CT3532

Database Systems II: Assignment 1

Andrew Hayes
Student ID: 21321503

Conor McNamara
Student ID: 21378116

Contents

1 Database Design 1
1.1 Schema Diagram 1
1.2 The `teams` Table 1
1.3 The `players` Table 2
1.4 The `games` Table 2
1.5 The `results` Table 2
1.6 The `active_players` Table 3
1.7 The `substitutions` Table 4
1.8 The `sendoffs` Table 4
1.9 The `goals` Table 4
1.10 SQL Code to Create Tables 5

2 Query Design 6
2.1 List all players playing for a given team 6
2.2 List all players who have scored in a given game 7
2.3 List the top five goal scorers in the league 8
2.4 List all teams and the amount of points they have so far 8

1 Database Design

1.1 Schema Diagram

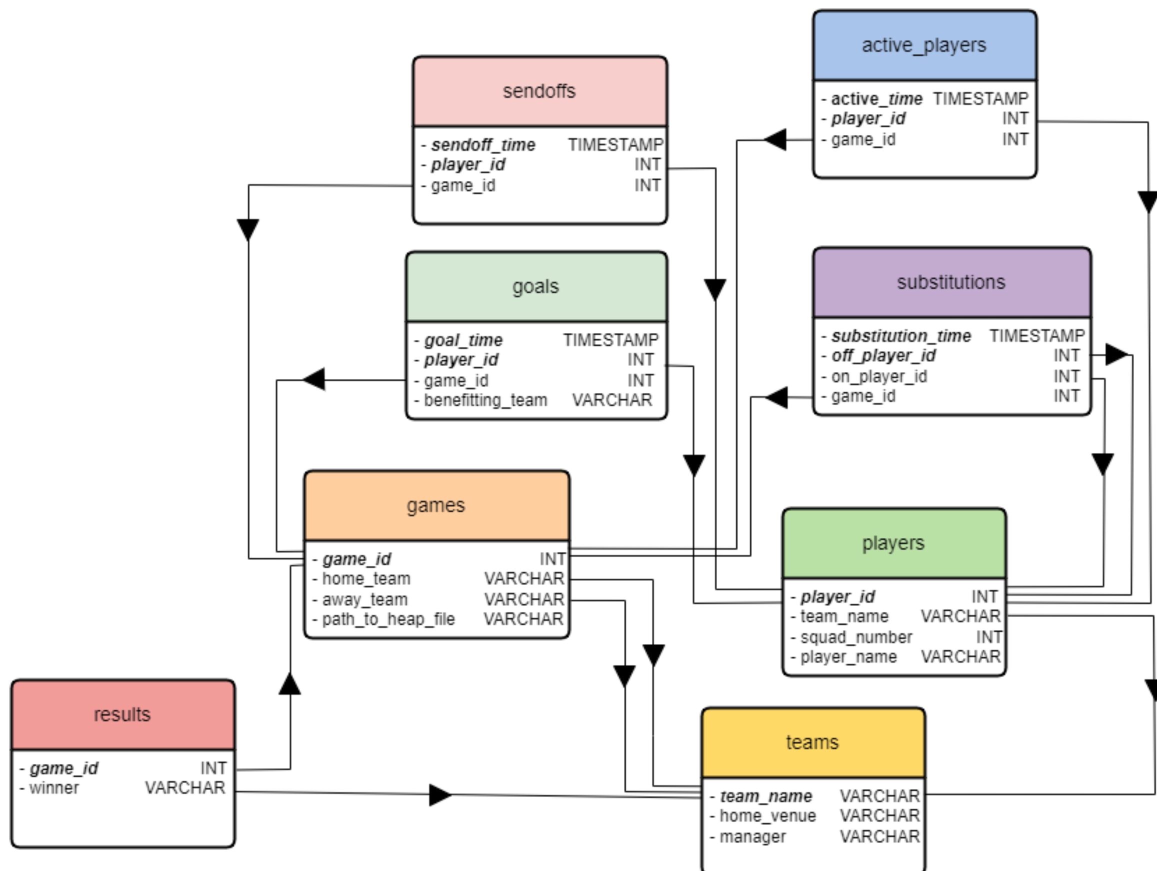

Figure 1: Diagram of the Proposed Database Schema

1.2 The `teams` Table

The `teams` table consists of three columns:

1. The primary key `team_name`, represented as a `VARCHAR`. We assume that team name's are unique, i.e. that there cannot be two teams with the same name in the league. If the league for some reason allowed different teams to share the same name, we could instead use an auto-incrementing integer ID as the primary key for this table. One drawback to not using an auto-incrementing integer team ID is that the `team_name` must be duplicated in every table that has a foreign key reference to this table. This means that to update a team's name, it would have to be changed across multiple tables, potentially leading to update anomalies. There is also the question of whether it is more efficient to use a the natural primary key, or to add an additional column to be the primary key instead: duplicating a `VARCHAR` across several tables is costly, but it is also potentiall wasteful to insert an entirely new column into the table when there is a perfectly valid pre-existing primary key.

   The primary benefit to using the `team_name` as the primary key over an integer ID is that it reduces the need for joins in queries and makes the output of queries more readable. If for example, we wanted to query the names of the teams involved in a given game, we could simply `SELECT` the `team_name` from the `games` table. If we use the integer ID, we would have to join the `games` table to the `teams` table to achieve this.

2. The team's `home_venue`, represented as a `VARCHAR`. We make no assumptions about the uniqueness of a team's home venue, and only assume that each team has only one home venue. Our design allows for two teams to share a home venue with no issues, but does not accommodate for one team having several home venues.

3. The team's `manager`. We store only the name of the manager here as a `VARCHAR`, as we assume that no other information about the manager is known/stored. If more information was known about the managers, this could be replaced with a foreign key to a `managers` table instead.

1.3 The `players` Table

The `players` table consists of four columns:

1. The primary key `player_id`, which is an auto-incrementing integer. We do not use a composite primary key consisting of the `team_name` & `squad_number` columns, although this would be a valid primary key, to prevent having to duplicate both the player's team name and their squad number in each table that references them.

2. The `team_name` of the player, which is a foreign key to the column of the same name in the `teams` table. Here, we assume that each player can play for only one team.

3. The player's `squad_number`, represented as an `INT`, which we assume is unique within the team.

4. The `player_name`,represented as a `VARCHAR`, which we do not assume is unique, unlike with team names.

1.4 The `games` Table

The `games` table consists of four columns:

1. The primary `game_id`, which is an auto-incrementing integer. Since each pair of team plays each other twice, we have no natural candidate key regarding the game, except perhaps the path to the heap file which stores the (x, y) co-ordinates of each player & the football throughout the game. Since there is one of these per game, and they are presumably located at different file paths, this is a candidate key for the table, but it is a poor choice of primary key because this file could potentially be moved or deleted, resulting either in the primary key value having to be changed or a `NULL` primary key value, which is not possible. Furthermore, the filepath could be quite long, and duplicating this long `VARCHAR` in every table that has a foreign key to the `games` table would be quite wasteful.

2. The `home_team`, which is a foreign key to the `team_name` column in `teams` table. This column allows us to infer the location of the game, as it is at the `home_team`'s home grounds, and we are told that each game is played at one of the team's home grounds. This assumes that a team does not change the location of their home grounds, because if the team changes the location of their home grounds in, say, the year 2030, all the games up to that point would no longer be recorded correctly as the location that they were played in, but in the new home grounds of the team.

3. The `away_team`, which is also a foreign key to the `team_name` column `teams` table.

4. The `path_to_heap_file`, stored as a `VARCHAR`, which points to the location in which the data heap file for the game is stored. See above for why this was not chosen as a primary key.

