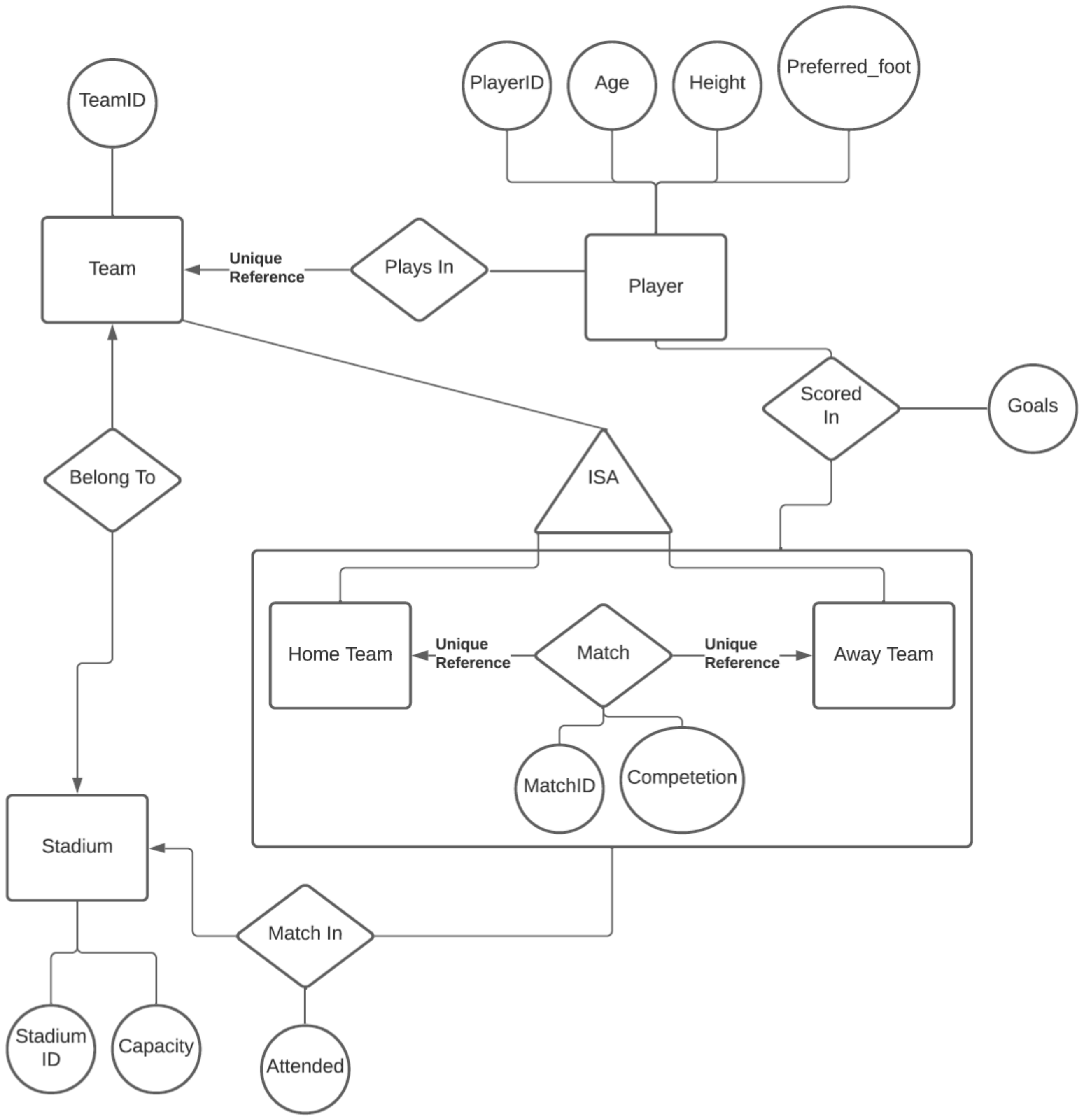
Maman HW2 Dry Assignment

Omer Meshulam 205949886

Yaniv Holder 207025297

Our design decisions were derived from the following ERD table we sketched up:



Our Tables & views are as drawn in the ERD and detailed below:
Underline depicts Primary Table key

Tables

Teams

TeamID	(Unique Positive Integer)

Explanation: ERD, ID is the only feature of a Team

Players

PlayerID	(Unique Positive Integer)
TeamID	(FK of Teams)
Age	(Positive Integer)
Height	(Positive Integer)
Preferred_foot	(String in {"Left", "Right"})

Explanation: ERD, features of a Player

Stadiums

StadiumID	(Unique Positive Integer)
Capacity	(Positive Integer)
BelongsTo	(FK of Teams)

Explanation: ERD, decided to add BelongTo to this table since it's a FK of Teams, and required when inserting a stadium.

Matches

MatchID	(Unique Positive Integer)
Competition	(String in {"International", "Domestic"})
Home	(FK of Teams)
Away	(FK of Teams, different then Home)

Explanation: ERD, features of a match.

ScoredIn

MatchID	(FK of Matches)
PlayerID	(FK of Players)
Goals	(Unique Positive Integer)

Explanation: ERD, only after adding a player and a match a player can score in a match. Therefore, we decided to have a separate table to track all goals that were scored by a player in a specific game. Also, the fact that from the assignment instructions, we can only have one entry of goals per couple of match and player, it made sense to make a table with those fields as a tuple of the primary keys.

MatchIn

<u>MatchID</u>	(FK of Matches)
StadiumID	(FK of Stadiums)
Attended	(Unique Integer greater-equal to 0)

Explanation: ERD, a stadium where a match took place & the number of attendees that participated is a standalone property of a match and a stadium, and since its possible for a match to not take place in a specific stadium, we decided to have an additional table for this relation.

<u>Views</u>

MatchAndTotalGoals

<u>MatchID</u>	(FK of Matches)
TotalGoals	(Unique positive integer, represents total goals scored in the match)

Explanation: In order to track the total number of goals per match, we decided to create a view that will hold the total number of goals per match.

ActiveTallTeams

<u>TeamID</u>	(FK of Matches, Active Tall Team as defined in hw2.pdf)

Explanation: To track the current active tall teams, we decided to create a view that will hold their ID's. It's useful for two separate queries that we run: ActiveTallTeams and ActiveTallRichTeams.

RichTeams

<u>TeamID</u>	(FK of Matches, Rich Team as defined in hw2.pdf)

Explanation: To track the current active rich teams, we decided to create a view that will hold their ID's.

Views were created to make the code clearer and more readable, and also prevent code duplication.

ValidTallPeople

<u>PlayerID</u>	(FK of Players)
TeamID	(FK of Teams)

Explanation: View of all the players that are tall and their team played at least one home/away game. From Players, Matches, we filter player height > 190 and player's team played home/away.

Views were created to make the code clearer and more readable, and also prevent code duplication.

API Implementation

Please note that in all function API's, if we encounter errors during the execution, we return it to the user as the correct error as defined in the exercise.

createTables()

Initializes the tables and the views presented in the previous section using CREATE TABLE and CREATE VIEW.

clearTables()

Deletes all entries from Teams, Players and Stadiums. The delete will cascade and delete all entries references to these entities.

dropTables()

Drops all the tables presented in the previous section and applies on delete cascade to delete all entries that reference the objects in the table we drop.

addTeam(teamID: int)

Using the command INSERT, we insert into teams the TeamID we are given. Due to unique TeamID and positive value restrictions, we enforce that the user cannot enter a team identified by a negative integer or that is already in the database.

addMatch(match: Match)

Using the command INSERT, we insert into Matches the new match entry with all the parameters that define the match we are given. Due to unique MatchID, positive value and foreign key restrictions (and an additional check that Home <> Away), we enforce the restrictions provided to us by the exercise.

getMatchProfile(matchID: int)

Selects the entry from Matches where the MatchID is equal to the matchID requested (unique entry) and returns it.

deleteMatch(match: Match)

