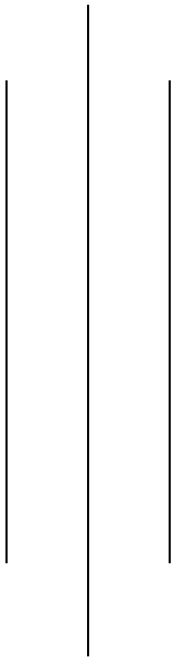
TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
PULCHOWK CAMPUS

A PROJECT REPORT ON
DATABASE MANAGEMENT SYSTEM



IPL ARCHIVAL SYSTEM

Submitted By:
Aditya Timalsina (077BEI009)
Ashutosh Bohara (077BEI012)
Pranav Joshi: (077BEI029)

Submitted To:
Bibha Sthapit
Department of Electronics and Computer Engineering

Acknowledgements:

In accordance to our partial curriculum requirements of the fifth semester of Electronics and Communication Engineering, we have completed a project on "IPL archive system" and thereafter prepared this report on the basis of our practical findings about an end to end application based on relational database management system.
We express our heartfelt thanks to our faculty facilitator Mrs. Bibha Sthapit for providing us the opportunity to implement our findings in the classroom in a real world project, and somewhat beyond it. Due to this activity, we were able to devise a system with our creative forte at work, which amalgamated with the subject rhetoric.

Abstract:

This report is aimed at implementation of an archival system for a cricket tournament in a league-based format. It utilizes relational database model to construct a comprehensive system for both the management side and the enthusiasts. A lot of data is generated every game, which if had to be logged individually, would be a tedious task. Our archive system is a repertoire for match records, player statistics and team based performances. We have implemented both addition to and retrieval from our database for an encompassing archival system experience. The system is accessed through a web-based interface by the end user. Relational database model is brought about by a relevant SQL design platform, with different languages for backend and frontend implementation.

Contents

1. Introduction

The game of cricket is a complex sport involving many variables to be considered by both players and the management team. Each inning of a cricket game generates a lot of data that, for the purpose of archiving, must be stored properly. Currently, the most popular cricket league is the Indian Premier League (IPL) with over 170 players joining in from across the world. The IPL is played in seasons occurring in the interval of 1 year, with each season having up to 80 games. With each new season of IPL, new players are added, and some players may retire.
IPL Archival System is a comprehensive cricket database, modelled on the Indian Premier League. It can work as a resource for cricket enthusiasts with a repository of one of the most prestigious cricket leagues globally. The system will provide access to an expansive collection of match records, player statistics, team performances, and a myriad of additional resources. Furthermore, the system will be accessible through a web-based user interface making it convenient and easy-to-use for the end user. The system will make the process of data retrieval, and performing complex queries effortless.
Although we will be designing our system around the Indian Premier League, it can be used with any other cricket leagues such as DPL, PPL, and Nepal T20 League. We decided to go with IPL, as the data for IPL games are more easily available online. However, the system can be exported to be used with other leagues with minimal redesign.

2. System Requirements

Since our primary task was to model a database in relational model, we have used PostgreSQL for it. Backend and frontend are designed with GoLang and ReactJS respectively. An end user or enterprise wishful of eperiencing our archival system must have the aforementioned environments installed and functional.

2.1. PostgreSQL

PostgreSQL, often simply referred to as "Postgres," is an open-source **relational database management** system (RDBMS) known for its robustness, extensibility, and advanced features. It is a powerful and feature rich database system, with significant support of the standard SQL system. Along with that, it has support for advanced data types and ACID com-

pliance. It supports Triggers and stored procedures that are used widely in our project. It supports creation of custom constraints for entities, and custom datatypes in an entity. PostgreSQL includes several procedural languages with the base distribution: PL/pgSQL, PL/Tcl, PL/Perl, and PL/Python. PostgreSQL allows developers to create custom data types, operators, functions, and aggregates using various programming languages. This extensibility enables developers to tailor the database to their specific application needs. PostgreSQL has advanced query optimization capabilities that help improve query performance. It also supports indexing, partitioning, and parallel query execution, allowing for efficient data retrieval and manipulation.