1.5 The `results` Table

The `results` table records the outcome for each game. We chose to store the results in a separate table because of our design choice of how to represent a drawn game. The `results` table consists of two columns:

1. The primary key `game_id` which is a foreign key to the `game_id` column in the `games` table.

2. The `winner` of the game, which is a foreign key to the `team_name` column in the `teams` table. The value of this is `NULL` if the game is drawn, as there is no winner: this is the primary reason why we opted to put the results in their own separate table. If the results were included in the `games` table as a column, there would have to be a value chosen for the result as soon as the game was entered into the table, which is likely before the game has been completed, potentially resulting in a `NULL` value in the results column. This would create a confusion about whether or not the `NULL` value means that there was no winner, i.e. a draw, or if there is on winner *yet*, i.e. the game has not yet finished – the classic issue of "What does `NULL` mean?". This could potentially be avoided by a special pseudo-team to assign wins to in the case of a draw, named something like `draw`, or a Boolean value to indicate whether or not the game was a draw, or perhaps a Boolean value to indicate whether or not a game the game has finished.

We assume that the `results` table is only inserted into upon the completion of a game, so that there is never any ambiguity about whether the `NULL` means that there was a draw or whether it means that the result is not known: data should only ever be inserted when the results are known, so that we can be certain that the `NULL` represents a draw. In practice, using `NULL` to represent a draw could cause a lot of confusion and data may be inserted prematurely, under the assumption that `NULL` indicates that the result is not yet known. Although technically valid, this approach of using `NULL` to represent a known value is not a good one to take in practice.

We do not store the points awarded to each team depending on the outcome, as this can be inferred from the fact that the name of the winning team is stored or if there was no winner. The final example query at the end of this document demonstrates how this could be done. This approach is more correct from a theoretical design standpoint, as we are not storing data that can be inferred from other, more fundamental data, but this may not be the best approach in practice. Determining the number of points that a team has accrued is likely something that will be done frequently, and therefore it probably makes more sense to keep track of the score of the team explicitly, which would take slightly more disk space and have a database that is not fully normalised, but would save a lot of computing power and time when trying to determine the score of a team.

There is one practical, albeit niche, benefit to not storing the points that a team has accrued and only storing the winner of the game: at any point, we can change how we value a win, a loss, or a draw. Since we compute this only at query time, we can decide at any time in the future that a win should be worth 5 points, and a loss -2 points if we so desired, allowing for greater flexibility. Furthermore, if the ruling on the winner of the game changes at some point in the future due to, for example, evidence of cheating, we only need to change one value: `results.winner` rather than having to subtract the appropriate number of points from the scores of the teams.

1.6 The `active_players` Table

The `active_players` table records which players were "active" (i.e., playing) at a given moment in time. It consists of three columns:

1. The `active_time` column, which stores a `TIMESTAMP` of when the player was recorded as active. We assume that this is recorded frequently at regular intervals throughout the game, but the design also works if it is only updated when there is a change in which players are active. We chose to represent the time not in terms of the minutes of the game, but as a regular timestamp for simplicity's sake, although this does have the benefit of allowing us to implicitly determine the times during which a specific game was played.

2. The `game_id` column which is a foreign key to the `game_id` column in the `games` table, indicating the game in which this player was active. We make the (quite reasonable) assumption that a player can't be in two games at the same time.

3. The `player_id` of the active player, which is a foreign key to the `player_id` column in the `players` table. This, together with the `active_time` column, constitute a composite primary key `active_time, player_id`. We don't need to include the `game_id` in this primary key as it's assumed that a player can't be playing in two separate games at the same time.

There is potentially a lot of redundant data in this table if data is inserted every time it is recorded, at frequent intervals, as the specific players who are active on the pitch don't change that often; in fact, we know that there can be a maximum of five substitutions per team, and therefore they can only change a maximum of ten times per game. A more efficient approach might be to insert new data only when there's a change, such as a substitution or a player is sent off. This however would increase the time taken to insert data, as we would first have to check that the data we're inserting is different from the last data in the database. We also maintain this data indefinitely so that we can look back and see what players were playing at any given recorded instant in time. If this is not desired, we could replace this approach with a database whose content is truncated whenever new data is known. Then, we could do away with the `active_time` column and store only the players active at this moment, and the game, although this would make it difficult to ensure that the data is up-to-date and valid.

