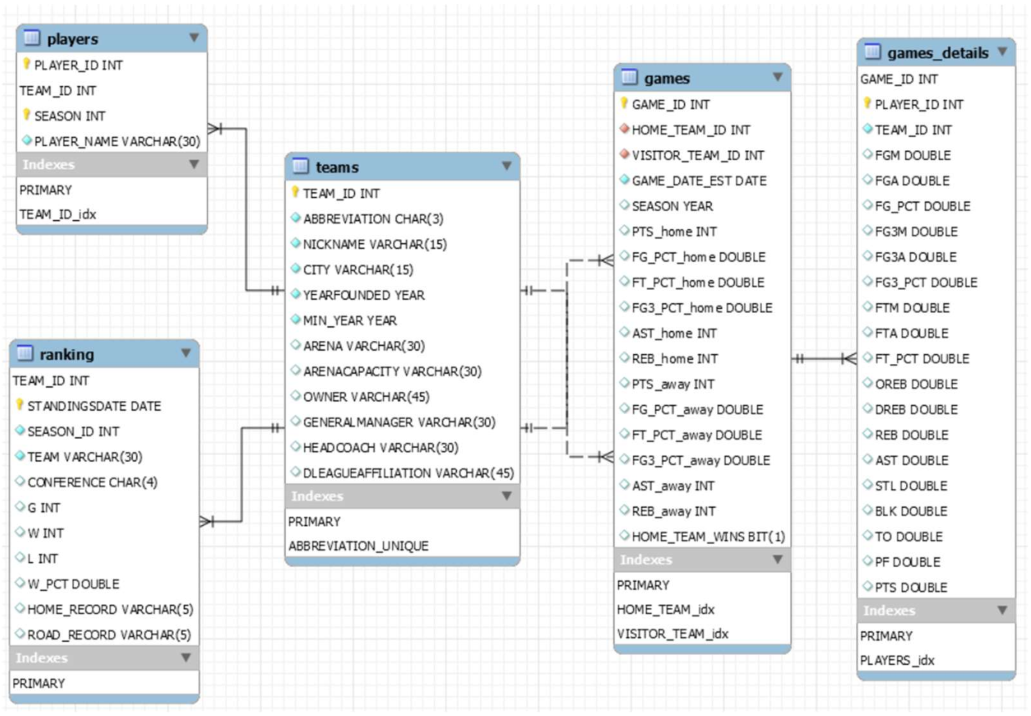
NoSQL (MongoDB)

The database I used is the same as the first two homeworks but with a few differences. Here you can see the initial SQL database schema. There are 5 tables related to players, matches and respective statistics:



Compared to this, in the new database I introduced the PLAYER_NAME attribute for the games_details table which was replicated from the players table. This operation is useful to simplify the structure of many queries. Secondly, indexes and keys were not considered.

To implement this homework I used the MONGODB DBMS and, in particular, I used the visual tool NoSQLBooster to import and manage the database. As a result, we move from a relational database that assumes a table structure, to a document-oriented database with a json file structure. So, tables become collections and rows become documents.

The queries I will analyze were almost all taken and redesigned from the first homework in order to highlight the differences between SQL and a NoSQL language.

QUERIES

(To see the direct comparison between sql and nosql queries look at the powerpoint)

1-2) return players who have scored at least 60 points in a game

1)

```
db.games_details.find({PTS:{$gte:60}},
                    {PLAYER_NAME:1,PTS:1,_id:0},
).sort({PTS:-1})
```

2)

```
db.games_details.aggregate([
   {$match:{PTS:{$gte:60}}},
   {$lookup:{
       from: "games",
       localField:"GAME_ID",
       foreignField:"GAME_ID",
       as:"games"
   }},
   {$unwind:"$games"},
   {$project:{PLAYER_NAME:1,"SEASON":"$games.SEASON",PTS:1,_id:0}},
   {$sort:{PTS:-1}}
])
```

With the sql language the query has a simple syntax where there are two joins on the games_details table.

For MongoDB one of the two joins does not have to be translated because the PLAYER_NAME attribute is also present in the games_details collection.

We have two options. As the first option we can use the find function to return documents from the games_details collection. The first argument of the find function is the selection criteria. In this case with the gte operator we impose that the player must have scored at least 60 points. The second argument of find is the projection. Here we return the player's name and the number of points, but not the id. Finally with the sort function we sort the players by the number of points in descending order. The problem with this query is that we cannot get the SEASON attribute present in the games collection.

For this reason an alternative is to use an aggregation pipeline. The filtering of players is done with the match operator. The lookup operator relates the games_details collection to the games collection, and the unwind operator destructures the results. Finally with the project and sort operators the fields are selected and sorted.

In the results of the two queries you can see that in both cases a set of objects is returned instead of a set of documents because there are no identifiers.

3) for each team, return the number of days in which that team had a league winning percentage greater than 50%

```
db.ranking.aggregate([

   {$match:{W_PCT:{$gt:0.5}}},

   {$group:{_id:"$TEAM_ID",TOT:{$count:{}}}},

   {$lookup:{

      from:"teams",

      localField:"_id",

      foreignField:"TEAM_ID",

      as:"teams"

   }},

   {$unwind:"$teams"},

   {$project:{"NICKNAME":"$teams.NICKNAME",TOT:1,_id:0}},

   {$sort:{TOT:-1}}

])
```

The information about the number of days is found in the ranking collection.

The most important aspect of this query is the use of the group aggregation operator. This operator allows to aggregate the results by the TEAM_ID and it also allows to compute the number of days through the count operator.