2.2. Golang

GoLang has extensive support for **backend** development and provides strong concurrency and parallelism support. Go's compiled nature and efficient runtime make it a high-performance language. It offers fast execution speeds and low memory overhead, which are essential for handling a large number of requests and data processing tasks in a backend environment. Go's characteristics are particularly advantageous in backend development, since our archival system is a complex system, and needs to be built and maintained over time to provide secure usability. Go's lightweight threads (goroutines) and its ability to handle a large number of concurrent connections make it suitable for building highly scalable systems. This is crucial for backend development where applications often need to handle a large number of users simultaneously.

2.3. ReactJS

React is built around the concept of reusable components. Each component encapsulates its own state, logic, and rendering, making it easier to manage and maintain complex UIs. This modular approach improves code organization and reusability. Moreover, our project follows a unidirectional data flow, where data flows in a single direction—from parent components to child components. This makes it easier to track data changes and debug issues with React, as the flow of data is more predictable. While React is commonly used for web applications, it can also be used to build mobile apps using frameworks like React Native. This allows developers to share code between web and mobile platforms, saving time and effort.

3. System Design:

Modelled on relational database architecture, the IPL archive system has data stored in a row-column format. The main constituents of the database are entities, relations, types and views.

3.1. Database

IPL archive system is a database system in which teams participate in a cricket league, modelled after the Indian Premier League. Different teams with their player roster, coaches, owners etc. compete in this league. The data requirements of the system are summarized as follows.
The league has different teams, each of which is identified by a unique ID. Each team is owned by one or many shareholding owners. Each team participates in a season/edition marked by a specific season number. In each season, a team recruits/retains players for a certain sum of money. A player can have dexterous preferences and affinity towards certain facet of the game . The support staff consists of Coaches, who are assigned to various responsibilites. A player is recognized by its unique id, while name, DoB and categorical affinity are other attributes associated with a player. A player can partake in a game in either of Batting or Bowling departments, also in both. Various stats are associated with a player's performance in a game in both batting and bowling departments. A game is always a part of a specific season and can either be a league stage game or a playoff. The teams with highest points total in the league stage qualify for the playoffs, with the 1^{st} and 2^{nd} teams playing the qualifier, and the 3^{rd} and 4^{th} teams competing in the eliminator. A season has one winner and personal accolades like Orange Cap for the highest run getter and Purple Cap for the highest wicket taker are awarded to players.

