Basketball League (NBA)

CS 4354 Database Project

Group 4

Members:

Reagan Schulte

Seth Michaels

Hardik Poudel

Yathartha Regmi

Bibek Pokharel

Date: May 7th, 2021

Table of Contents

1) Problem Definition (Bibek Pokharel)

2) Approach (Reagan Schulte)

3) Results (Hardik Poudel)

4) Conclusion (Yathartha Regmi)

1) Motivation and Problem Definition

Motivation:

The NBA is a huge league with 82 games per season. 30 teams are competing for the title and each team has 17 players in an active roster. So, managing all this data about the teams, players, and seasons is complicated. Sometimes it is hard to find all the data on the same page. Therefore, we decided to create a database with all the information about NBA teams and players. This database will consist of tables for Team, player, player contract, player stat, team stat with all the information.

Problem definition:

While designing this database, first we had to discuss the trustworthy website from where we can extract the required data about the player and teams. For this database first, we decided what information we need to include and what we will not include. We decided to include the information of players and teams from the NBA. We also discussed what will be our primary key and what will be our foreign key. We also decided to create five tables for the database which are as follows:

i. Team
ii. Player
iii. Player stat
iv. Team record
v. Player's contract

Creating these tables will help to find all the data and stats of NBA teams and players in the same place.

2) Approach

When thinking of what encompasses a basketball league, teams and players come to mind. There is no league without multiple teams and players who play on those teams. These

entities are the main part of a league, so they are the main tables of the database. Most foreign keys will derive from the team's and player's tables.

As teams play each other, stats are accumulated for the players and teams. These are two other conceptual entities that are part of the database. When players play in a game, they get points, rebounds, and assists to help their team win but sometimes their team comes up short and loses.

In a professional basketball league, like the NBA, players have contracts to play for their teams. These contracts establish how long a player will play for their specific team and how much money. A player can only have one contract at a time for only one specific team.

For the team's table, a unique identifier is needed to identify each team. For this, an auto-increment field, called teamID, is used as a primary key that will be used as another table's foreign key to track back to the team's table. This will allow easy access to the team's table that holds the location of the team and their nickname as both varchars of 25 lengths. A team consists of at least 5 players and a max of 17. A team also has a record of their wins and losses. Of course, as seasons go by a team has a record for each season.

Relationships from the Teams table

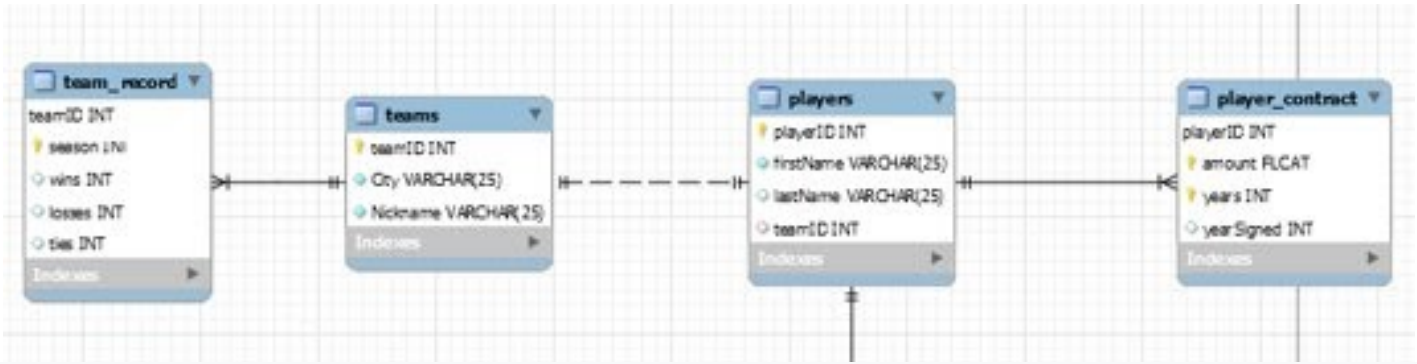
The player's table is similar to the team's table, for which there is an auto-increment field that is the primary key to identify each player in the league, called playerID. Each player consists of a first name and the last name, although the last name can be null for players with only one name. These names are varchars with a length of 25. And since each player plays for only one team, teamID is used as a foreign key in this table, connected back to the team's table. Some players might not be playing for a team currently if a team cuts them. These players are called "free agents" so this field can be null in this table. Players accumulate stats, and like Team record, each season a player accumulates stats. Players can also have multiple contracts throughout their careers.