1.7 The `substitutions` Table

The `substitutions` table consists of four columns:

1. The `substitution_time` column stores a `TIMESTAMP` of when the substitution occurred.

2. The `game_id` column stores the game in which the substitution occurred, and is a foreign key to the `game_id` column in the `games` table.

3. The `off_player` column stores the player ID of the player being replaced, and is a foreign key which references `players.player_id`. This constitutes one half of the primary key along with the `substitution_time` column, as we assume that a player can only be substituted with one other player at one time. We include the `off_player` in this composite key as we assume that several substitutions could occur in the same instant, so there must be some way to distinguish them.

4. The `on_player` column stores the ID of the player that is replacing the `off_player` and is also a foreign key that references the `player_id` column in the `players` table.

1.8 The `sendoffs` Table

The `sendoffs` table keeps track of which players have been sent off and when. It consists of three columns:

1. The `sendoff_time` of the sending-off, as a `TIMESTAMP`.

2. The `game_id` of the game in which the sending-off occurred, which is a foreign key to `games.game_id`.

3. The `player_id` of the player who is being sent off, which is a foreign key to `players.player_id`. The primary key is a composite of `(sendoff_time, player_id)` as we assume that the same player cannot be sent off from two different games at once, and that multiple send-offs could occur at the same instant.

If a send-off was always followed by a player being substituted on in their place, we could do away with the `sendoffs` table and use only the `substitutions` table, with a Boolean column to indicate whether the substitution was as a result of a send-off or whether it was voluntary. However, the requirements specify that there is a maximum of 5 substitutions per team per game, so it is possible that if a team had used up all their substitutions, they would be unable to substitute on another player if one was sent off. Furthermore, it's possible that there would be no one available to substitute for the player who was sent off. For these reasons, we created a separate `sendoffs` table, to allow us to have send-offs without a substitution.

1.9 The `goals` Table

The `goals` table consists of four columns:

1. The `goal_time` that the goal was scored, as a `TIMESTAMP`.

2. The `game_id` of the game in which the goal was scored, as a foreign key to `games.game_id`. The primary key is a composite of `(goal_time, game_id)`, as we assume that there can only be one goal scored at one instant in a game (the ball can only be in one place at once).

3. The `player_id` of the player who scored the goal, as a foreign key to `players.player_id`. We store only one `player_id` as we assume that each goal can have only one scorer and that we do not count assists.

4. The `benefitting_team`, or the team to which the points are awarded for this goal. The sole purpose of this column is to distinguish own-goals from other goals; if we didn't count own-goals, then we could deduce the `benefitting_team` by looking at which team the scoring player belonged to. This column allows us to attribute responsibility for an own-goal to a player without awarding that player's team any points. An alternative to using this column could be to attribute any goal to the last player who touched the ball that was not on the team that conceded the goal. This would avoid confusion regarding own-goals, but would not allow us to distinguish whether or not a team gained points from an own-goal, and could make it seem like a player was a better scorer than they actually were.

