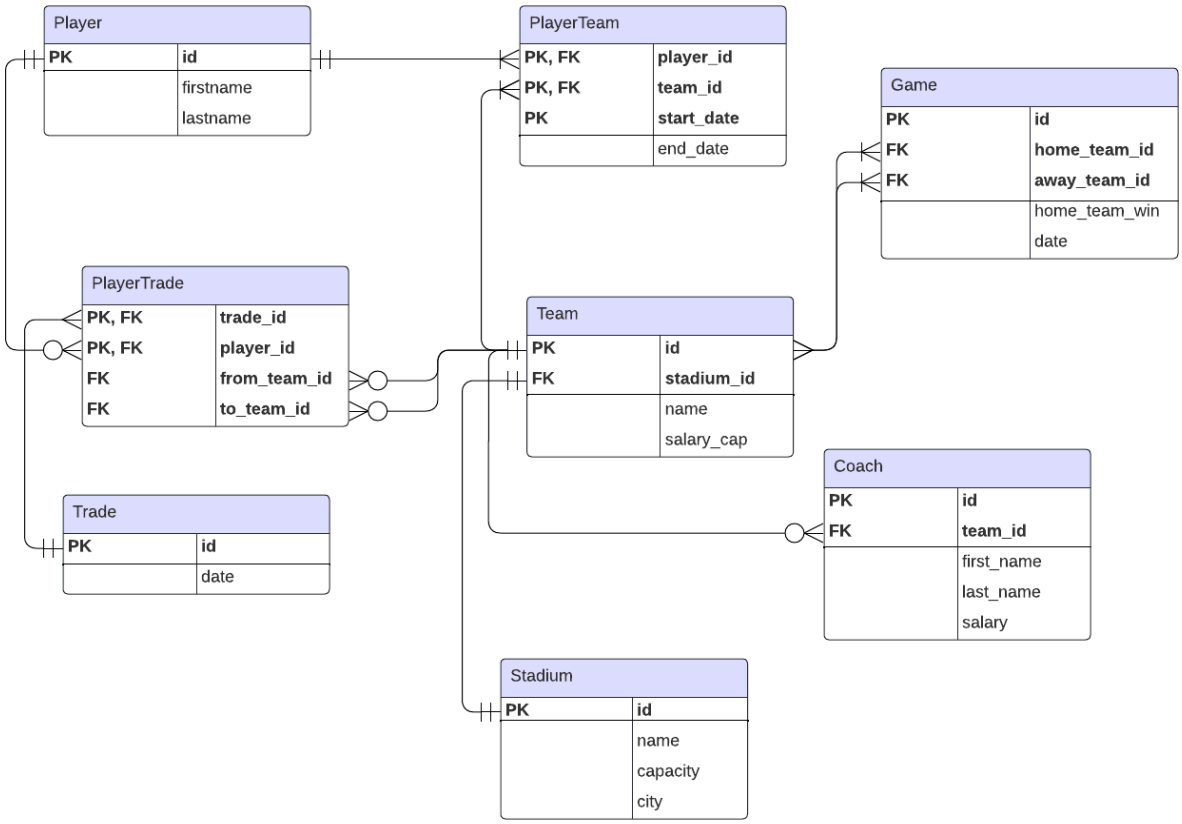
Names: Ethan Fang (ewf22)
William Schneider (was103)
Jackson Schuetzle (jss270)

Name of Database: Football League

Crow's Foot Diagram:



Tables

Our ERD fully encompasses all the use cases for the football league. A couple notes about the relationship between tables in the database is given below.

Player & Team: There is a many-to-many relationship because (1) many players are on a team, and (2) a player can play for many teams during their career. This creates a need for the *PlayerTeam* table, which serves as a bridging table. The *start_date* and *end_date* attributes are a necessary part of the primary key for *PlayerTeam* because there are situations where a player gets traded many times in their career and they end up playing for the same team twice. While a player is active on a team, the *end_date* field will be NULL. Once the player retires or is traded, the *end_date* field will be updated.

Trade, Player, and Team: The trading system in our league limits a trade to only two teams, however any number of players are allowed to be included in the trade from each team. This means the relationship between team and trade is one-to-many because a team can have many trades, but a trade can only have a team listed once (in reality there's only two teams per trade, so a two-to-many relationship).