Relationships from the Players table

There are two tables for stats. A player records individual stats, this table is called player_stats. In this table, points, rebounds, and assists are kept for each player. These stats are per game stats, so they are floats. To uniquely identify each of these player's stats playerID and another attribute called season are the primary key. Players accumulate their stats over a season when their teams play games to see who's the best. Then another season comes and they start over. The season attribute, which is an int, will distinguish which season the player accumulated which stats. And the playerID attribute is used as a foreign key back to the player's table.

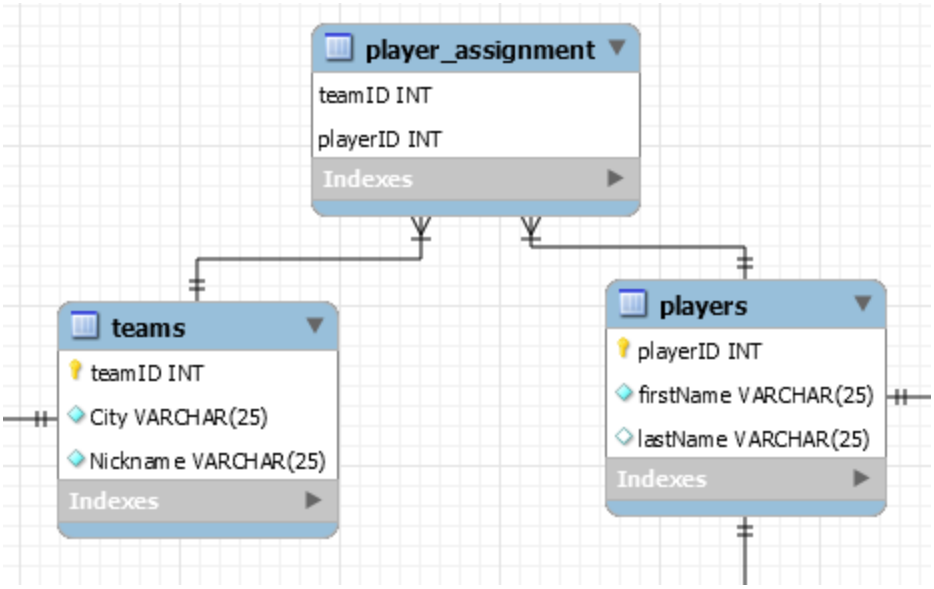
The other table for stats is the team_stats record. This table holds all the wins, losses, and ties attributes which are all ints. Like the player_stats table, there is a season attribute to establish which season a team had their particular stats. This along with the playerID make up the primary key. The playerID attribute is also a foreign key connecting back to the player's table.

Finally, the player_contracts table holds all the contracts players have currently and have had in the past. To replicate these contracts in the database, there is a yearSigned attribute that is an int and shows the year the player signed the contract. There is a year's attribute that shows how long the contract is for and is also an int. And there is an amount attribute that shows the total amount that will be paid to the player over the years the contract was signed for. Of course, there is a playerID attribute that is a foreign key connecting back to the player's table. PlayerID with amount and years makes up the primary key.

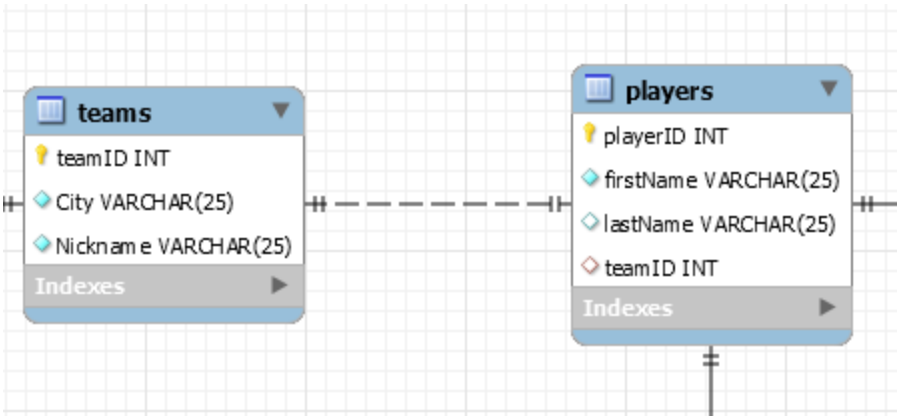


Database Schema

Initially, to more normalize the database there was another table called player_assignment created. This table would have held playerID and teamID, signaling what player is assigned to what team. But, a denormalized form of this, putting teamID into the player's table, was better so that every time a query is run with a player only one join would have to be used to find the team name a player plays for instead of two joins. A lot of queries like this would be run on the database as well since anytime we want to get information about the player we will always need the team they play for.



Normalization of players and teams



Final denormalization

3) Results

After the data was added to the database, several queries were added to make the database functional.