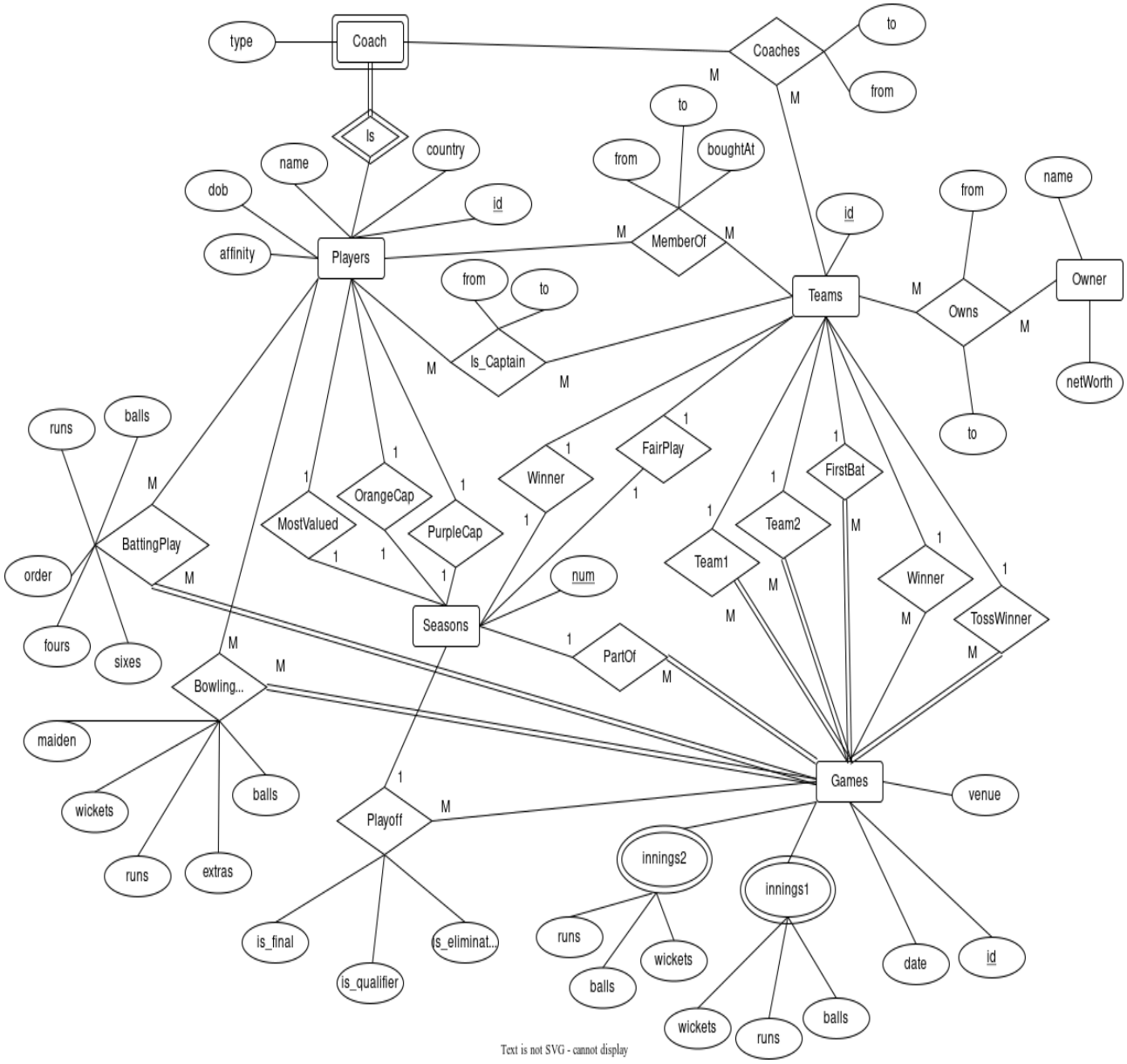


Figure 1: ER model of the database

BattingAffnity
LEFT_HANDED
RIGHT_HANDED

BowlingAffinity
RIGHT_HAND_FAST
RIGHT_HAND_MEDIUM_FAST
RIGHT_HAND_OFF_BREAK
RIGHT_HAND_LEG_BREAK
RIGHT_HAND_WRIST_SPIN
LEFT_HAND_FAST
LEFT_HAND_MEDIUM_FAST
LEFT_HAND_OFF_BREAK
LEFT_HAND_LEG_BREAK
LEFT_HAND_WRIST_SPIN

PlayerAffinity
BATSMAN
BOWLER
ALL_ROUNDER
WICKET_KEEPER
WICKET_KEEPER_BATSM

CoachType
BATTING
BOWLING
FIELDING
HEAD
FITNESS
PERFORMANCE

WicketMethod
BOWLED
CATCH
RUNOUT
STUMP
LBW
OTHERS
NOT_OUT

Figure 2: Attribute Value Constraints

Figure 3: Relational model of the database

3.1.1. Relational Schema

Players(Id, country, name, DoB, affinity, photo)

Teams(Id, name, abbrev, logo)

Coach(type, player_Id)

Games(Id, venue, date, team1, team2)

innings1(Wickets, Runs, balls, game_Id, innings1)

innings2(Wickets, runs, balls, game_Id, innings2)

Seasons(num, Player_Id, Teams_Id, Purple Cap, Orange Cap, Most Valued, Winner, Fair Play)

MemberOf(Player_Id, Teams_Id, from, to, boughtAt)

Is_captain(Players_Id, Teams_Id, to, from)

Coaches(to, from, team_Id,coach_Id)

Owner(Owner_Id, Owner_Name, netWorth)

Owns(Owner_Id, Teams_Id, to, from)

BattingPlay(Player_Id, Games_Id, runs, balls, order, fours, sixes)

BowlingPlay(Player_Id, Games_Id, runs, balls, extras, maiden, wickets)

