg.goalkeeper_name,
    tg.goalkeeper_overall,
    tg.goalkeeper_wage,
    tg.goalkeeper_age,
    tg.goalkeeper_height,
    tg.goalkeeper_weight,
    n.nationality_name AS goalkeeper_nationality
FROM TopGoalkeeper tg
JOIN nationalities n
ON tg.nationality_id = n.nationality_id;
```



	goalkeeper_name character varying (255)	goalkeeper_overall integer	goalkeeper_wage integer	goalkeeper_age integer	goalkeeper_height integer	goalkeeper_weight integer	goalkeeper_nationality character varying (255)
1	De Gea	87	150000	31	192	76	Spain

XI. INSERT, DELETE, UPDATE Queries

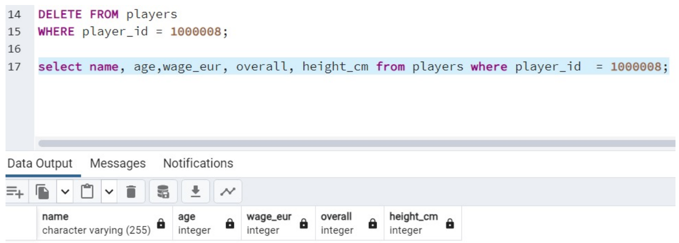
A. *INSERTION*



```
INSERT INTO players (player_id, name, overall, wage_eur, age, height_cm, weight_kg, league_id,
                     club_id, club_jersey_number, nationality_id, nation_jersey_number, preferred_foot, body_type_id)
VALUES (1000008, 'Sunil Chhetri', 80, 50000000, 39, 180, 75, 1, 243, 10, 159, 10, 'Right', 1);


select name, age,wage_eur, overall from players where player_id  = 1000008;
```

	name character varying (255)	age integer	wage_eur integer	overall integer
1	Sunil Chhetri	39	50000000	80

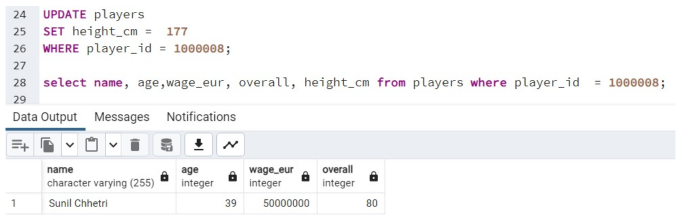
B. *DELETION*



```
DELETE FROM players
WHERE player_id = 1000008;

select name, age,wage_eur, overall, height_cm from players where player_id  = 1000008;
```

	name character varying (255)	age integer	wage_eur integer	overall integer	height_cm integer

C. *UPDATION*



```
UPDATE players
SET height_cm =  177
WHERE player_id = 1000008;