4) return players who participated in at least one game in which the home team won by more than 30 points over the visiting team during the season 2021

```
db.games.aggregate([

   {$project:{GAME_ID:1,SEASON:1,"DIFF":{$subtract:["$PTS_home","$PTS_away"]}}},

   {$match:{$and:[{SEASON:2021},{DIFF:{$gt:30}}]}},

   {$lookup:{

      from:"games_details",

      localField:"GAME_ID",

      foreignField:"GAME_ID",

      as:"gd"

   }},

   {$unwind:"$gd"},

   {$project:{"PLAYER_NAME":"$gd.PLAYER_NAME"}},

   {$group:{ _id:"$PLAYER_NAME"}}

])
```

To answer this question for sql I had used a nested query in the from clause. Then I had placed the condition on the difference between the points scored by the home team and the away team in the where clause.

In MongoDB the order of the operators is different and more important. The difference of the points is computed at the beginning in the project operator with subtract and then the condition is placed with the match operator. Finally at the end of the query the group operator on the PLAYER_NAME is used to replace the select distinct clause of the sql language.

The result of this query is a set of documents with no values.

5) return the number of points and the average points per game of all players who have played more than 300 games

```
db.games_details.aggregate([

   {$group:{
_id:"$PLAYER_NAME",N_MATCHES:{$count:{}},SUM_PTS:{$sum:"$PTS"},AVERAGE_PT
S:{$avg:"$PTS"}}},

   {$match:{N_MATCHES:{$gt:300}}},

])
```

Here there are 3 aggregation operators: count, sum and avg. These are used to return the total number of games, the total number of points scored, and the average points scored per game by each player. So, due to the presence of the PLAYER_NAME attribute in the games_details collection, there is no need to perform any lookup in this query to reproduce the left join of sql.

6) return the names of the teams in which at least one player has scored more than 60 points in a game

```
db.games_details.aggregate([

   {$match:{PTS:{$gt:60}}},

   {$lookup:{

      from:"players",

      localField:"PLAYER_ID",

      foreignField:"PLAYER_ID",

      as:"players"

   }},

   {$lookup:{

      from:"teams",

      localField:"TEAM_ID",

      foreignField:"TEAM_ID",

      as:"teams"
```

```
    }},
    {$unwind:"$teams"},
    {$group:{ _id:["$teams.CITY","$teams.NICKNAME"]}}
])
```

In this query, the condition is set first with the match operator. The double nested query is then replaced by two lookups between the games_details collection and the other two collections. It is also interesting to note that the document identifiers are arrays created with the group operator.

7) return games won by the Lakers in 2019 as home team

```
db.games.aggregate([
    {$lookup:{
        from:"teams",
        localField:"HOME_TEAM_ID",
        foreignField:"TEAM_ID",
        as:"teams1"
    }},
    {$lookup:{
        from:"teams",
        localField:"VISITOR_TEAM_ID",
        foreignField:"TEAM_ID",
        as:"teams2"
    }},
    {$unwind:"$teams1"},
    {$unwind:"$teams2"},
{$project:{"HOME":"$teams1.NICKNAME","OPPONENT":"$teams2.NICKNAME",GAME_DATE_EST:1,PTS_home:1,PTS_away:1,HOME_TEAM_WINS:1,SEASON:1}},
    {$match:{HOME:"Lakers",HOME_TEAM_WINS:true,SEASON:2019}},

    {$project:{OPPONENT:1,GAME_DATE_EST:1,PTS_home:1,PTS_away:1,_id:0}}

])
```

In this mongodb query there are two lookups between the games collection and two copies of the teams collection. Next ,we find two project operators in order to easily use the match operator on the HOME and OPPONENT attributes obtained from teams1 and teams2.

8) return the player with the highest average points per game

```
db.games_details.aggregate([

    {$group:{ _id:"$PLAYER_NAME",AVERAGE_PTS:{$avg:"$PTS"}}},

    {$sort:{AVERAGE_PTS:-1}},

    {$limit:1}

])
```

In sql I had used a nested query in the having clause. In this nested query there was also the all operator.

In MongoDB, however, I used a different approach. First I calculated the average points per game of each player with group. After that, I sorted the results in descending order and selected only the first result. A limitation of this query is that if there were two players with the same average points per game it would have selected only one of them. But fortunately this is not the case.

The last question is a new question that was not in the first homework and asks:

9-10) return players and their total of 3-points scored, in games in which they scored more than 30 points

9)

```
db.games_details.aggregate([

    {$match:{PTS:{$gt:30}}},

    {$group:{ _id:"$PLAYER_NAME",SHOTS3:{$sum:"$FG3M"}}}
])
```

10)

```
db.games_details.mapReduce(
   function(){emit(this.PLAYER_NAME,this.FG3M); },
   function(key,values){return Array.sum(values); },
   {
      query:{PTS:{$gt:30}},
      out:"3SHOTS"
   }
)
```

The first method I used is the classical approach with an aggregation pipeline. I first filtered out the games in which players scored more than 30 points. Then I grouped each player's games and computed the total points scored by each player.

As a second method, I used a map-reduce approach on the games_details collection. The map-reduce paradigm for MongoDB supports two functions and one object. The map function returns a set of key-value pairs. In this case the keys are the PLAYER_NAMES and the values are the 3-shots scored. Then the reduce function groups the values with the same key. In this case the values are added to find the total number of 3-shots scored by each player. Finally, the object consists of the query that is initially executed and the output.

As you see from the results, the map-reduce paradigm returns a single object with 501 fields instead of a set of documents.