As we can see the major four entites are: Players, Teams, Seasons and Games. Their definition in the SQL is as:

Player Entity Definition

```
create table if not exists Players (
      id serial not null primary key,
      "name" varchar(50) not null,
      country varchar(50) not null,
      "bYear" int,
      "affinity" PlayerAffinity NOT NULL,
      battingAffinity BattingAffinity,
      bowlingAffinity BowlingAffinity,

      photo varchar(100) not null default 'images/6.png'
);
```

Team Entity Definition

```
create table if not exists Teams (
      id serial not null primary key,
      "name" varchar(50),
      abbrev varchar(5),

      logo varchar(100) not null default 'images/6.png'
);
```

Seasons Entity Definition

```
create table if not exists Seasons (
      num int not null primary key,
      winner int not null,
      orangeCap int not null,
      purpleCap int not null,
      mostValued int not null,
      fairPlay int,

      constraint fk_winner foreign key(winner) references Teams(id
         ) on delete cascade on update cascade,

      constraint fk_orangeCap foreign key(orangeCap) references
         Players(id) on delete cascade on update cascade,
      constraint fk_purpleCap foreign key(purpleCap) references
         Players(id) on delete cascade on update cascade,
      constraint fk_mostValued foreign key(mostValued) references
         Players(id) on delete cascade on update cascade,

      constraint fk_fairPlay foreign key(fairPlay) references
         Teams(id) on delete cascade on update cascade
);
```

Games Entity Definition

```
create table if not exists Games (
      id serial not null primary key,
      team1 int not null,
      team2 int not null,
      gYear int not null,
      gMonth int not null,
      gDay int not null,
      tossWon int not null,
```

```
    firstBat int not null,
    winner int,
    venue int not null,
    innings1 int not null,
    innings2 int not null,
    seasonNo int not null,

    constraint fk_team1 foreign key(team1) references Teams(id)
       on delete cascade on update cascade,
    constraint fk_team2 foreign key(team2) references Teams(id)
       on delete cascade on update cascade,
    constraint fk_tossWon foreign key(tossWon) references Teams(
       id) on delete cascade on update cascade,
    constraint fk_firstBat foreign key(firstBat) references
       Teams(id) on delete cascade on update cascade,
    constraint fk_winner foreign key(winner) references Teams(id
       ) on delete cascade on update cascade,

    constraint fk_innings1 foreign key(innings1) references
       Innings(id) on delete cascade on update cascade,
    constraint fk_innings2 foreign key(innings2) references
       Innings(id) on delete cascade on update cascade,

    constraint fk_seasonNo foreign key(seasonNo) references
       Seasons(num) on delete cascade on update cascade
);
```

The following are the **tables** of our database:

- **Teams**
- **Players**
- **Innings**
- **Seasons**
- **Games**
- **MemberOf**
- **Coach**
- **BattingInnings**
- **BowlingInnings**
- **Owner**

- **Owns**
- **Captaincy**

3.1.2. Integrity Constraints, Dependencies and Normalization

Integrity Constraints are enforced to ensure the quality of information. We use an example of **Seasons** table to illustrate it.

a. Constraints on Single relation:

i. Primary Key Constraint: Here, the edition of a season is serialized by 'num' attribute. Each season entity is uniquely identified by value in its 'num' field, within the entity set of 'Seasons'.

ii. Not null: Since a null value is a common member of all domains, it is an allowed value for every attribute in PostgreSQL. However, for a valid IPL season, all the attributes must be non-zeros.

iii. Unique: Since we do not construct any super-key, the unique specification of no two tuples to have the same value on the listed attributes is not relevant.

iv. The Check Clause: We apply no predicates to any relation declaration directly.