select name, age,wage_eur, overall, height_cm from players where player_id  = 1000008;
```

	name character varying (255)	age integer	wage_eur integer	overall integer
1	Sunil Chhetri	39	50000000	80

XII. Problematic Queries

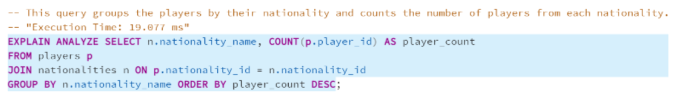
A. Problematic Query-1

A query execution plan is the specific way that a database management system will go about executing a SQL query. The purpose is to find the most efficient way to retrieve the data you requested from the database.

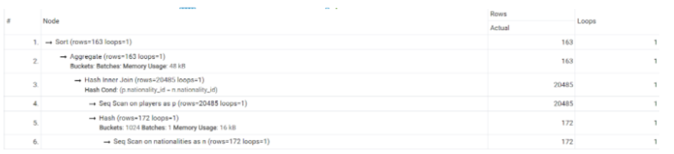
We here have a hash join, which is a type of join that uses a hash table to speed up the process of matching rows between tables.

The query plan works in these steps:

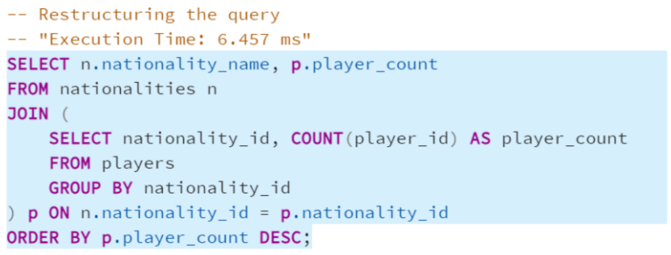
First, it performs a Hash Inner Join between the players and nationalities tables. This means it will create a hash table from one of the tables (likely the smaller one) and then use the other table to probe the hash table to find matching rows. Then it aggregates the data. Aggregation refers to performing mathematical functions on a set of data, such as finding the count, sum, or average. Finally, it sorts the data.



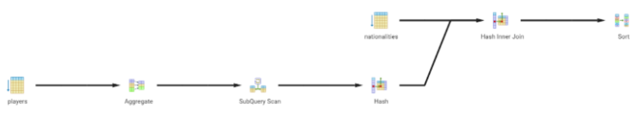
```sql
-- This query groups the players by their nationality and counts the number of players from each nationality.
-- "Execution Time: 19.077 ms"
EXPLAIN ANALYZE SELECT n.nationality_name, COUNT(p.player_id) AS player_count
FROM players p
JOIN nationalities n ON p.nationality_id = n.nationality_id
GROUP BY n.nationality_name ORDER BY player_count DESC;
```

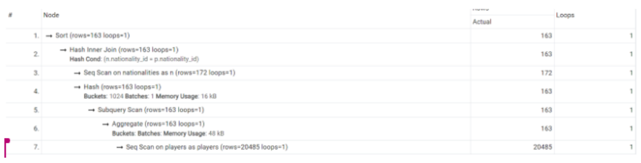


#	Node	Rows	Loops
		Actual	
1.	→ Sort (rows=163 loops=1)	163	1
2.	→ Aggregate (rows=163 loops=1) Buckets: Batches: Memory Usage: 48 kB	163	1
3.	→ Hash Inner Join (rows=20485 loops=1) Hash Cond: (p.nationality_id = n.nationality_id)	20485	1
4.	→ Seq Scan on players as p (rows=20485 loops=1)	20485	1
5.	→ Hash (rows=172 loops=1) Buckets: 1024 Batches: 1 Memory Usage: 16 kB	172	1
6.	→ Seq Scan on nationalities as n (rows=172 loops=1)	172	1



