****

**Gebze Technical University**

**CSE 414 Introduction to Database**

*ESports Database*

*Project Report*

Kağan Çakıroğlu

200104004038

Index

1. User Requirements
2. Entity Relationships & Schema
3. Functional Dependencies
4. Normalization Examples
5. Implemented Database on Microsoft SQL Server
6. Built Queries
7. User Interface
8. **User Requirements**

When I start collecting information about user requirements for databases like mine I did think of myself as a company first who needs a wide and comprehensive database for E-Sport communities. I have specified every requirement based on my entities:

**1. Manager Requirements**

* **Manage Teams**: Ability to create, update, and delete teams.
* **Assign Managers**: Assign managers to teams.
* **Track Manager Experience**: Store and update manager's experience details.
* **Contact Information**: Store manager's contact information including email and phone number.

**2. Sponsor Requirements**

* **Sponsor Information**: Maintain sponsor details including name, contact email, and phone.
* **Sponsor Teams and Tournaments**: Ability to link sponsors to multiple teams and tournaments.
* **Manage Sponsorship Deals**: Create and manage sponsorship deals with financial details and duration.

**3. Game Requirements**

* **Game Catalog**: Maintain a catalog of games with details like name and unique identifiers.
* **Link to Players, Casters, and Streamers**: Track which players, casters, and streamers are associated with which games.

**4. Team Requirements**

* **Team Management**: Create, update, and delete team information.
* **Assign Manager**: Link teams to their respective managers.
* **Assign Sponsors**: Link teams to their sponsors.
* **Manage Players**: Assign players to teams, and track their involvement.

**5. Player Requirements**

* **Player Profile**: Create and manage player profiles with details like nickname, nationality, age, and contact information.
* **Team Membership**: Track team membership, allowing for players to be part of multiple teams over time.
* **Player Statistics**: Maintain detailed statistics for players including matches played, wins, losses, and performance metrics.
* **Build Management**: Track the builds created by players for different games.

**6. Caster Requirements**

* **Caster Profile**: Create and manage caster profiles with contact details and main casting game information.
* **Game Association**: Track which games casters specialize in.

**7. Tournament Requirements**

* **Tournament Management**: Create, update, and delete tournaments with details like name, location, prize pool, and dates.
* **Sponsor Association**: Link tournaments to their sponsors.
* **Venue Management**: Assign venues to tournaments.
* **Match Scheduling**: Schedule matches within tournaments and maintain match details.

**8. Team Owner Requirements**

* **Owner Profile**: Create and manage team owner profiles with contact information.
* **Team Ownership**: Track which teams are owned by which owners.

**9. Build Requirements**

* **Build Management**: Create and manage builds associated with players and games.

**10. Streamer Requirements**

* **Streamer Profile**: Create and manage streamer profiles linked to player profiles.
* **Streaming Platform Details**: Track streaming platform information and favorite games.

**11. Match Requirements**

* **Match Details**: Create and manage match details including participating teams, dates, and results.
* **Tournament Association**: Link matches to specific tournaments.

**12. Venue Requirements**

* **Venue Information**: Maintain venue details including name, location, and capacity.
* **Tournament Hosting**: Link venues to tournaments.

**13. Schedule Requirements**

* **Match Scheduling**: Create and manage schedules for matches within tournaments.

**14. Sponsorship Deal Requirements**

* **Deal Details**: Create and manage sponsorship deals, including financial details, start and end dates.
* **Sponsor and Team Association**: Link deals to sponsors and teams.

**15. Player Statistics Requirements**

* **Performance Metrics**: Maintain detailed performance statistics for players.

**16. Team Statistics Requirements**

* **Team Performance Metrics**: Maintain detailed performance statistics for teams.

**17. Event Logging Requirements**

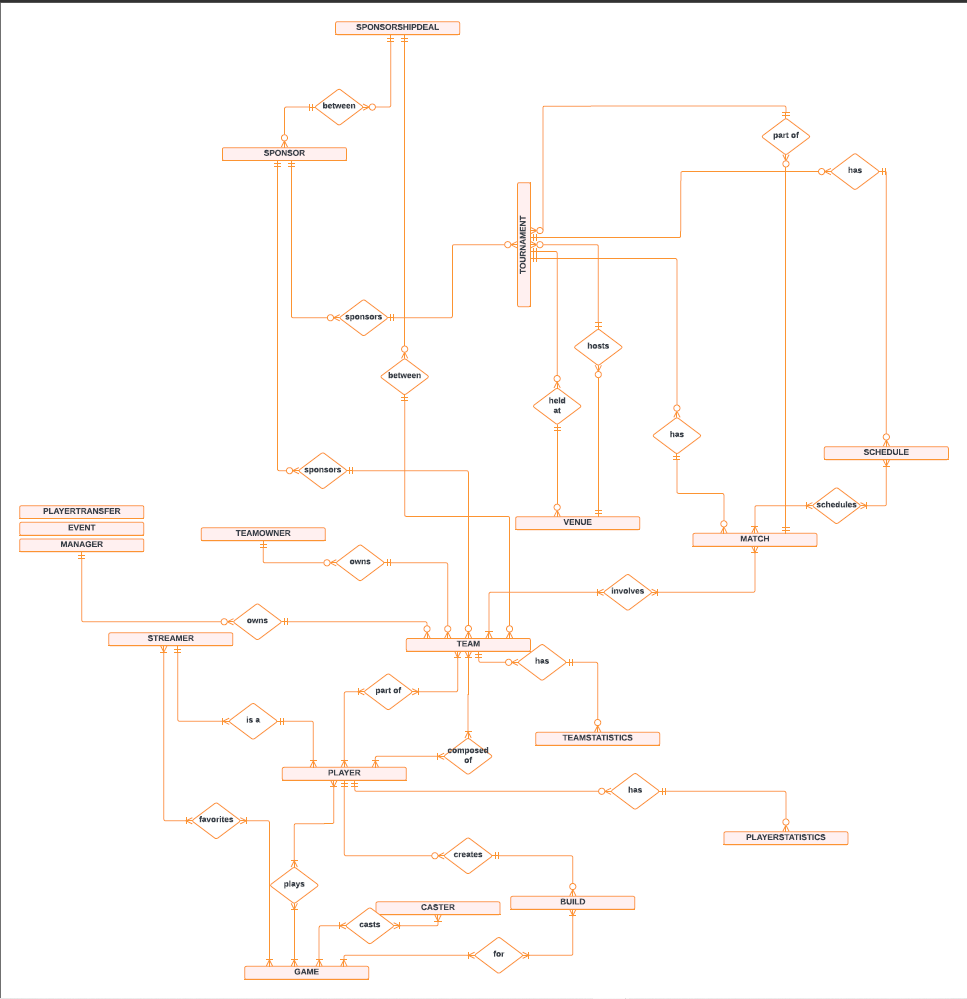
* **Event Management**: Log various events within the organization, such as player transfers and significant achievements.