b. Referential Integrity Constraint: Here, Seasons reference multiple other relations for some attributes. It references 'Teams' for fairPlay and winner. The constrains 'fk_winner' and 'fk_fairPlay' are foreign key constraints in this case. It means, for every value of foreign key in 'Seasons' must be available in 'Teams'. To further strengthen the integrity constraints, not form a design viewpoint but from a implementation viewpoint, we have also used triggers. The trigger below adds the winner for a season when a game with isFinal attribute set to 'true' is inserted.

```
CREATE OR REPLACE FUNCTION update_season_winner()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.isFinal THEN
       UPDATE Seasons
       SET winner = (SELECT winner FROM Games WHERE id = NEW.
          gameId)
       WHERE num = (SELECT seasonNo FROM Games WHERE id = NEW.
          gameId);
    END IF;
    RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_update_season_winner
AFTER INSERT ON Playoffs
FOR EACH ROW
EXECUTE FUNCTION update_season_winner();
```

Functional dependencies are an important part of our database design since we have modeled relational architecture, and it also further contributes to the aspect of normalization. Delving into trivial dependencies would be tiresome since there is a vast number of attributes in our database. Concerned with non-trivial functional dependencies, following are present:

a. Compound Determinants: From figure 2, In 'Owner' table, the time of stakeholding by an owner in a team is functionally dependent on the owner Id, and the team Id. That is, the composite attribute of owner Id and team Id is a determinant to the dependent attributes to and from. However, since (ownerId, teamId) constitute a candidate key, this does not violate the conditions for normalization in 2NF form. Similar arguments can be made for other relations as well.

b Full Functional Dependency: From figure 2, In 'MemberOf' table, the attributes of time of association of a player with a team, and the rate for the services acquired are two attributes that are fully functionally dependent upon the composite key of Player Id and Team Id, since a player can play for numerous teams in different time frames, it is necessary to validate the association with both team Id and player Id.

c. Partial Functional Dependency: Partial dependencies have been normalized in our database.

d. Transitive Dependency: A transitive dependency occurs when some attributes are fucntionally dependent on a non-prime attribute. There are no transitive dependencies present in the relational schema above, hence the schema is also is in 3NF form.

e. Multivalued Dependency: Previously, innings1 and innings2 were present as multivalued attributes in a Game. When converting them to relational schema the multi-valued attributes were flattened. It introduced attributes balls, wickets and runs for each innings. Hence, MVD existed in the initial relational model and can be stated as innings1$\rightarrow\rightarrow$balls

innings1→→ wickets
innings1 →→ runs
The relations were then normalized to BCNF by introducing separate relations for Innings that references the Games relation as present in the relational table.

To accomodate values which are limited to certain types of constraints only, we have made use of 'type' functionality of PostgreSQL and assigned enums to each attribute. The following are the **type attributes** in our database:

- **PlayerAffinity**
- **BattingAffinity**
- **BowlingAffinity**
- **CoachType**
- **WicketMethod**

Types

```
create type PlayerAffinity as enum ('BATSMAN', 'BOWLER', 'ALL
   ROUNDER', 'WICKET KEEPER', 'WICKET KEEPER BATSMAN');
create type BattingAffinity as enum ('RIGHT HANDED', 'LEFT HANDED'
   );
create type BowlingAffinity as enum ('RIGHT HAND FAST', 'RIGHT
   HAND MEDIUM FAST',
                                    'RIGHT HAND OFF BREAK', 'RIGHT
                                       HAND LEG BREAK', 'RIGHT HAND
                                       WRIST SPIN',
                                    'LEFT HAND FAST', 'LEFT HAND
                                       MEDIUM FAST',
                                    'LEFT HAND OFF BREAK', 'LEFT HAND
                                        LEG BREAK', 'LEFT HAND WRIST
                                       SPIN');
create type CoachType as enum ('HEAD', 'BATTING', 'BOWLING', '
   FIELDING', 'FITNESS', 'PERFORMANCE');
create type WicketMethod as enum ('BOWLED', 'CATCH', 'RUNOUT',
                                  'STUMP', 'LBW', 'OTHERS');
```

To provide tailored presentation of data contained in one or more tables and aggregate the result of queries in a systematic way, we have used views. Views here are created on multiple tables. Also, view relations are stored once updated, therefore the actual relations used in the views are thus kept up-to-date.
The following are the **views** in our database:

- **BattingStats**
- **BowlingStats**
- **TeamStats**
- **Tableinfo**

View for TeamStat

```
create view TeamStats (
      teamId, playerCount, gamesPlayed, seasonsPlayed, gamesWon,
         seasonsWon
) as
select t0.id, coalesce(t1.playerCount, 0), coalesce((t2.
   gamesPlayed1 + t3.gamesPlayed2), 0) as gamesPlayed, coalesce((
   t2.seasonsPlayed1 + t3.seasonsPlayed2), 0) as seasonsPlayed,
   coalesce(t4.gamesWon, 0), coalesce(t5.seasonsWon, 0)
from
(((((
      (select id
       from Teams) t0

left join
      (select teamId, count(playerId) as playerCount
       from MemberOf
       group by teamId) t1
on id = teamId)

left join
      (select team1, count(id) as gamesPlayed1, count(distinct
         seasonNo) as seasonsPlayed1
       from Games
       group by team1) t2
on teamId = team1)

left join
      (select team2, count(id) as gamesPlayed2, count(distinct
         seasonNo) as seasonsPlayed2
       from Games
       group by team2) t3
on teamId = team2)

left join
      (select winner as gwinner, count(winner) as gamesWon
       from Games
       group by winner) t4
on teamId = gwinner)

left join
      (select winner as swinner, count(winner) as seasonsWon
       from Seasons
       group by winner) t5
```

```
on teamId = swinner);
```

3.2. Backend

Backend environment consists of a .mod file and a main program. The .mod file lists all the dependencies i.e. the external packages and modules that the project relies on. Dependencies are specified along with its version in this module. The semantic versioning of the system that we used was `go 1.20`

And the requirements for this project are specified as:

Dependencies

```
module github.com/autives/iplarchive

go 1.20

require (
  github.com/blockloop/scan v1.3.0
  github.com/gorilla/mux v1.8.0
  github.com/lib/pq v1.10.9
  github.com/rs/cors v1.9.0
)

require (
  github.com/jmoiron/sqlx v1.3.5 // indirect
  github.com/mattn/go-sqlite3 v1.14.6 // indirect
)
```

The entities of the database are represented here as struct encapsulation, with each of the attributes being variables of a relevant data type. Moreover, team stats and table info are accomodated.

Structure Definition

```
type Player struct {
  Id             int    `json:"id" sql-insert:"f" db:"id"`
  Name           string `json:"name" db:"name"`
  Country        string `json:"country" db:"country"`
  BYear          int    `json:"bYear" db:"bYear"`
  Affinity       string `json:"playerAffinity" db:"affinity"`
  BattingAffinity string `json:"battingAffinity" db:"
     battingaffinity"`
```

```
  BowlingAffinity string `json:"bowlingAffinity" db:"
     bowlingaffinity"`
  Photo          string `json:"photo" db:"photo"`
}

type Team struct {
  Id     int    `json:"id" sql-insert:"f" db:"id"`
  Name   string `json:"name" db:"name"`
  Abbrev string `json:"abbrev" db:"abbrev"`
  Logo   string `json:"logo" db:"logo"`
}

type Seasons struct {
  Num       int `json:"num" sql-insert:"f" db:"num"`
  Winner    int `json:"winner" db:"winner"`
  OrangeCap int `json:"orangeCap" db:"orangecap"`
  PurpleCap int `json:"purpleCap" db:"purplecap"`
  MostValued int `json:"mostValued" db:"mostvalued"`
  FairPlay  int `json:"fairPlay" db:"fairplay"`
}

type Games struct {
  Id       int `json:"id" sql-insert:"f" db:"id"`
  Team1    int `json:"team1" db:"team1"`
  Team2    int `json:"team2" db:"team2"`
  GYear    int `json:"gYear" db:"gyear"`
  GMonth   int `json:"gMonth" db:"gmonth"`
  GDay     int `json:"gDay" db:"gday"`
  TossWon  int `json:"tossWon" db:"tosswon"`
  FirstBat int `json:"firstBat" db:"firstbat"`
  Winner   int `json:"winner" db:"winner"`
  Venue    int `json:"venue" db:"venue"`
  Innings1 int `json:"innings1" db:"innings1"`
  Innings2 int `json:"innings2" db:"innings2"`
  SeasonNo int `json:"seasonNo" db:"seasonno"`
}
```

This struct is of entity Games. It defines attributes ID, Team names, date of the game, toss status, innings status, venue and the winner. A game can be either in league stage or playoffs. A Go struct retrieves data in the form of variable present as json in the database. For example, json:"id": This struct tag indicates that when the struct is serialized to JSON, the field should be represented with the key "id" in the JSON object.