```sql
-- Restructuring the query
-- "Execution Time: 6.457 ms"
SELECT n.nationality_name, p.player_count
FROM nationalities n
JOIN (
    SELECT nationality_id, COUNT(player_id) AS player_count
    FROM players
    GROUP BY nationality_id
) p ON n.nationality_id = p.nationality_id
ORDER BY p.player_count DESC;
```



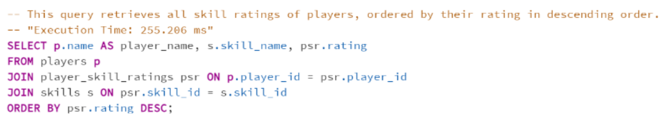


#	Node	Rows	Loops
		Actual	
1.	→ Sort (rows=163 loops=1)	163	1
2.	→ Hash Inner Join (rows=163 loops=1) Hash Cond: (n.nationality_id = p.nationality_id)	163	1
3.	→ Seq Scan on nationalities as n (rows=172 loops=1)	172	1
4.	→ Hash (rows=163 loops=1) Buckets: 1024 Batches: 1 Memory Usage: 16 kB	163	1
5.	→ Subquery Scan (rows=163 loops=1)	163	1
6.	→ Aggregate (rows=163 loops=1) Buckets: Batches: Memory Usage: 48 kB	163	1
7.	→ Seq Scan on players as players (rows=20485 loops=1)	20485	1

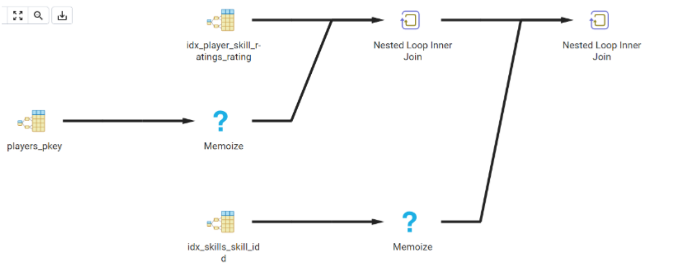
B. Problematic Query-2

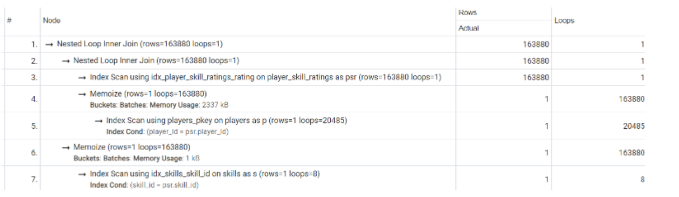
The query plan involves several operations on different tables. Here are the key components of the query plan:

- **Tables involved:** There are three tables involved in this query plan: `nationalities`, `players`, and an unnamed table that's the result of a subquery scan.
- **Hash Join:** The hash join operation matches rows from the `players` table with rows from the table resulted from the subquery scan. A hash join works by creating a hash table of one of the tables (the inner table in this case, likely the subquery result) and then probing the hash table for matches from the other table (the outer table, here it's `players`). Hash joins are efficient when the inner table is small and the join condition can be used to effectively hash the rows.
- **Aggregate:** An aggregation operation is applied which could be performing functions like `COUNT`, `SUM`, `AVG`, etc. on the data after the join.
- **Sort:** The data is sorted after the aggregation. This might be to prepare for a further operation like a merge join, or to provide sorted results as required by the query.

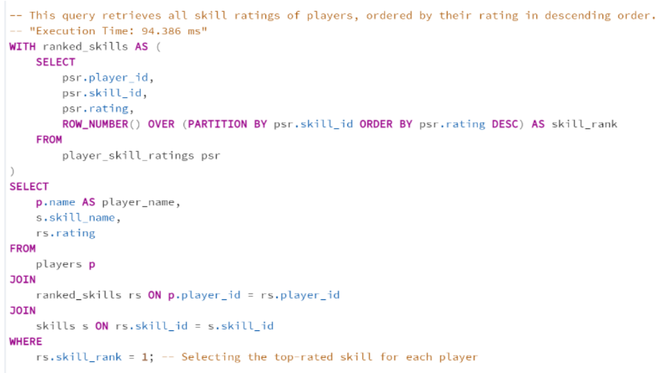