**18. Player Transfer Requirements**

* **Transfer Details**: Track player transfers between teams, including transfer dates and fees.

1. **Entity Relationships**

*ER Diagram:*



1. **Functional Dependencies:**

**Manager**

1. ManagerID → Name, Email, PhoneNumber, Experience

**Sponsor**

1. SponsorID → SponsorName, ContactEmail, ContactPhone

**Game**

1. GameID → GameName

**Team**

1. TeamID → TeamName, ManagerID, SponsorID
2. ManagerID → TeamID (if each manager can only manage one team, otherwise, this is a one-to-many relationship)
3. SponsorID → TeamID (if each sponsor can only sponsor one team, otherwise, this is a one-to-many relationship)

**Player**

1. PlayerID → Nickname, MainGame, Nationality, Age, TeamID, Email, Phone

**Caster**

1. CasterID → Name, Email, PhoneNumber, MainCastingGameID
2. MainCastingGameID → GameID (if each caster can only cast one game, otherwise, this is a one-to-many relationship)

**Tournament**

1. TournamentID → TournamentName, Location, PrizePool, SponsorID

**TeamOwner**

1. OwnerID → Name, Email, PhoneNumber, TeamID

**Build**

1. BuildID → BuildName, PlayerID, GameID
2. PlayerID → BuildID (if each player can only create one build, otherwise, this is a one-to-many relationship)

**Streamer**

1. StreamerID → PlayerID, StreamingPlatform, Email, PhoneNumber, FavoriteGameID
2. PlayerID → StreamerID (if each player can only be one streamer, otherwise, this is a one-to-one relationship)

**Match**

1. MatchID → TournamentID, Team1ID, Team2ID, MatchDate, WinnerTeamID
2. TournamentID → MatchID (if each tournament can only have one match, otherwise, this is a one-to-many relationship)
3. Team1ID, Team2ID → MatchID (if the combination of teams uniquely identifies a match)

**Venue**

1. VenueID → VenueName, Location, Capacity

**Schedule**

1. ScheduleID → TournamentID, MatchID, ScheduledDateTime
2. TournamentID, MatchID → ScheduleID (if each match in a tournament has a unique schedule, otherwise, this is a one-to-many relationship)

**SponsorshipDeal**

1. SponsorshipDealID → SponsorID, TeamID, DealValue, StartDate, EndDate
2. SponsorID, TeamID → SponsorshipDealID (if the combination of sponsor and team uniquely identifies a sponsorship deal)

**PlayerStatistics**

1. PlayerStatisticsID → PlayerID, MatchesPlayed, Wins, Losses, KDA, AverageScore
2. PlayerID → PlayerStatisticsID (if each player can only have one set of statistics, otherwise, this is a one-to-many relationship)

**TeamStatistics**

1. TeamStatisticsID → TeamID, MatchesPlayed, Wins, Losses
2. TeamID → TeamStatisticsID (if each team can only have one set of statistics, otherwise, this is a one-to-many relationship)