Deletes from matches where MatchID is equal to the match requested. The delete cascades to the rest of the tables and deletes all entries that reference the match we deleted.

addPlayer(player: Player)

Using the command INSERT, we insert into Players the new player entry with all the parameters that define the player we are given. Due to unique PlayerID, positive value checks, foreign key restrictions and validation on the input for the preferred foot, we enforce the restrictions provided to us by the exercise.

getPlayerProfile(playerID: int)

Selects the entry from Matches where the MatchID is equal to the matchID requested (unique entry) and returns it.

deletePlayer(player: Player)

Deletes from Players the (unique) entry where PlayerID is equal to the match requested. The delete cascades to the rest of the tables and deletes all entries that reference the player we deleted.

addStadium(stadium: Stadium)

Using the command INSERT, we insert into Stadiums the new stadium entry with all the parameters that define the stadium we are given. Due to unique StadiumID, positive value checks and foreign key restrictions we enforce the restrictions provided to us by the exercise.

getStadiumProfile(stadiumID: int)

Selects the entry from Stadiums where the StadiumID is equal to the stadiumID requested (unique entry) and returns it.

deleteStadium(stadium: Stadium)

Deletes from Stadiums the (unique) entry where StadiumID is equal to the stadiumID requested. The delete cascades to the rest of the tables and deletes all entries that reference the stadium we deleted.

playerScoredInMatch(match: Match, player: Player, amount: int)

Using the command INSERT, we insert into ScoredIn an entry with the match the player scored in, the player ID and the number of goals he scored. Due to unique (MatchID, PlayerID) tuple, and positive value checks for the number of goals scored, we enforce entering only a positive number of goals scored per player and restrict a single entry that holds all the goals a player scored In a specific match.

playerDidntScoreInMatch(match: Match, player: Player)

Deletes from ScoredIn the (unique) entry where MatchID and PlayerID are equal to the matchID and playerID requested.

matchInStadium(match: Match, stadium: Stadium, attendance: int)

Using the command INSERT, we insert into MatchIn a new entry with the MatchID, the Stadium the match took place and the number of attendees provided. Due to unique StadiumID, positive value checks and foreign key restrictions we enforce the restrictions provided to us by the exercise.

matchNotInStadium(match: Match, stadium: Stadium)

Deletes from MatchIn the (unique) entry where MatchID and PlayerID are equal to the matchID and playerID requested. If the match takes place in a different stadium then provided or the entry doesn’t exist, returns the corresponding error.

averageAttendanceInStadium(stadiumID: int)

From MatchIn table, we select only rows with the stadiumID requested and aggregate using the AVG function on attended attribute and return the result.

stadiumTotalGoals(stadiumID: int)

Using a subquery, we first create a table called StadMatches, which contains all the matches that occurred in the requested stadium (We query MatchIn where StadiumID is equal to the stadium requested). Then we Inner Join StadMatches and ScoredIn on equal MatchID. We aggregate using the SUM function on Goals (from ScoredIn) attribute in order to calculate the total amount of goals that took place in the stadium and return the result.

playerIsWinner(playerID: int, matchID: int)

We Inner Join MatchAndTotalGoals and ScoredIn on equal MatchID. Then we select only rows with the matchID and playerID on the condition that the players goals in the match are greater or equal to half the total goals scored in the match. We return false if the query returns empty, otherwise the player fulfills the condition, and we return true.

getActiveTallTeams()

We order ActiveTallTeams view by TeamID descending, limit the output to 5 and return the result.

getActiveTallRichTeams()

We calculate the intersection between ActiveTallTeams view and RichTeams view, order by TeamID ascending, limit the output to 5 and return the result.

popularTeams()

We will first explain the subquery tables we create:

PopularHome: Table with teams that played at least one home match and are popular

Columns: Home (TeamID)
Calculated by left joining Matches and MatchIn on MatchID, grouping by Home and aggregating using MIN(Case Attended > 40000 then 1 else 0). This trick allows us to give true to the Home team only if all entries have Attended >= 40000. We then select where Popular=1.

BooleanNeverPlayedHome: Teams that never played as home.