Since the trades allow teams to give up any number of players, the relationship between players and trades is many-to-many because a trade can contain many players, and a player may be traded many times in their career. A *from_team_id* and *to_team_id* is necessary because it may be confusing to determine exactly where a player was traded to/from if their current team was unrelated to that trade.

Team and Game: A team must play at least one game to be considered a part of the league. Therefore, the relationship between team and game is 1:M because a team plays multiple games, and a game can have a team listed once (same situation as Team vs. Trade relationship…it's really a two-to-many relationship). Originally, we had an attribute *winning_team_id*, however there is no need for this extra foreign key. We replaced that attribute with the boolean value *home_team_win* which signifies whether the team *home_team_id* won the game.

Coach and Team: In our league, it is not required for a team to have a coaching staff, but there is also no limit to the number of coaches a team can have. Therefore, the team to coach relationship is one-to-many. The caveat is that each team has an attribute *salary_cap* which signifies the salary limit for the entire coaching staff, i.e. the salaries of all the coaches can't be higher than *salary_cap*. If a coach is fired, the coach's entry in the *Coach* table is deleted.

Team and Stadium: Each team can only have one associated home stadium with them. A team is not allowed to be in the league without a home stadium, therefore the team to stadium relationship is one-to-one, hence the *stadium_id* attribute in the *Team* table. If renovations are made to the stadium, the *capacity* attribute can be updated.

Functional Dependencies

The functional dependencies for our database design are quite simple. **Except for relations that serve to bridge two other relations** with a many-to-many relationship, all relations in the database design use **an attribute *tablename_id* as a candidate key**, i.e.

$$(id_R \rightarrow R) \text{ for } R \in \{Player,\ Team,\ Game,\ Trade,\ Coach,\ Stadium\}$$

For the bridging tables, *PlayerTeam* and *PlayerTrade*, the single-attribute *tablename_id* is only a part of a composite candidate key. This is because an instance of a player on a team is

dependent on the time they play for. The trading system allows for a player to get traded many times, and it could easily be the case where a player gets traded to a team they already played for. Hence, this is what the candidate key of the table looks like

$$\{id_{player}, id_{team}, startdate\} \rightarrow PlayerTeam$$

We originally had the *end_date* attribute as a part of the primary key, but realized that this wouldn't be a true candidate key. This is because the even though a player can play for the same team twice–a case where the pair (id_{player}, id_{team})–a player can never play for the same team twice with the same starting date. Therefore, the attribute *end_date* can be determined by only these three attributes and can't be a part of the primary key.

Finally, for the *PlayerTrade* relation, only the *trade_id* and *player_id* attributes are the primary key. The *trade_id* by itself is not a super key because $id_{trade} \nrightarrow id_{player}$ because a single trade contains multiple players. Therefore, adding the *player_id* to the primary key makes it a valid candidate key because a player can't show up more than once in the same trade, making the pair $(id_{trade}, id_{player})$ always unique. As a result,

$$\{id_{trade}, id_{player}\} \rightarrow PlayerTrade$$

Handling Normalization Issues

Our database follows all of the usual database normalization practices and is therefore in third normal form.

First normal form requires that our database doesn't have repeating attributes in individual tables. An example of this practice in our database would be our *PlayerTrade* table. Rather than storing the specific player IDs in our *Trade* table, we decided to populate the *PlayerTrade* table for every player that takes place in a trade. When a trade occurs between teams, we can trade 2 players for 3 players, 1 for 5, 3 for 7, etc. This way, we allow trades with various possibilities of player combinations to allow for a dynamic database. This also allows for a simple system for our user to query for every time a player has been traded (through the *player_id* attribute).

Second normal form requires that records shouldn't depend on anything other than a table's primary key (as a whole, not individual parts of the primary key if it is composite). All of our tables contain only the necessary attributes and have foreign keys to other tables to reference data that is repeated. Looking through our tables, our *Player, Team, Trade, Stadium,* and *Coach* tables only have *id* as their primary key, with additional attributes to tell basic information about each table (name, salary, etc). Our *PlayerTrade* and *PlayerTeam* tables are a bit more complicated as bridging tables, but have a simple layout to follow 2NF. *PlayerTeam* tracks every instance of a player during their time on a specific team. Since a player could technically be traded back to one of their old teams, all of the attributes in *PlayerTeam* are necessary for the

primary key. Our *PlayerTrade* table depends on *trade_id* and *player_id*, which will never repeat since a player can only appear once a trade. Our *from_team_id* and *to_team_id* fully depend on the composite primary keys since a player must go from one team to another, and we are only tracking one specific team.