Here, Coach is a weak entity set dependent on Player entity set. Therefore,

Player's primary key, PlayerId behaves as a primary key for this entity, with discriminant as Coach ID in ID.
Now, let us briefly walk through some of the functions actually responsible in generating queries.

Insert Query

```
func genericInsertQuery(db *sqlx.DB, table string, cols interface
   {}) string {
  val := reflect.ValueOf(cols).Elem()

  var colNames, values strings.Builder
  for i := 0; i < val.NumField(); i++ {
    f := val.Type().Field(i)
    if f.Tag.Get("sql-insert") == "f" {
      continue
    }

    colName := f.Tag.Get("db")
    if colName == "" {
      colName = f.Name
    }

    colNames.WriteString("\"" + colName + "\", ")
    values.WriteString(fmt.Sprintf(formats[f.Type.Kind()], val.
       Field(i).Interface()) + ", ")
  }
  colStr := strings.TrimRight(colNames.String(), ", ")
  valStr := strings.TrimRight(values.String(), ", ")

  query := fmt.Sprintf("INSERT INTO %s(%s) VALUES (%s)", table,
     colStr, valStr)
  fmt.Printf("Log [INSERT]: %s\n", query)
  return query
}
```

genericInsertQuery() is a function that takes three parameters: a *sqlx.DB representing a database connection, a table string specifying the name of the table to insert into, and cols which is an interface that is expected to contain a pointer to a struct. The function starts by using reflection to get the underlying value of the cols interface. It's assumed that cols is a pointer to a struct, so Elem() is used to get the actual struct value. The colNames and values builders are initialized. These builders will be used to construct the column names and values lists for the SQL query. A loop

iterates through the fields of the struct. For each field:

- The sql-insert struct tag is checked. If the tag has the value ”f”, the field is skipped, likely indicating it shouldn't be included in the insert query.
- The db struct tag is checked to get the column name from the tag. If the tag is not present or empty, the field name is used as the column name.
- The column name is appended to the colNames builder.
- The field value is formatted according to its type's kind (using the formats map) and appended to the values builder.
- After looping through all fields, trailing commas are trimmed from the colNames and values builders to remove the last comma.
- The final SQL INSERT query is constructed using the table name, the constructed column names, and the constructed values.
- The function returns the generated SQL query.

3.3. Frontend

The frontend implementation is a web-based implementation. There are primarily two areas in which the frontend application is designed upon.

- **Components:** The components here are mostly functional components. These are also known as ”functional stateless components.” They are defined using JavaScript functions. Functional components receive props (short for properties) as arguments and return JSX (JavaScript XML) elements, which describe what should be rendered on the screen. They don't have their own internal state and are mostly used for rendering UI based on props. A component describes the UI of a particular page, or a particular component of a page.
- **Pages:** Different views or screens that make up a user interface are encapsulated under pages.Each page represents a distinct section of our application with its own content and functionality.

Fetching the data using API

```
useEffect(() => {
   const GetEnums = async () => {
     const playerAff = await axios.get(`/getEnum?enum=
        PlayerAffinity`);
     setEnums((prev) => ({
       ...prev,
       PlayerAffinity: playerAff.data,
     }));

     const battingAff = await axios.get(`/getEnum?enum=
        BattingAffinity`);
     setEnums((prev) => ({
       ...prev,
       BattingAffinity: battingAff.data,
     }));

     const bowlingAff = await axios.get(`/getEnum?enum=
        BowlingAffinity`);
     setEnums((prev) => ({
       ...prev,
       BowlingAffinity: bowlingAff.data,
     }));
     setIsFetched(true);
   };

   GetEnums();
   const getTeamData = async () => {
     try {
       const res = await axios.get(`/teams`);
       setTeamData(res.data["teams"]);
     } catch (error) {
       if (error.response) {
         console.log(error.response.data);
         console.log(error.response.status);
         console.log(error.response.header);
         setIsErr(error.message);
       } else {
         console.log("Error :" + `${error.message}`);
       }
     }
   };
 })
```