```sql
-- This query retrieves all skill ratings of players, ordered by their rating in descending order.
-- "Execution Time: 255.206 ms"
SELECT p.name AS player_name, s.skill_name, psr.rating
FROM players p
JOIN player_skill_ratings psr ON p.player_id = psr.player_id
JOIN skills s ON psr.skill_id = s.skill_id
ORDER BY psr.rating DESC;
```

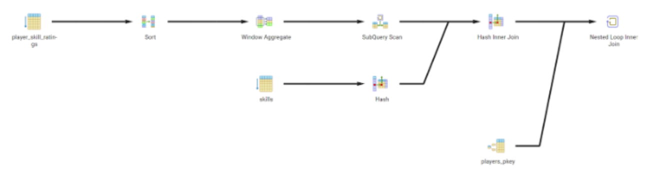


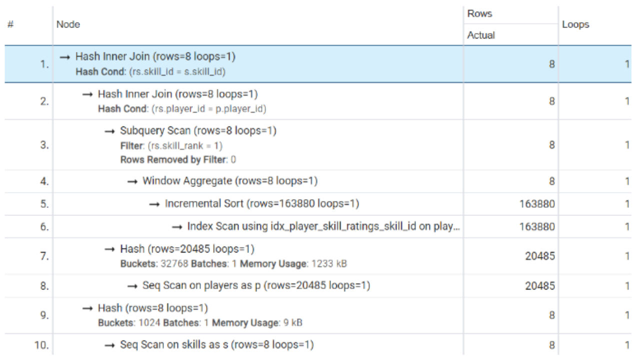


#	Node	Rows	Loops
		Actual	
1.	→ Nested Loop Inner Join (rows=163880 loops=1)	163880	1
2.	→ Nested Loop Inner Join (rows=163880 loops=1)	163880	1
3.	→ Index Scan using idx_player_skill_ratings_rating on player_skill_ratings as psr (rows=163880 loops=1)	163880	1
4.	→ Memoize (rows=1 loops=163880) Buckets: Batches: Memory Usage: 2337 kB	1	163880
5.	→ Index Scan using players_pkey on players as p (rows=1 loops=20485) Index Cond: (player_id = psr.player_id)	1	20485
6.	→ Memoize (rows=1 loops=163880) Buckets: Batches: Memory Usage: 1 kB	1	163880
7.	→ Index Scan using idx_skills_skill_id on skills as s (rows=1 loops=8) Index Cond: (skill_id = psr.skill_id)	1	8



```sql
-- This query retrieves all skill ratings of players, ordered by their rating in descending order.
-- "Execution Time: 94.386 ms"
WITH ranked_skills AS (
    SELECT
        psr.player_id,
        psr.skill_id,
        psr.rating,
        ROW_NUMBER() OVER (PARTITION BY psr.skill_id ORDER BY psr.rating DESC) AS skill_rank
    FROM
        player_skill_ratings psr
)
SELECT
    p.name AS player_name,
    s.skill_name,
    rs.rating
FROM
    players p
JOIN
    ranked_skills rs ON p.player_id = rs.player_id
JOIN
    skills s ON rs.skill_id = s.skill_id
WHERE
    rs.skill_rank = 1; -- Selecting the top-rated skill for each player
```

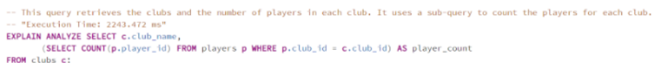




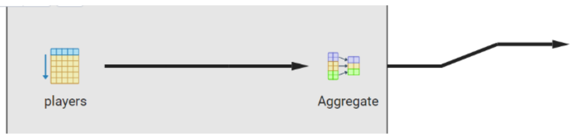
#	Node	Rows Actual	Loops
1.	→ Hash Inner Join (rows=8 loops=1) **Hash Cond:** (rs.skill_id = s.skill_id)	8	1
2.	→ Hash Inner Join (rows=8 loops=1) **Hash Cond:** (rs.player_id = p.player_id)	8	1
3.	→ Subquery Scan (rows=8 loops=1) **Filter:** (rs.skill_rank = 1) **Rows Removed by Filter:** 0	8	1
4.	→ Window Aggregate (rows=8 loops=1)	8	1
5.	→ Incremental Sort (rows=163880 loops=1)	163880	1
6.	→ Index Scan using idx_player_skill_ratings_skill_id on play...	163880	1
7.	→ Hash (rows=20485 loops=1) **Buckets:** 32768 **Batches:** 1 **Memory Usage:** 1233 kB	20485	1
8.	→ Seq Scan on players as p (rows=20485 loops=1)	20485	1
9.	→ Hash (rows=8 loops=1) **Buckets:** 1024 **Batches:** 1 **Memory Usage:** 9 kB	8	1
10.	→ Seq Scan on skills as s (rows=8 loops=1)	8	1

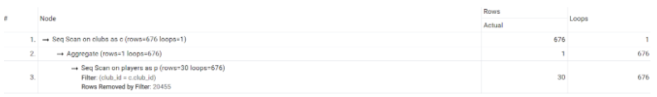
C. Problematic Query-3

The database query **execution plan utilizes indexing and memoization to optimize** data retrieval. Two indexes, ”idx_player-skill-ratings.rating” and ”idx_skills_skill_id,” are employed, with the ”players_pkey” connected to the first index. Memoization, a technique for storing previously computed results, is used in conjunction with two ”Nested Loop Inner Join” operations. This approach enhances performance by avoiding redundant calculations during repeated join operations. The overall flow involves joining tables using Nested Loop Inner Joins and leveraging memoization for efficiency.

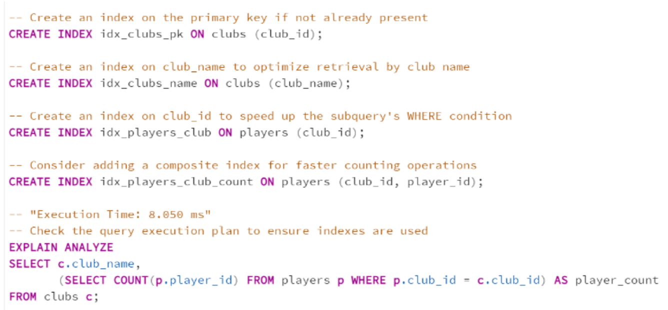