Third normal form requires that we eliminate transitive dependencies, which is when a non-key attribute is functionally dependent on another non-key attribute rather than directly on the primary key. Following 3NF means that we need to break down the data into smaller, more logically organized tables. In our database, all of our non-key attributes are fully dependent on their relation's primary key. As described in the previous paragraph on 2NF, all of our non-key attributes are basic information about entries in the database such as first_name, last_name, salary, and date. These attributes are specific information pertaining to the primary key (usually the **id**), so therefore our database follows 3NF since all attributes that could be used in multiple tables are stored in a separate table so it can be accessed/referenced separately. This avoids redundancy and creates ease of access, e.g. *Stadium* as a separate table, since it could be referenced in the future if sponsors were added.

Indexes

In the database frontend, users won't be searching by *player_id* or *team_id* because those fields should be abstracted. Rather, the user will be searching by *player_name* and *team_name*. Therefore, creating indexes for these columns is important and may look like so

```
CREATE INDEX playerFirstName ON player(firstname);
CREATE INDEX playerLastName ON player(lastname);
CREATE INDEX teamName ON team(name);
```

It also is the case where the ids of teams and players are referenced frequently by join commands. Since we want lookups to be optimized, indexes with these bridging table attributes should be created like so

```
CREATE INDEX playerTeamPlayerID ON playerTeam(player_id);
CREATE INDEX playerTeamTeamID ON playerTeam(team_id);
CREATE INDEX playerTradePlayerID ON playerTrade(player_id);
CREATE INDEX playerTradeTeamID ON playerTrade(from_team_id);
```

Finally, the history of a team's win-loss record may be examined quite frequently for statistical analysis. Therefore, the two team ids in the game table can be put inside an index.

```
CREATE INDEX homeTeamID ON game(home_team_id);
CREATE INDEX awayTeamID ON game(away_team_id);
```

Triggers

- When a trade occurs, insertions into the *PlayerTrade* table occur. However, there are certain procedures that should automatically be happening in the *PlayerTeam* table as a result
    1. There should be an UPDATE to the *end_date* attribute of the record of the *player_id* in the trade (associated with the *from_team_id*)
    2. There should be an insertion with the *player_id, to_team_id,* and *start_date*. The end_date should be instantiated as null, as the player is currently on the team.

- When you add a coach to the team, there should be a trigger to check that a coach doesn't already exist for that team, since we have a 1-1 relationship between coach and team tables.

- Every team has a *salary_cap* for the coaching staff. It would be more realistic to incorporate the players into this salary cap, but we wanted to keep our database implementation simple before we increased the complexity. Therefore, when a coach is hired, i.e. an insertion into the *coach* table occurs, the database must execute a trigger to verify that the hiring won't exceed the current salary_cap of the team. Similarly, when the *salary_cap* attribute is updated, a trigger must verify that the current coaching staff doesn't exceed the new salary cap, or else the update will be canceled.

Desired application of database / Outline of Use Cases

Our football league database application solves the problem of managing and tracking a football league. Our database will be able to store information on players, coaches, teams, games, and trades of players between teams. Having a digital method of entering, storing, and updating this information is much more efficient and dynamic than the alternative (pen and paper, spreadsheet, etc.). Use cases for our database could include simple data analytics regarding player/coach history, team winning percentages, and trade history.

Data scientists can analyze the historical data stored in our database to help drive predictive models for team performance during season based on player and coach composition.

Here a few use cases that could drive analytical insights:

- Find team's win percentage during a season
- Find player's win percentage during time on a certain team, or overall
- Find team win percentage when a certain coach is on the coaching staff
- Analyze player performance during their time on a certain team

Work of each member

Will - Frontend
Ethan - Java to Frontend
Jackson - Java to Database