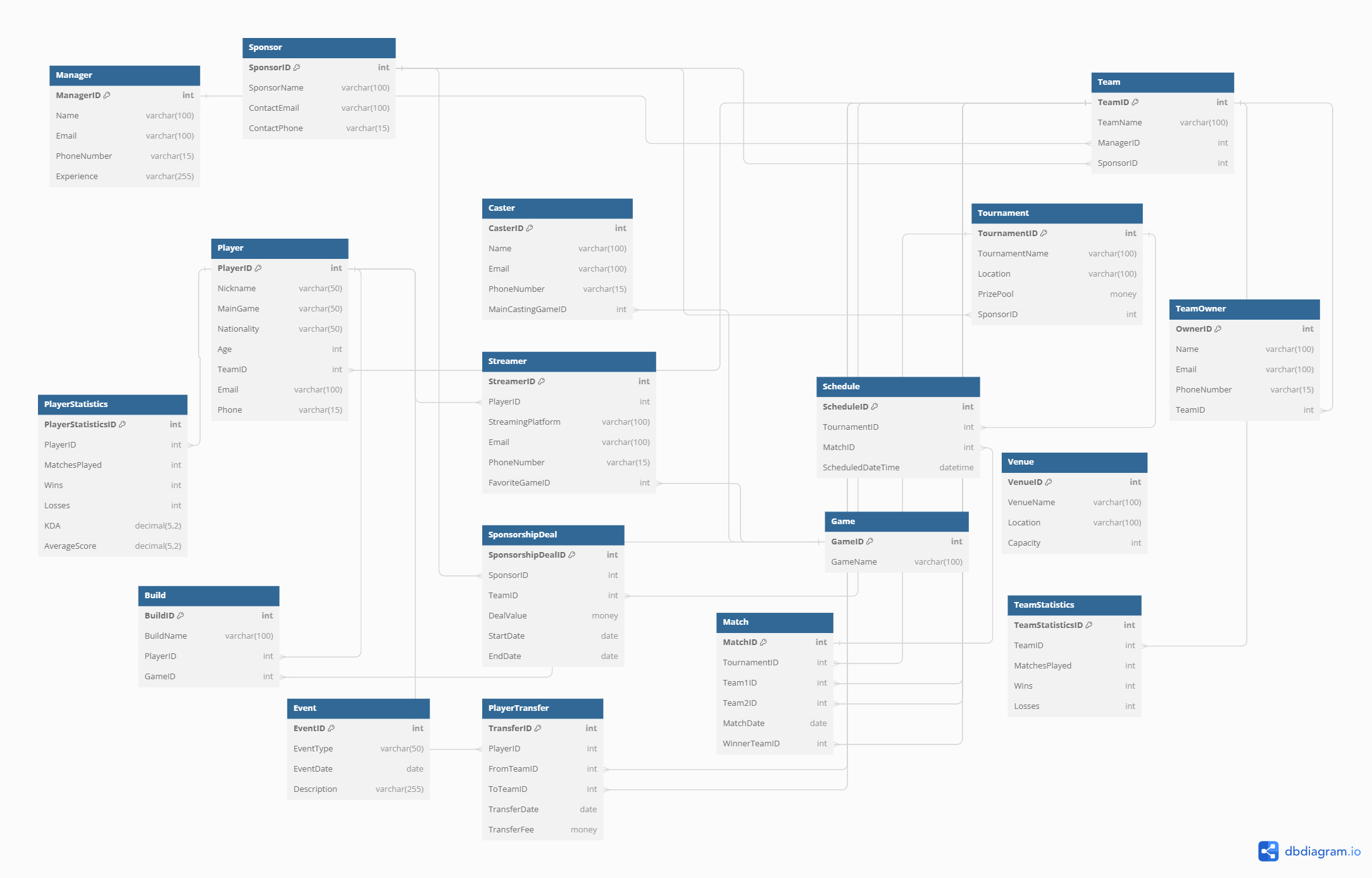
**Event**

1. EventID → EventType, EventDate, Description

**PlayerTransfer**

1. TransferID → PlayerID, FromTeamID, ToTeamID, TransferDate, TransferFee
2. PlayerID, FromTeamID, ToTeamID → TransferID (if the combination of player and teams uniquely identifies a transfer)

*Database Schema:*

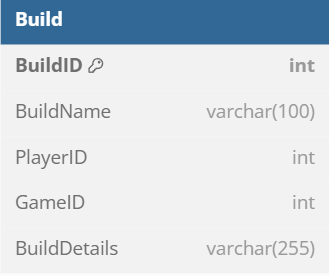


1. **Normalization Examples**

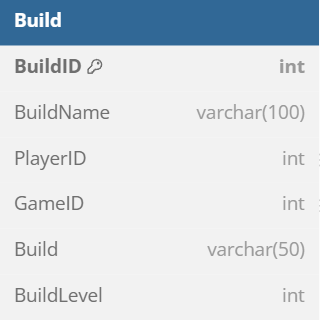
*Example 1:*

For the first example let’s take a look at Build Table,

Original Table

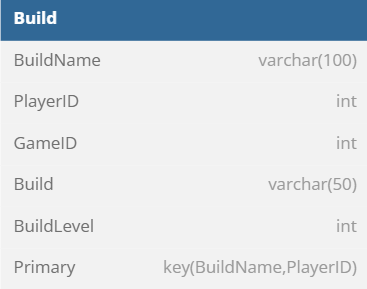


Step 1: **Normalize to 1NF**



Step 2: **Normalize to 2NF**

Partial dependencies occur when non-key attributes depend on part of a composite primary key. However, here BuildID is a single-column primary key, so I need to introduce a composite key scenario to demonstrate 2NF.

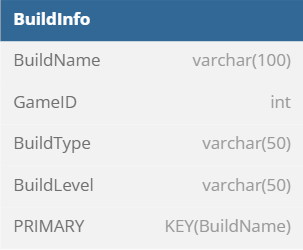


**Identify Partial Dependencies:**

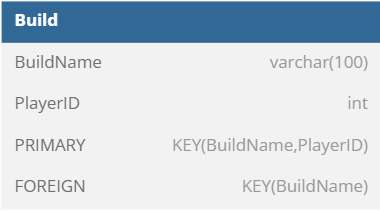
* GameID is dependent on BuildName
* BuildType and BuildLevel are dependent on BuildName

**Remove Partial Dependencies:**

1. Create a new table for Build Info



1. Update Build table to Reference BuildInfo



Step 3: **Normalize to 3NF**

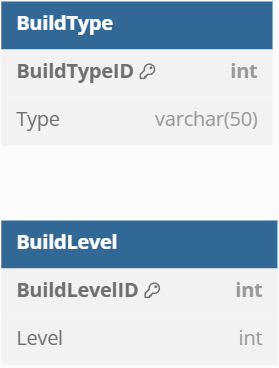
Let’s remove transitive dependencies. A transitive dependency exists when non-key attributes depend on other non-key attributes.

**Identify Transitive Dependencies:**

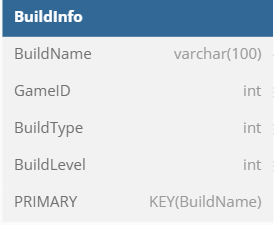
* BuildType and BuildLevel may have additional details (descriptions, etc) that do not directly depend on BuildName.

**Remove Transitive Dependencies:**

1. Create tables for BuildType and BuildLevel



1. Update BuildInfo Table to Reference BuildType and BuildLevel



Step 4: **Normalize to Boyce-Codd Normal Form (BCNF)**

To achieve BCNF, we must ensure that for every functional dependency X→Y, X is a superkey.

With updated tables this condition is already met.

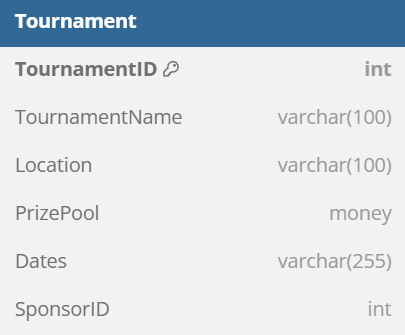
**Summary**

1. **Original Table (1NF):** The table had non-atomic values in BuildDetails.
2. **1NF:** Split BuildDetails into BuildType and BuildLevel.
3. **2NF:** Removed partial dependencies by splitting the table into Build and BuildInfo.
4. **3NF:** Removed transitive dependencies by creating BuildType and BuildLevel tables.
5. **BCNF:** Ensured that all functional dependencies have superkeys as their determinants.

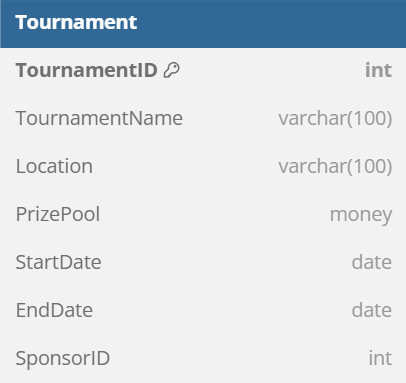
*Example 2:*

Let’s take a look at Tournament Table,

Original Table

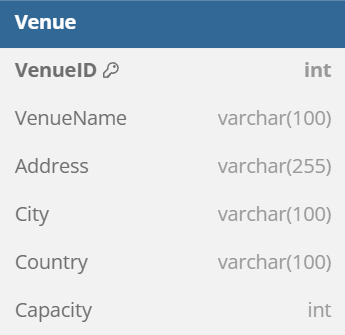


Step 1: **Normalize to 1NF**



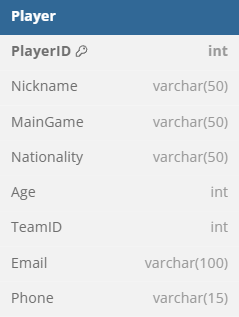
Step 2: **Normalize to 2NF** the table is already in 2NF as all non-key columns depend on the primary key.