1.10 SQL Code to Create Tables

```
CREATE TABLE teams (
    team_name VARCHAR(255) NOT NULL,         -- assuming team name is unique
    home_venue VARCHAR(255) NOT NULL,        -- assuming that two teams could share a home venue
    ↪  (e.g, A teams and B teams)
    manager VARCHAR(255),                    -- assuming no other information is known/required for
    ↪  managers other than name

    PRIMARY KEY (team_name)
);

CREATE TABLE players (
    player_id INT NOT NULL AUTO_INCREMENT,
    team_name VARCHAR(255) NOT NULL,
    squad_number INT NOT NULL,               -- assuming squad number is unique within squads
    player_name VARCHAR(255) NOT NULL,

    PRIMARY KEY (player_id),
    FOREIGN KEY (team_name) REFERENCES teams(team_name)
);

CREATE TABLE games (
    game_id INT NOT NULL AUTO_INCREMENT,     -- no other uniquely identifying information about a
    ↪  game (there could be two otherwise identical games)
    home_team VARCHAR(255) NOT NULL,         -- venue can be inferred from this
    away_team VARCHAR(255) NOT NULL,
    path_to_heap_file VARCHAR(255),

    PRIMARY KEY (game_id),
    FOREIGN KEY (home_team) REFERENCES teams(team_name),
    FOREIGN KEY (away_team) REFERENCES teams(team_name)
);

-- this table should only be inserted into upon the completion of a game
CREATE TABLE results (
    game_id INT NOT NULL,
    winner VARCHAR(255),                     -- NULL value indicates a draw

    PRIMARY KEY (game_id),
    FOREIGN KEY (winner) REFERENCES teams(team_name)
);

-- data redundancy here: players rarely change during match
CREATE TABLE active_players (
    active_time TIMESTAMP NOT NULL,
    game_id INT NOT NULL,
    player_id INT NOT NULL,

    PRIMARY KEY (active_time, player_id),
    FOREIGN KEY (game_id) REFERENCES games(game_id),
    FOREIGN KEY (player_id) REFERENCES players(player_id)
);
```

```sql

CREATE TABLE substitutions (
    substitution_time TIMESTAMP NOT NULL,
    game_id INT NOT NULL,
    off_player_id INT NOT NULL,
    on_player_id INT NOT NULL,

    PRIMARY KEY (substitution_time, off_player_id), -- assuming two substitutions could be done at the same time
    FOREIGN KEY (game_id) REFERENCES games(game_id),
    FOREIGN KEY (off_player_id) REFERENCES players(player_id),
    FOREIGN KEY (on_player_id) REFERENCES players(player_id)
);

CREATE TABLE sendoffs (
    sendoff_time TIMESTAMP NOT NULL,
    game_id INT NOT NULL,
    player_id INT NOT NULL,

    PRIMARY KEY (sendoff_time, player_id),  -- assuming two sendoffs could be done at the same time - otherwise could just use time
    FOREIGN KEY (game_id) REFERENCES games(game_id),
    FOREIGN KEY (player_id) REFERENCES players(player_id)
);

CREATE TABLE goals (
    goal_time TIMESTAMP NOT NULL,           -- assuming two goals can't be scored at the same time
    game_id INT NOT NULL,
    player_id INT NOT NULL,                 -- cam infer squad number from this
    benefitting_team VARCHAR(255) NOT NULL, -- the team to which the points are awarded for this goal

    PRIMARY KEY (goal_time, game_id),
    FOREIGN KEY (game_id) REFERENCES games(game_id),
    FOREIGN KEY (player_id) REFERENCES players(player_id),
    FOREIGN KEY (benefitting_team) REFERENCES teams(team_name)
);
```

Listing 1: SQL Code to Create the Tables

2 Query Design

2.1 List all players playing for a given team

The below SQL query takes the simplest interpretation of "a player playing for a given team", i.e. a player who belongs to a certain team, regardless of whether or not they are actively playing for that team at this moment in time. It also assumes that the resulting information desired from the query is the player's name; `player_name` could be replaced with * to get all the information pertaining to the player.

```sql
SELECT player_name
FROM players
WHERE team_name = "<insert given team name here>";
```

Listing 2: Simple Query to List All Players Associated with a Given Team

If we take the more complicated interpretation of "playing for a given team" to mean "a player who at this specific moment in time is on the pitch playing a game for this team", the following query would work:

```
SELECT
    player_name
FROM
    active_players JOIN players ON active_players.player_id = players.player_id
WHERE
    team_name = "<insert team name here>" AND
    active_time = "<insert desired TIMESTAMP here>";
```

Listing 3: Query to List All Players Playing for a Given Team at a Given Moment in Time

Note that for this query, we have to supply the timestamp of the instant at which we want to list all the players who are playing for the team. To do this with the most recent timestamp available in the database, we could use a sub-query:

```
SELECT
    player_name
FROM
    active_players JOIN players ON active_players.player_id = players.player_id
WHERE
    team_name = "<insert team name here>" AND
    active_time = (
        SELECT MAX(active_time) FROM active_players
    );
```