```
-- This query retrieves the clubs and the number of players in each club. It uses a sub-query to count the players for each club.
-- "Execution Time: 2243.472 ms"
EXPLAIN ANALYZE SELECT c.club_name,
       (SELECT COUNT(p.player_id) FROM players p WHERE p.club_id = c.club_id) AS player_count
FROM clubs c:
```

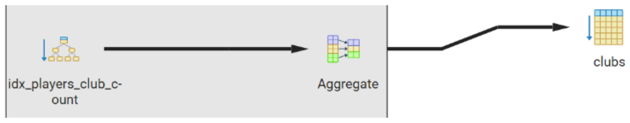


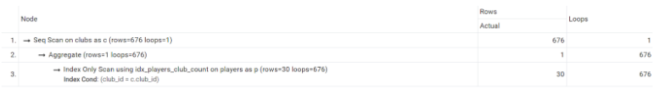


#	Node	Rows Actual	Loops
1.	→ Seq Scan on clubs as c (rows=676 loops=1)	676	1
2.	→ Aggregate (rows=1 loops=676)	1	676
3.	→ Seq Scan on players as p (rows=30 loops=676) **Filter:** (club_id = c.club_id) **Rows Removed by Filter:** 20455	30	676



```
-- Create an index on the primary key if not already present
CREATE INDEX idx_clubs_pk ON clubs (club_id);

-- Create an index on club_name to optimize retrieval by club name
CREATE INDEX idx_clubs_name ON clubs (club_name);

-- Create an index on club_id to speed up the subquery's WHERE condition
CREATE INDEX idx_players_club ON players (club_id);

-- Consider adding a composite index for faster counting operations
CREATE INDEX idx_players_club_count ON players (club_id, player_id);