Step 3: **Normalize to 3NF** Merge Location and Venue tables to avoid redundancy.



*Example 3:*

Let’s take a look at Player Table,



Step 1: **Normalize to 1NF**

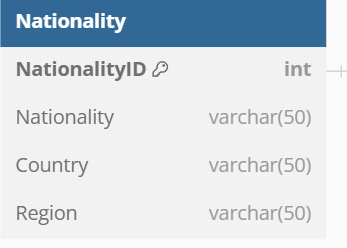
It meets the requirements of 1NF already

Step 2: **Normalize to 2NF**

The table is already in 2NF as all non-key columns depend on the primary key.

Step 3: **Normalize to 3NF**

Create a new Nationality table if more detailed information is needed.



1. **Implemented Database In Microsoft SQL**

Whole design is moved to functional SQL environment now. I have used various queries to build my Database.

Database Creation:

-- Create the ESportsDB database

CREATE DATABASE ESportsDB;

GO

-- Use the ESportsDB database

USE ESportsDB;

GO

I have changed some tables and structures on the way to fulfill my user requirements.

Example Table Creation:

-- Player Table

CREATE TABLE Player (

PlayerID INT PRIMARY KEY IDENTITY,

Nickname NVARCHAR(50) NOT NULL,

MainGame NVARCHAR(50) NOT NULL,

NationalityID INT,

Age INT,

TeamID INT NULL,

Email NVARCHAR(100),

Phone NVARCHAR(15),

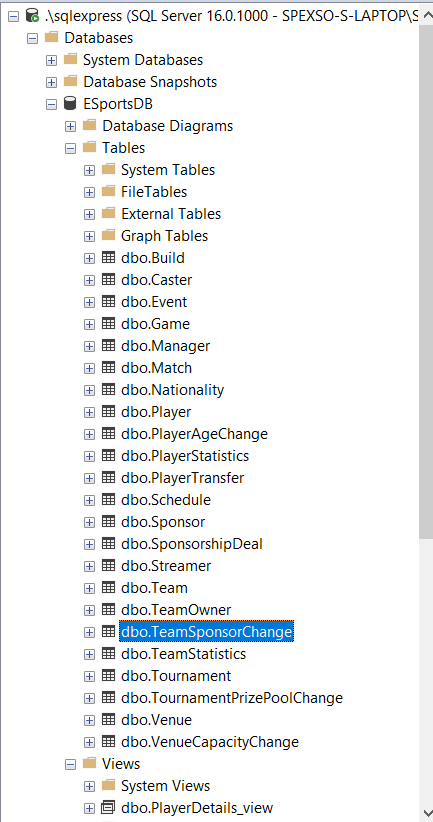
MarketValue int,

FOREIGN KEY (TeamID) REFERENCES Team(TeamID),

FOREIGN KEY (NationalityID) REFERENCES Nationality(NationalityID)

);

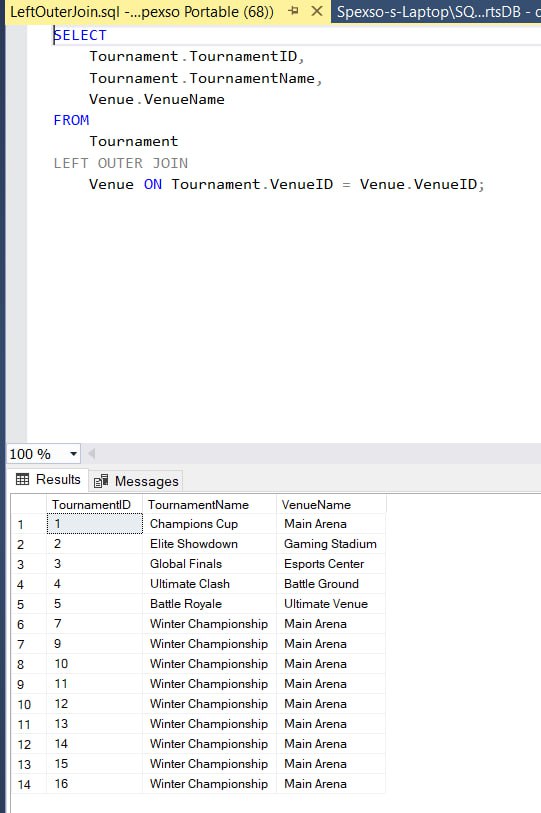
Database Structure:



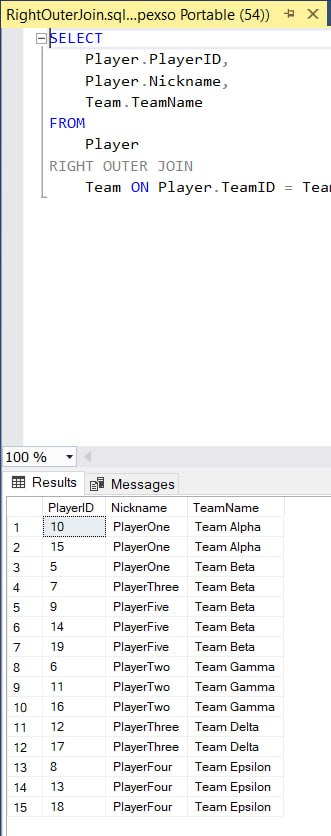
1. **Built Queries**

*Joins*

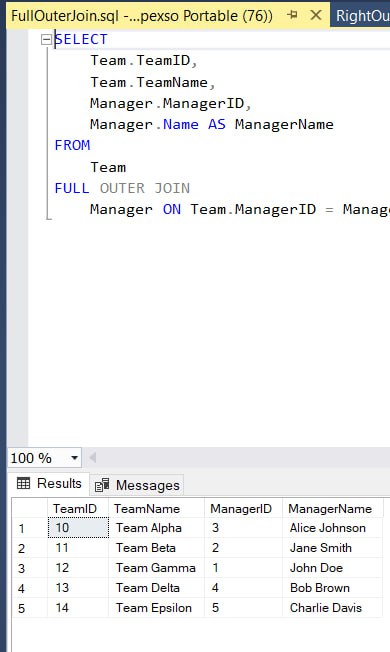
*Left Outer Join:*



*Right Outer Join:*



*Full Outer Join:*



*Triggers*

*Query:*

CREATE TRIGGER PlayerAgeChange\_trigger

ON Player

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

IF EXISTS (SELECT 1 FROM inserted i JOIN deleted d ON i.PlayerID = d.PlayerID WHERE i.Age <> d.Age)