Listing 4: Query to List All Players Playing for a Given Team at the Last Recorded Instant

2.2 List all players who have scored in a given game

```
SELECT DISTINCT player_id -- DISTINCT so we only list each scorer once
FROM goals
WHERE game_id = "<insert given game ID here>";
```

Listing 5: Query to List the Player IDs of All Players Who Have Scored in a Given Game

If we wanted more relevant information that just a numeric, auto-generated player ID, we could use a query something like the following:

```
SELECT DISTINCT
    player_name, team_name
FROM
    goals JOIN players ON goals.player_id = players.player_id
WHERE game_id = "<insert given game id here>";
```

Listing 6: Query to List the Name & Team of All Players Who Have Scored in a Given Game

Of course, we may not want to count players who have only scored own goals throughout the game. The following query does the same as the above query, but excludes those who have only scored own-goals:

```
SELECT DISTINCT
    player_name, team_name
FROM
    goals JOIN players ON goals.player_id = players.player_id
```

```
5 WHERE
6     benefitting_team = team_name AND
7     game_id = "<insert given game id here>";
```

Listing 7: Query to List the Name & Team of All Players Who Have Scored Against the Other Team in a Given Game

2.3 List the top five goal scorers in the league

```
1 SELECT
2     player_id, COUNT(*) as goals_scored
3 FROM goals
4 GROUP BY player_id
5 ORDER BY goals_scored DESC
6 LIMIT 5;
```

Listing 8: Query to List the Player IDs of the Top 5 Goal Scorers in the Leagues

Again, if we wanted more relevant information that just the player ID, we could use something like the following query:

```
1 SELECT
2     player_name, team_name, COUNT(*) as goals_scored
3 FROM
4     goals INNER JOIN players ON goals.player_id = players.player_id
5 GROUP BY goals.player_id
6 ORDER BY goals_scored DESC
7 LIMIT 5;
```

Listing 9: Query to List the Names & Team of the Top 5 Goal Scorers in the Leagues

Again, we of course might want to exclude own-goals from our calculations:

```
1 SELECT
2     player_name, team_name, COUNT(*) as goals_scored
3 FROM
4     goals INNER JOIN players ON goals.player_id = players.player_id
5 WHERE benefitting_team = team_name
6 GROUP BY goals.player_id
7 ORDER BY goals_scored DESC
8 LIMIT 5;
```

Listing 10: Query to List the Names & Team of the Top 5 Goal Scorers in the Leagues, Excluding Own-Goals

2.4 List all teams and the amount of points they have so far

The following query lists the name of each team and counts up the number of points that they have earned so far, 3 for a win, 1 for a draw, and 0 for a loss.

```
1 SELECT
2     teams.team_name,
3     SUM(
4         CASE
5             WHEN results.winner = teams.team_name THEN 3    -- assigning 3 points for a win
6             WHEN results.winner IS NULL THEN 1              -- assigning 1 point for a draw
7             ELSE 0                                          -- no points for a loss
8         END
```

```
9       ) AS total_points
10  FROM
11      -- creating a temporary table containing each team and every game they have played, either at
        ↪  home or away, and joining that to the results table
12      teams LEFT JOIN games ON teams.team_name = games.home_team OR teams.team_name =
        ↪  games.away_team LEFT JOIN results ON games.game_id = results.game_id
13  GROUP BY teams.team_name;
```

Listing 11: Query to List All Teams & their Amount of Points So Far

This is quite a complicated and costly way of calculating the score of a team in the league. While this has its advantages (as explored in the above `results` table section), it likely is too inefficient for practical uses. A more practical, but perhaps "less technically correct" approach would be to store the number of points awarded to each team for each game, and simply sum these up each time, or even just store the total number of points the team has earned so far in its own table, and increment the number by the appropriate amount for each match that they complete. Both of these approaches would greatly improve the efficiency of querying this information.