SELECT was used to view the contents of the table. For eg:

```
4  SELECT * from teams;
5  SELECT * from team_record;
6  SELECT * from players;
7  SELECT * from player_stats;
8  SELECT * from player_contract;
9  SELECT * from players_account;
```

Here we can replace * with the column names to view only the required columns from the desired table.

Once the data was added to the table player_contract, a new field was needed to see how much tax each player paid. To make things easier we used a trigger function.

A trigger was added so that whenever a new contract of the player was added, the tax paid by the player was automatically added to the table player_account.

```
218  -- Triggering
219  -- This trigger function playerInfo will add tax paid by the players and random age (for demo) whenever
220  -- data is inserted in the table player_contract
221
222  drop trigger playerInfo;
223  create trigger playerInfo
224  AFTER INSERT ON player_contract
225  FOR EACH ROW
226  INSERT INTO players_account(tax, age)
227         VALUES (new.amount * 37 /100 , FLOOR(RAND() * (45-18 + 1) + 18));
```

Instead of writing long queries, all the time views were created that resemble a new table for each query. For eg., a view players_city was created to view players from Dallas city.

```
10
11      -- Create view players_city to view players from Dallas city.
12 •    DROP VIEW players_dallas;
13 •    CREATE VIEW players_dallas as SELECT * from players NATURAL JOIN teams where city = "Dallas";
14 •    SELECT firstName, lastName from players_dallas;
15 •    SELECT * from players_dallas NATURAL JOIN player_stats;
16
```

This view can be used as a new table and can be combined with other queries.

Joins were added to combine rows and columns from different tables such as INNER JOIN, OUTER JOIN, LEFT JOIN, RIGHT JOIN, and NATURAL JOIN. But we found NATURAL JOIN as the most favorable JOIN because it combined the table without showing the duplicate column.

```
16   -- To view players for all cities
17   select * from players INNER JOIN teams ON players.teamID = teams.teamID;
18
```

```
25      -- To view players and their contracts;
26 •    select * from players NATURAL JOIN player_contract;
27
```

Some built-in methods like >,<, <=, >= were used to filter out the table for eg: to view the player names with contracts for more than 2 years would filter out the rows from the column years with a value greater than 2.

```
31  -- To view players names who has player contract more than 2 years
32  select firstName, lastName ,years from players NATURAL JOIN player_contract where years > 2;
33
```

Likewise, methods like max and min are useful to find the maximum and minimum values from the column. For eg: the below queries to filter out the player with the maximum amount from the table player contract and display the details of the players.

```
34  -- To view players details with its team who earns the maximum amount
35  select * from players NATURAL JOIN player_contract NATURAL JOIN teams where amount = (select max(amount) from player_contract);
36
37  -- To view player details with its team who ears the minimum amount
38  select * from players NATURAL JOIN player_contract NATURAL JOIN teams where amount = (select min(amount) from player_contract);
```

Queries such as sum are pretty useful to calculate the total from the column.

```
58  -- To get the total tax paid by all players
59  select sum(tax) from players_account;
60
```

The ORDER BY was used to view the table of players' names and their accumulated points in ascending order.

```
43  -- To view player first name in order of the points
44  select firstName, Points from players NATURAL JOIN player_stats order by Points ASC;
```

LIKE queries are useful to find the row with similar strings using regex like below. The following query displays the players that play from a city similar to “Boston”.

```
19  -- To view players with city like boston
20  select * from players NATURAL JOIN teams where city LIKE "%boston%";
```

These queries have been used throughout the source code in various ways to make the database functional.

4) Conclusion

We know, in today's world, data is really important to make any decision. In the field of sports science, databases that have tools to capture, store, manage, and retrieve

info have the potential to be one of the most powerful tools in sports science. When we were designing our database we kept this in mind and implemented functions such as views, where, join, order by, sum, and many others. In our case, for the NBA, in the future, we could add attributes to store a player's stats for each game, or a season, and recruiters or scouts can use this data to find an ideal player for their team.

When we were designing our database we had two things in mind, data integrity and Accessibility. We had to make sure that the data we used was accurate, so we had to use a website to get all the players and team stats for a season. For accessibility, we made sure our data was easily available to users and the data we provide is helpful. By giving them the team stats and players stats weren't enough, we also have to take into consideration other things that can show a player's full potential. For now, we have attributes that show a player's stats in a game, their total rebounds, their assists, and so on. In future updates we talked about adding other characteristics that scouts look for like versatility - their defensive, attacking, and passing skills; Athleticism- their vertical stat, lateral quickness, and dribbling speed.

As we were looking at the primary keys to do lookups, and each key was unique we didn't have to check for redundant elements and this eliminated the need for indexing.