BEGIN

INSERT INTO PlayerAgeChange (PlayerID, OldAge, NewAge, ChangeDate)

SELECT d.PlayerID, d.Age, i.Age, GETDATE()

FROM inserted i

JOIN deleted d ON i.PlayerID = d.PlayerID

WHERE i.Age <> d.Age;

END

END;

*Output:*



*Query:*

CREATE TRIGGER PlayerTransfer\_trigger

ON Player

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

-- Check if the player's team has changed

IF EXISTS (SELECT 1 FROM inserted i JOIN deleted d ON i.PlayerID = d.PlayerID WHERE i.TeamID <> d.TeamID)

BEGIN

-- Insert the transfer details into PlayerTransfer

INSERT INTO PlayerTransfer (PlayerID, FromTeamID, ToTeamID, TransferDate, TransferFee)

SELECT d.PlayerID, d.TeamID, i.TeamID, GETDATE(), i.MarketValue

FROM inserted i

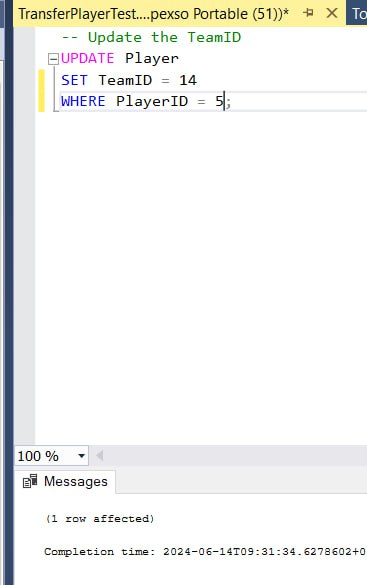
JOIN deleted d ON i.PlayerID = d.PlayerID

WHERE i.TeamID <> d.TeamID;

END

END;

*Output:*



*Query:*

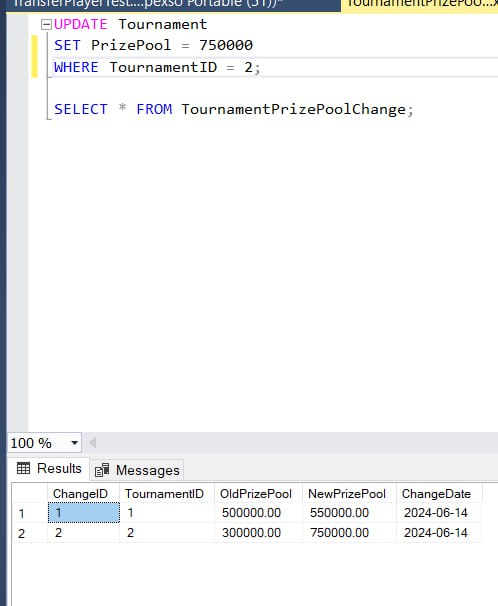
UPDATE Tournament

SET PrizePool = 550000

WHERE TournamentID = 1;

SELECT \* FROM TournamentPrizePoolChange;

*Output:*



*Query:*

CREATE TRIGGER TeamSponsorChange\_trigger

ON Team

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

IF EXISTS (SELECT 1 FROM inserted i JOIN deleted d ON i.TeamID = d.TeamID WHERE i.SponsorID <> d.SponsorID)

BEGIN

INSERT INTO TeamSponsorChange (TeamID, OldSponsorID, NewSponsorID, ChangeDate)

SELECT d.TeamID, d.SponsorID, i.SponsorID, GETDATE()

FROM inserted i

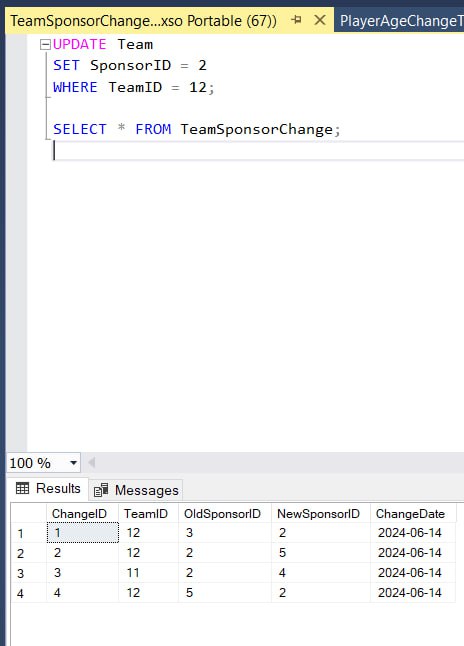
JOIN deleted d ON i.TeamID = d.TeamID

WHERE i.SponsorID <> d.SponsorID;

END

END;

*Output:*



*Query:*

CREATE TRIGGER VenueCapacityChange\_trigger

ON Venue

AFTER UPDATE

AS

BEGIN

SET NOCOUNT ON;

IF EXISTS (SELECT 1 FROM inserted i JOIN deleted d ON i.VenueID = d.VenueID WHERE i.Capacity <> d.Capacity)

BEGIN

INSERT INTO VenueCapacityChange (VenueID, OldCapacity, NewCapacity, ChangeDate)

SELECT d.VenueID, d.Capacity, i.Capacity, GETDATE()

FROM inserted i

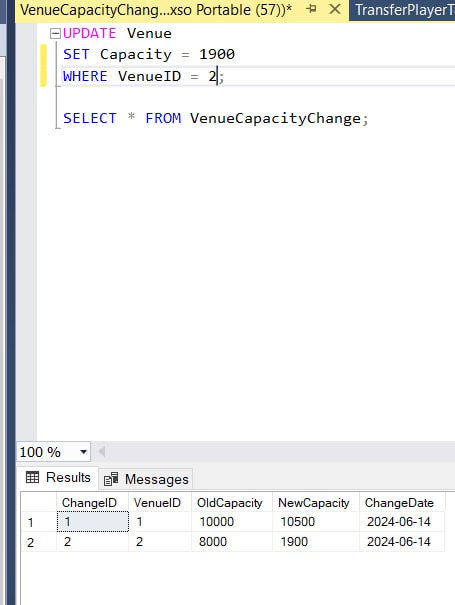
JOIN deleted d ON i.VenueID = d.VenueID

WHERE i.Capacity <> d.Capacity;

END

END;