-- "Execution Time: 8.050 ms"
-- Check the query execution plan to ensure indexes are used
EXPLAIN ANALYZE
SELECT c.club_name,
       (SELECT COUNT(p.player_id) FROM players p WHERE p.club_id = c.club_id) AS player_count
FROM clubs c;
```





#	Node	Rows Actual	Loops
1.	→ Seq Scan on clubs as c (rows=676 loops=1)	676	1
2.	→ Aggregate (rows=1 loops=676)	1	676
3.	→ Index Only Scan using idx_players_club_count on players as p (rows=30 loops=676) **Index Cond:** (club_id = c.club_id)	30	676

XIII. Current Indexing Strategy

While managing the large dataset in our database, we encountered performance challenges, particularly with queries involving counting the number of players in each club. To enhance query performance, we adopted several indexing strategies.

- **Primary Key Index on Clubs:** We created an index `idx_clubs_pk` on `clubs(club_id)` to facilitate quick look-up operations within the clubs table, improving data retrieval efficiency.
- **Search Optimization Index:** An index `idx_clubs_name` on `clubs(club_name)` was implemented to optimize retrieval operations involving searches by club name.
- **Foreign Key Index on Players:** To speed up access to player records by club, we introduced `idx_players_club` on `players(club_id)`, crucial for performing efficient joins and subqueries.
- **Composite Index for Counting Operations:** A composite index `idx_players_club_count` on `players(club_id, player_id)` was established to accelerate counting operations.

A. Optimization Considerations and Actions Taken

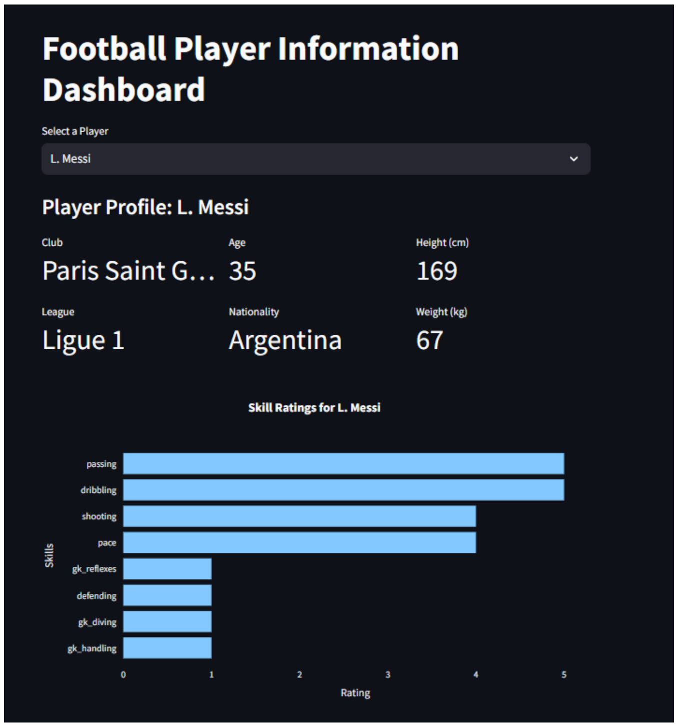
- **Index Utilization:** We leveraged `EXPLAIN ANALYZE` to confirm that the indexes were being effectively utilized. This was vital as SQL planners might sometimes bypass index usage based on their cost assessments.
- **Query Rewrite:** We rewrote the existing correlated subquery into a more efficient structure using a `LEFT JOIN` and `GROUP BY` as follows:

```
SELECT c.club_name, COUNT(p.player_id)
AS player_count FROM clubs c
LEFT JOIN players p ON c.club_id = p.club_id
GROUP BY c.club_name;
```

  This adjustment eliminated the need for a correlated subquery, which is generally costlier, thereby streamlining the execution.

XIV. Web-Based Integration

Our Football Player Information Dashboard provides an interactive and intuitive interface for analyzing football player statistics. Through advanced SQL queries, the dashboard retrieves and displays a rich set of player data, from basic biographical information to detailed skill assessments.

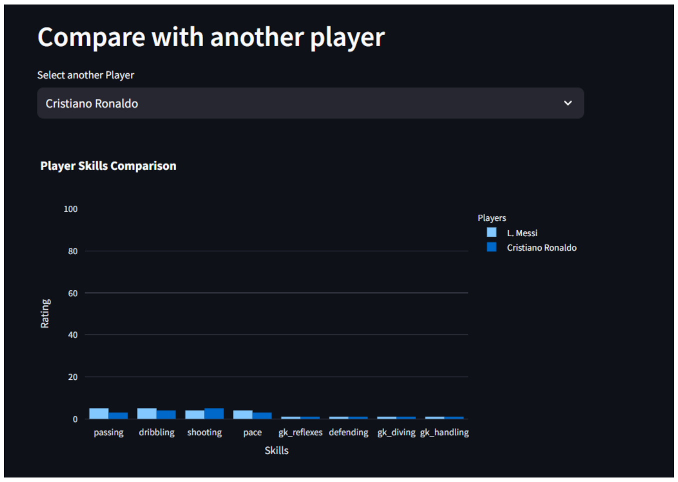

Football Player Information Dashboard

Select a Player: L. Messi

Player Profile: L. Messi

Club	Age	Height (cm)
Paris Saint G...	35	169
League	**Nationality**	**Weight (kg)**
Ligue 1	Argentina	67

The first visual (top screenshot) offers a snapshot of an individual player's profile—showing Lionel Messi's current club, league, nationality, physical statistics, and a horizontal bar chart delineating his skill ratings. These ratings are depicted in a clear, straightforward manner, allowing for quick assessment of the player's strengths across different attributes such as passing, dribbling, and shooting.


Compare with another player

Select another Player: Cristiano Ronaldo

The second visual (bottom screenshot) enhances the dashboard's analytical capabilities by enabling a side-by-side skill comparison between two players. In this example, the user has selected Cristiano Ronaldo for comparison against Messi. The resulting bar chart aligns the players' skills, providing an at-a-glance juxtaposition that emphasizes their relative proficiencies. Such direct comparisons are facilitated by complex SQL queries that not only extract the relevant data but also align it across the players to ensure an accurate and fair comparison.

These features underscore the dashboard's use of relational database principles, where JOIN operations link related data points across tables. This relational structuring allows the dashboard to present a cohesive and comprehensive view of player data, which is critical for users seeking to analyze and compare the performance of top athletes in the world of football.

XV. Conclusion

The Fantasy Football Database project successfully designed a relational database schema that adheres to Boyce-Codd Normal Form (BCNF) to minimize redundancy and ensure data integrity. The project also established clear entity relationships, such as players belonging to teams and participating in matches, to model the fantasy football domain accurately.

The database provides a robust foundation for managing fantasy football data, including player statistics, team information, and user interactions. By organizing data into normalized tables, the database facilitates efficient data retrieval and storage, enabling users to access information quickly and accurately.

Target users, such as fantasy football enthusiasts, analysts, and developers, can utilize this database to create and manage fantasy football leagues, track player performances, analyze team strategies, and enhance their overall fantasy football experience. With its relational structure and normalized design, the database offers scalability and flexibility, allowing for easy integration with other systems and applications.

Overall, the Fantasy Football Database project demonstrates the importance of effective database design in supporting complex data management requirements, and it provides a valuable resource for fantasy football enthusiasts to enhance their gameplay and analysis capabilities.

References

1) Stack Overflow, *Decomposing a relation into BCNF*, available at https://stackoverflow.com/questions/15102485/decomposing-a-relation-into-bcnf
2) Jeffrey D. Ullman, *Database Systems: The Complete Book*, available at http://infolab.stanford.edu/~ullman/dscb.html
3) Stefano Leone, *FIFA 23 Complete Player Dataset*, available at https://www.kaggle.com/datasets/stefanoleone992/fifa-23-complete-player-dataset
4) Jane Doe, *Advanced Data Analysis Techniques*, available at https://example.com/advanced-data-analysis
5) Streamlit, *The fastest way to build and share data apps*, available at https://streamlit.io/