Columns: Home (FK TeamID), NotHome (Boolean – 1 if never played home, otherwise 0)
Similar to PopularHome, calculated by left joining Teams and Matches on TeamID, grouping by TeamID and aggregating using MIN(Case Home is Null then 1 else 0). This trick allows us to give true to teams only if all home entries are null – meaning they never played a home game.

We use Union on Popular home and on BooleanNeverPlayedHome on the condition that NotHome = 1. The table we receive is the group of teams that in all of their home games they had over 40k attended. We order the output by TeamID descending and return the TeamID's.

getMostAttractiveStadiums()

We will first explain the subquery table we create:

StadiumGoals: Total goals scored in each stadium (StadiumID will not appear in the table if no goals were scored in it)

Columns: StadiumID (FK of Stadiums) and Goals (Total goals scored in the stadium)
Calculated by inner joining MatchAndTotalGoals and MatchIn on MatchID, grouping by StadiumID and aggregating using Sum(MatchAndTotalGoals) as Goals (we also select StadiumID).

We Left join Stadiums with StadiumGoals on stadiumID, select StadiumID and calculate the total goals with Coalesce in order to convert the null values to 0.
We then order by Goals descending and StadiumID ascending and return the output.

mostGoalsForTeam(teamID: int)

We will first explain the subquery tables we create:

Scorers: Contains players that scored goals for the requested team.

Columns: PlayerID (FK of Stadiums) and Goals (Total goals scored by the player)
Calculated by inner joining ScoredIn and Players on PlayerID, grouping by PlayerID and aggregating using Sum(ScoredIn.Goals) as Goals (we also select PlayerID). We finally filter for where the player's team is equal to the requested team.

TeamPlayers: Contains players play for the requested team.

Columns: PlayerID (FK of Stadiums)
We select from Players the player that their TeamID is equal to the requested team, and select their playerID.

We Left join TeamPlayers with Scorers on PlayerID, select PlayerID and calculate the total goals they scored with Coalesce in order to convert the null values to 0 (for players that never scored).
We then order by Goals descending and PlayerID ascending and return the output.

getClosePlayers(playerID: int)

We will first explain the subquery tables we create:

<u>Player_Goal_Matches</u>: Contains all the matches that the requested player scored in.

Columns: MatchID (FK of Matches)
Select of MatchID from ScoredIn where the player is the requested player.

<u>Our_Player_Matches</u>: A single entry – the number of matches the requested player scored in.

Columns: Tot_Player_Matches
Count aggregation from table ScoredIn where the player is the requested player.

<u>Other_Players_Matches</u>: A table that for each player that scored in a match with out player, contains the number of matches they scored together.

Columns: PlayerID, Num_Matches (number of matches the player scored with the requested player)

We Inner join Player_Goal_Matches with ScoredIn on MatchID, filter where playerID is not the requested player. Group by the playerID and aggregate with COUNT to count the number of entries – the number of times the player scored with the requested player.

<u>Player_Pool</u>: An empty table if the requested player doesn't exist, or a table containing all players otherwise

Column: PlayerID
Calculated by Joining Players with itself and selecting only columns where the requested player appears on column 2.

<u>Our_Player_Empty</u>: Total count of our player's Matches, to check if the requested player played no games

Columns: Tot_Player_Matches
Count aggregation from table ScoredIn where the player is the requested player.

We first left join Other_Players_Matches with Our_Player_Matches on true condition (to get all possibilities). We then select lines where Other_Player_Matches.Num_Matches is greater then half of Our_Player_Matches.Tot_Player_Matches. This gives us all the players that are close to the player if the player has played any games.

Since the other possibility is that the player played zero games, we union the results with Player_Pool Join OurPlayerEmpty on the condition that the total player matches that the requested player played is equal to 0. We do this since we would like to receive all players in the pool in case the requested player never played any matches (the condition is fulfilled vacuously). We then filter out the player requested from the results.

This result is correct since only one of the union parts can be fulfilled at any given time on a specific player, and the union gives us the complete result.