*Output:*



*Query:*

CREATE TRIGGER DeletePlayer\_trigger

ON Player

AFTER DELETE

AS

BEGIN

SET NOCOUNT ON;

-- Delete related records

DELETE FROM PlayerStatistics

WHERE PlayerID IN (SELECT PlayerID FROM DELETED);

DELETE FROM PlayerAgeChange

WHERE PlayerID IN (SELECT PlayerID FROM DELETED);

DELETE FROM Build

WHERE PlayerID IN (SELECT PlayerID FROM DELETED);

DELETE FROM Streamer

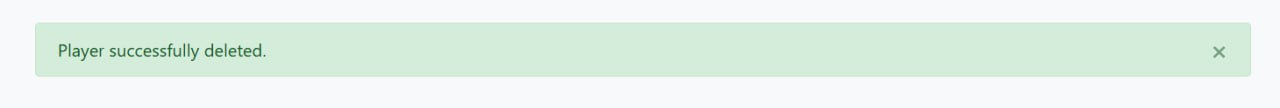
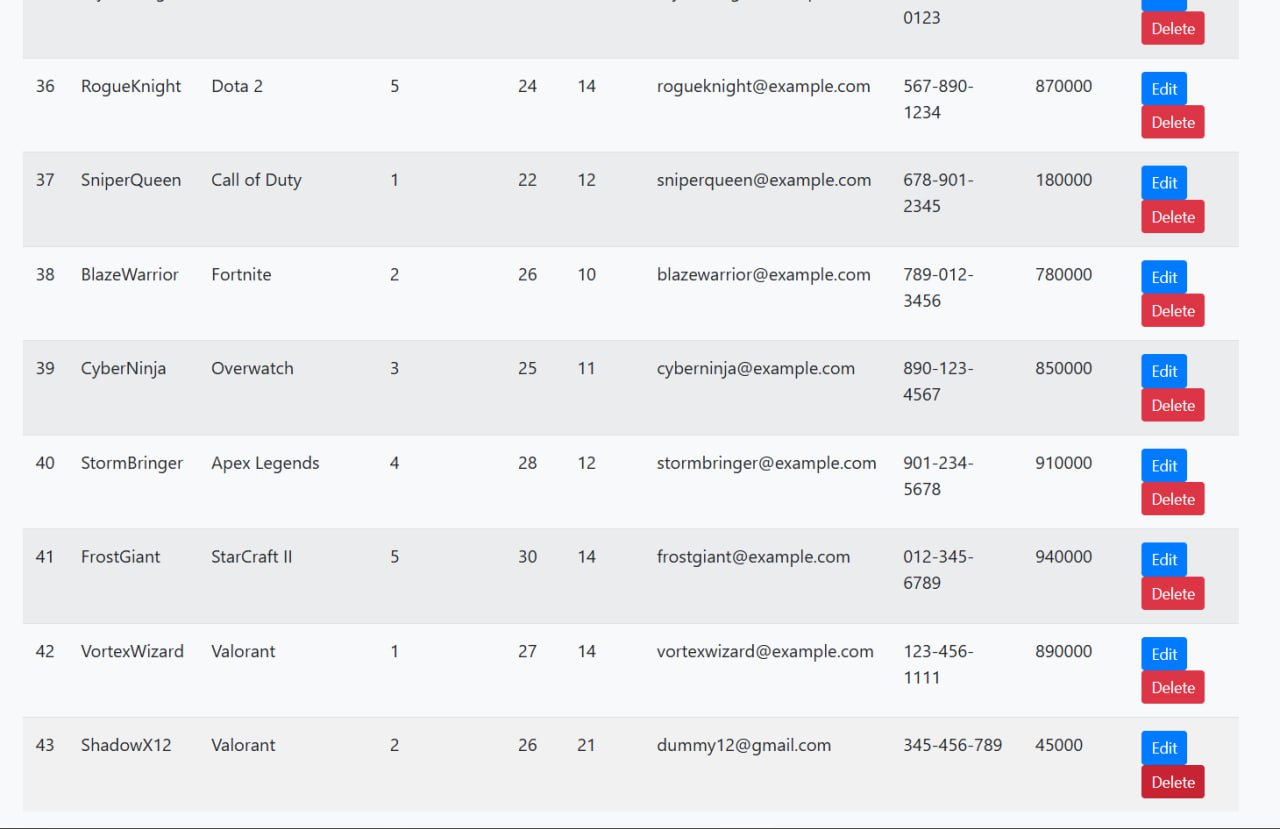
WHERE PlayerID IN (SELECT PlayerID FROM DELETED);

END;

*Output:*

*Before deletion*

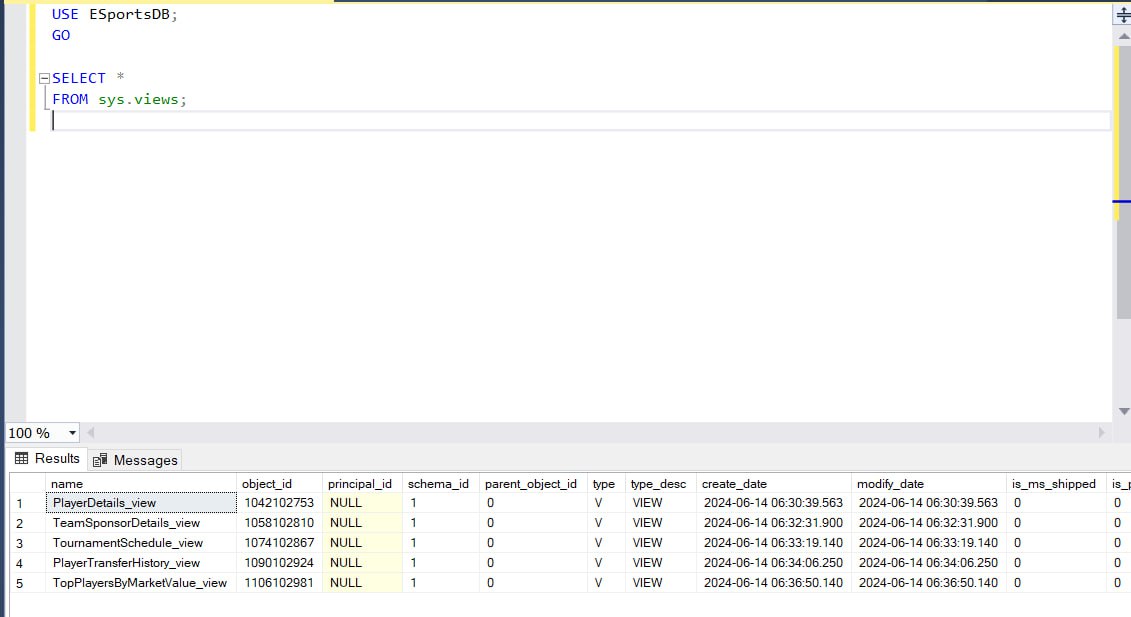
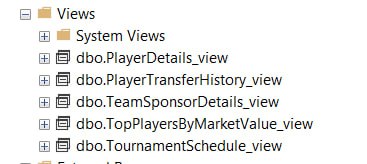
*Delete notification*