A. Snapshots

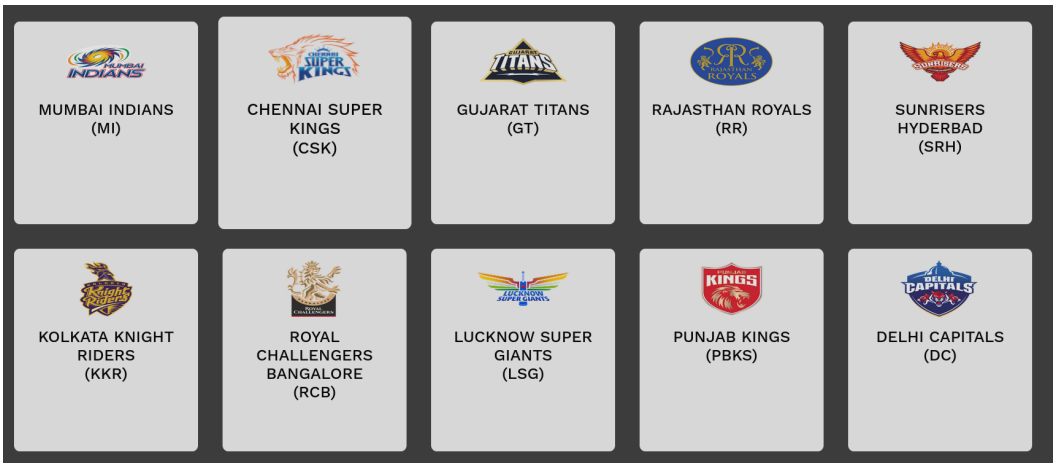


Figure 4: List of Teams

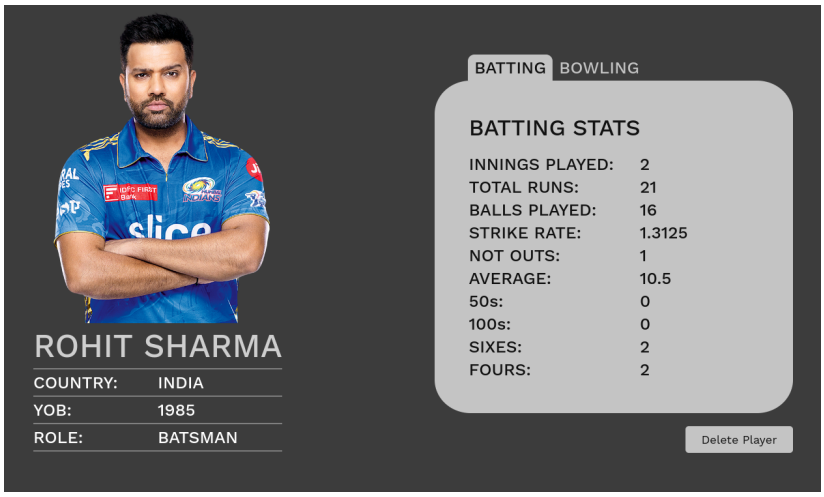


Figure 5: Player Stats

Figure 6: Forms

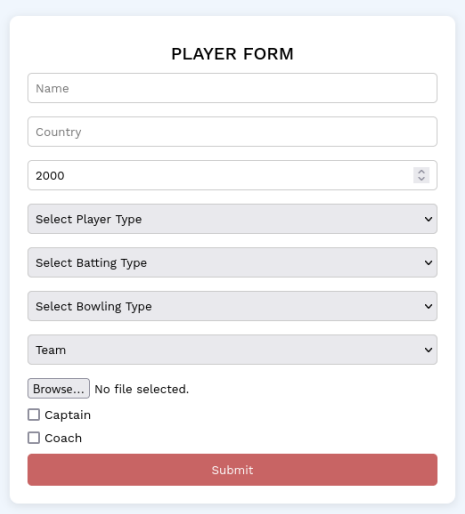


Figure 7: Player Form

Figure 8: Game Form