*After Deletion*



*Views*



*One of the queries:*

*CREATE VIEW PlayerTransferHistory\_view AS*

*SELECT*

*pt.TransferID,*

*p.Nickname AS PlayerName,*

*pt.FromTeamID,*

*ft.TeamName AS FromTeam,*

*pt.ToTeamID,*

*tt.TeamName AS ToTeam,*

*pt.TransferDate,*

*pt.MarketValue*

*FROM*

*PlayerTransfer pt*

*LEFT JOIN*

*Player p ON pt.PlayerID = p.PlayerID*

*LEFT JOIN*

*Team ft ON pt.FromTeamID = ft.TeamID*

*LEFT JOIN*

*Team tt ON pt.ToTeamID = tt.TeamID;*

*Transactions*

*Add\_Tournament*

*BEGIN TRANSACTION;*

*BEGIN TRY*

*-- Create the new tournament*

*INSERT INTO Tournament (TournamentName, VenueID, PrizePool, SponsorID, StartDate, EndDate)*

*VALUES (:tournament\_name, :venue\_id, :prize\_pool, :sponsor\_id, :start\_date, :end\_date);*

*DECLARE @NewTournamentID INT;*

*SET @NewTournamentID = SCOPE\_IDENTITY();*

*-- Schedule the initial match for the new tournament*

*INSERT INTO Match (TournamentID, Team1ID, Team2ID, MatchDate, WinnerTeamID)*

*VALUES (@NewTournamentID, :team1\_id, :team2\_id, :match\_date, NULL);*

*-- Commit the transaction if both operations succeed*

*COMMIT TRANSACTION;*

*END TRY*

*BEGIN CATCH*

*-- Rollback the transaction if any operation fails*

*ROLLBACK TRANSACTION;*

*-- Optionally throw the error*

*THROW;*

*END CATCH;*

The other 2 transactions not implemented in UI So they are just with static data

*Update\_Players\_Team*

*BEGIN TRANSACTION;*

*BEGIN TRY*

*-- Update the player's team*

*UPDATE Player*

*SET TeamID = 11, MarketValue = 550000*

*WHERE PlayerID = 5;*

*-- Log the transfer*

*INSERT INTO PlayerTransfer (PlayerID, FromTeamID, ToTeamID, TransferDate, MarketValue)*

*VALUES (5, 10, 11, GETDATE(), 550000);*

*-- Commit the transaction if both operations succeed*

*COMMIT TRANSACTION;*

*END TRY*

*BEGIN CATCH*

*-- Rollback the transaction if any operation fails*

*ROLLBACK TRANSACTION;*

*-- Optionally throw the error*

*THROW;*

*END CATCH;*

*Manager\_Assignment\_to\_New\_Team*

*BEGIN TRANSACTION;*

*BEGIN TRY*

*-- Assign the new manager to the team*

*UPDATE Team*

*SET ManagerID = 3*

*WHERE TeamID = 10;*

*-- Update the manager's experience*

*UPDATE Manager*

*SET Experience = '6 years'*

*WHERE ManagerID = 3;*

*-- Commit the transaction if both operations succeed*

*COMMIT TRANSACTION;*

*END TRY*

*BEGIN CATCH*

*-- Rollback the transaction if any operation fails*

*ROLLBACK TRANSACTION;*

*-- Optionally throw the error*

*THROW;*

*END CATCH;*

*Privileges*

*Creating Roles*

-- Create AdminRole

CREATE ROLE AdminRole;

-- Create ManagerRole

CREATE ROLE ManagerRole;

-- Create ViewerRole

CREATE ROLE ViewerRole;

*Granting Roles*

-- Grant all privileges to AdminRole

GRANT CONTROL ON DATABASE::ESportsDB TO AdminRole;

-- Grant privileges to manage teams

GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.Team TO ManagerRole;

-- Grant privileges to manage players

GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.Player TO ManagerRole;

-- Grant privileges to manage matches

GRANT SELECT, INSERT, UPDATE, DELETE ON dbo.Match TO ManagerRole;

-- Grant read-only access to all tables

GRANT SELECT ON dbo.Manager TO ViewerRole;

GRANT SELECT ON dbo.Sponsor TO ViewerRole;

GRANT SELECT ON dbo.Game TO ViewerRole;

GRANT SELECT ON dbo.Team TO ViewerRole;

GRANT SELECT ON dbo.Nationality TO ViewerRole;

GRANT SELECT ON dbo.Player TO ViewerRole;

GRANT SELECT ON dbo.Caster TO ViewerRole;

GRANT SELECT ON dbo.Venue TO ViewerRole;

GRANT SELECT ON dbo.Tournament TO ViewerRole;

GRANT SELECT ON dbo.TeamOwner TO ViewerRole;

GRANT SELECT ON dbo.Build TO ViewerRole;

GRANT SELECT ON dbo.Streamer TO ViewerRole;

GRANT SELECT ON dbo.Match TO ViewerRole;

GRANT SELECT ON dbo.Schedule TO ViewerRole;

GRANT SELECT ON dbo.SponsorshipDeal TO ViewerRole;

GRANT SELECT ON dbo.PlayerStatistics TO ViewerRole;

GRANT SELECT ON dbo.TeamStatistics TO ViewerRole;

GRANT SELECT ON dbo.Event TO ViewerRole;

GRANT SELECT ON dbo.PlayerTransfer TO ViewerRole;

*Concurrency*

*\*\*Not implemented in UI so used with just static data*

*Manager Assignemnt To New Team*

SET TRANSACTION ISOLATION LEVEL READ COMMITTED;

BEGIN TRANSACTION;

BEGIN TRY

-- Acquire a lock on the team row

SELECT \* FROM Team WITH (UPDLOCK) WHERE TeamID = 1;

-- Assign the new manager to the team

UPDATE Team

SET ManagerID = 3

WHERE TeamID = 1;

-- Acquire a lock on the manager row

SELECT \* FROM Manager WITH (UPDLOCK) WHERE ManagerID = 3;

-- Update the manager's experience

UPDATE Manager

SET Experience = '6 years'

WHERE ManagerID = 3;

-- Commit the transaction if both operations succeed

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback the transaction if any operation fails

ROLLBACK TRANSACTION;

-- Optionally, throw the error

THROW;

END CATCH;

*Player Transfer Updates Transfers Table*

SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;

BEGIN TRANSACTION;

BEGIN TRY

-- Acquire a lock on the player row

SELECT \* FROM Player WITH (UPDLOCK, HOLDLOCK) WHERE PlayerID = 8;

-- Update the player's team and market value

UPDATE Player

SET TeamID = 12, MarketValue = 10000

WHERE PlayerID = 25;

-- Log the transfer

INSERT INTO PlayerTransfer (PlayerID, FromTeamID, ToTeamID, TransferDate, MarketValue)

VALUES (25, 12, 13, GETDATE(), 550000);

-- Commit the transaction if both operations succeed

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback the transaction if any operation fails

ROLLBACK TRANSACTION;

-- Optionally, throw the error

THROW;

END CATCH;

*Tournament Creation Creates a Match*

SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;

BEGIN TRANSACTION;

BEGIN TRY

-- Create the new tournament

INSERT INTO Tournament (TournamentName, VenueID, PrizePool, SponsorID, StartDate, EndDate)

VALUES ('Winter Championship', 1, 300000, 1, '2024-12-01', '2024-12-10');

DECLARE @NewTournamentID INT;

SET @NewTournamentID = SCOPE\_IDENTITY();

-- Ensure that no other transactions modify the matches for this tournament

SELECT \* FROM Match WITH (UPDLOCK, HOLDLOCK) WHERE TournamentID = @NewTournamentID;

-- Schedule the initial match for the new tournament

INSERT INTO Match (TournamentID, Team1ID, Team2ID, MatchDate, WinnerTeamID)

VALUES (@NewTournamentID, 10, 12, '2024-12-02', NULL);

-- Commit the transaction if both operations succeed

COMMIT TRANSACTION;

END TRY

BEGIN CATCH

-- Rollback the transaction if any operation fails

ROLLBACK TRANSACTION;

-- Optionally, throw the error

THROW;

END CATCH;

1. **User Interface**

I have designed my User interface for Administrators use only. Any other user should not be able to access it.

I have used python with Flask for building a local website with the help of html.

*App.py*

**Key Routes and Their Functions**

1. **Home Page**
   * **Route:** /
   * **Function:** Displays lists of all teams and players.
   * **Template:** index.html
2. **Add Manager**
   * **Route:** /add\_manager
   * **Methods:** GET, POST
   * **Function:** Adds a new manager to the database.
   * **Template:** add\_manager.html
3. **Team Sponsor Changes**
   * **Route:** /team\_sponsor\_changes
   * **Function:** Displays a list of team sponsor changes.
   * **Template:** team\_sponsor\_changes.html
4. **Add Team**
   * **Route:** /add\_team
   * **Methods:** GET, POST
   * **Function:** Adds a new team to the database.
   * **Template:** add\_team.html
5. **Add Player**
   * **Route:** /add\_player
   * **Methods:** GET, POST
   * **Function:** Adds a new player to the database.
   * **Template:** add\_player.html
6. **Add Tournament**
   * **Route:** /add\_tournament
   * **Methods:** GET, POST
   * **Function:** Adds a new tournament and schedules an initial match.
   * **Template:** add\_tournament.html
7. **List Matches**
   * **Route:** /matches
   * **Function:** Displays a list of matches.
   * **Template:** matches.html
8. **Player Details**
   * **Route:** /player\_details
   * **Function:** Displays detailed information about players.
   * **Template:** player\_details.html
9. **Team Manager Details**
   * **Route:** /team\_manager\_details
   * **Function:** Displays details of teams and their managers.
   * **Template:** team\_manager\_details.html
10. **Team Sponsor Details**
    * **Route:** /team\_sponsor\_details
    * **Function:** Displays details of teams and their sponsors.
    * **Template:** team\_sponsor\_details.html
11. **Edit Team**
    * **Route:** /edit\_team/<int:id>
    * **Methods:** GET, POST
    * **Function:** Edits the details of a team.
    * **Template:** edit\_team.html
12. **Player Transfer List**
    * **Route:** /player\_transfer\_list
    * **Function:** Displays a list of player transfers.
    * **Template:** player\_transfer\_list.html
13. **Edit Player**
    * **Route:** /edit\_player/<int:id>
    * **Methods:** GET, POST
    * **Function:** Edits the details of a player.
    * **Template:** edit\_player.html
14. **Delete Player**
    * **Route:** /delete\_player/<int:id>
    * **Function:** Deletes a player and their related records from the database.

*Model.py*

Models of tables in database

**Manager**

Purpose: Stores information about team managers, including their contact details and experience.

**Sponsor**

Purpose: Keeps track of sponsors who financially support teams and tournaments, including their contact information.

**Game**

Purpose: Contains a list of games that are played in the eSports tournaments.

**Team**

Purpose: Holds details about the teams participating in tournaments, including references to their managers and sponsors.

**Nationality**

Purpose: Manages information about different nationalities, including country and region details.

**Player**

Purpose: Stores information about the players, including their team affiliations, contact details, and market value.

**Caster**

Purpose: Contains information about casters (commentators) for the games, including their contact details and the main game they cast.

**Venue**

Purpose: Holds details about venues where tournaments are held, including location and capacity information.

**Tournament**

Purpose: Manages data about tournaments, including their names, venues, prize pools, sponsors, and dates.

**TeamOwner**

Purpose: Stores information about the owners of the teams, including their contact details.

**Build**

Purpose: Contains details about player builds for different games, associating players with specific game configurations.

**Streamer**

Purpose: Manages information about players who also stream games, including their streaming platform and contact details.

**Match**

Purpose: Stores information about matches in tournaments, including participating teams, match dates, and the winner.

**Schedule**

Purpose: Contains scheduling details for matches in tournaments, including date and time.

**SponsorshipDeal**

Purpose: Manages details about sponsorship deals between sponsors and teams, including deal values and duration.

**PlayerStatistics**

Purpose: Holds statistical data about players' performance in matches, such as matches played, wins, losses, and average scores.

**TeamStatistics**

Purpose: Manages statistical information about team performance in matches.

**Event**

Purpose: Stores information about various events related to the eSports ecosystem, including their type, date, and description.

**PlayerTransfer**

Purpose: Contains details about player transfers between teams, including transfer fees and dates.