SQL (MySQL)

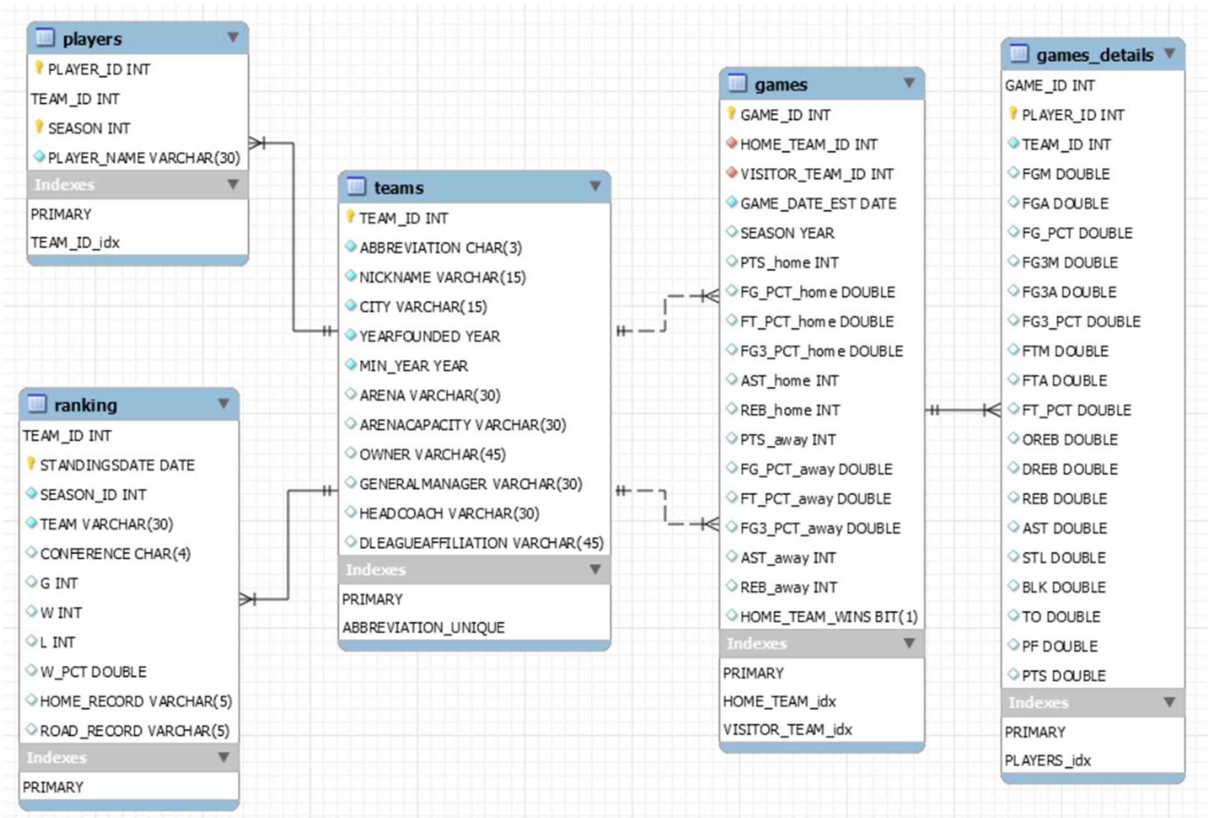
1) DATABASE SCHEMA

This work is related to a database on NBA games between the season 2004 and 2021.

To implement these two homeworks I used the MYSQL DBMS and, in particular, I used the visual tool MySQL WorkBench to manage the database.

The tables that are taken into consideration have been downloaded directly from the Kaggle platform in csv format and then imported into MySQL through the Import Wizard Tool that provides MySQL Workbench.

As you can see from this ER diagram, the database is composed of 5 tables that are linked to each other.



The first table is about players and the second one is about teams. The "games" table contains information about the games and their statistics, such as, points scored by the home and visitor teams. The "games_details" table contains the statistics of each game played by each player. Finally, the "ranking" table, set on a daily basis, contains the games played, won and lost by each team in a specific season.

The primary keys of each table are marked in yellow or without symbols in the diagram: the "players" table has a primary key composed by PLAYER_ID, TEAM_ID and SEASON, the "teams" table by TEAM_ID, then the "games" table by GAME_ID, the "games_details" table by GAME_ID and PLAYER_ID and finally the "ranking" table by TEAM_ID and STANDINGSDATE.

There are also foreign keys: the TEAM_ID which goes from "players" to "teams" and from "ranking" to "teams", the HOME_TEAM_ID and the VISITOR_TEAM_ID which go from "games" to "teams", and GAME_ID which goes from "games_details" to "games".

All of these relationships between tables belong to the “1 to N” type.

It is also important to point out that the tables "players" and "games_details" have not been linked together because, although, the games and the rankings refer to the period from 2004 to 2021, in the table "players" there are only those team members who have played at least one game between 2009 and 2018.

2) QUERIES

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

```
SELECT PLAYER_NAME, g.SEASON, PTS
FROM players p, games g, games_details gd
WHERE p.PLAYER_ID=gd.PLAYER_ID AND g.GAME_ID=gd.GAME_ID AND PTS>=60
GROUP BY gd.PLAYER_ID, gd.GAME_ID
ORDER BY PTS DESC
```

We select the PLAYER_NAME from the table "players", the SEASON from "games" and the points PTS from "games_details". We can see that in the WHERE clause there are two implicit joins and the condition that the player must have scored at least 60 points. The data is then grouped by game and player and finally placed in descending order by points scored.

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

```
SELECT NICKNAME, COUNT(*) AS TOT
FROM teams t JOIN ranking r ON t.TEAM_ID=r.TEAM_ID
WHERE W_PCT>0.5
GROUP BY r.TEAM_ID
ORDER BY TOT DESC
```

For this query we find the information in the tables "teams" and "ranking". We can see that there is an explicit join between these two tables on the TEAM_ID. In the WHERE clause it is set that the win ratio W_PCT must be greater than 0.5. And finally, we find the aggregation operator COUNT in the SELECT clause that with the GROUP BY allows to count the days.

3) *return players who participated in at least one game in which the home team won by more than 30 points over the visiting team between 2009 and 2018*

```
SELECT DISTINCT PLAYER_NAME
FROM players p, games g,
         (SELECT PLAYER_ID, gd.GAME_ID FROM games_details gd JOIN games g
ON
        gd.GAME_ID=g.GAME_ID WHERE SEASON>=2009 AND SEASON<=2018) AS gd1
WHERE p.PLAYER_ID=gd1.PLAYER_ID AND g.GAME_ID=gd1.GAME_ID AND
           PTS_home-PTS_away>30
```

In this query we select only the names of the players. We can use the operator DISTINCT on PLAYER_NAME instead of the GROUP_BY on PLAYER_ID because we know that there are no players with the same name and surname. There are also some joins and the condition on the difference between the points scored by the home team and the away team. The most important aspect of this query is the presence of a nested query in the FROM clause: this nested query is used to select only the games played by each player between 2009 and 2018.

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

```
SELECT PLAYER_NAME, gd.PLAYER_ID, COUNT(*) AS N_MATCHES, SUM(PTS) AS
            SUM_PTS, ROUND(AVG(PTS),2) AS AVERAGE_PTS
FROM games_details gd LEFT JOIN (SELECT * FROM players GROUP BY PLAYER_ID) p
         ON gd.PLAYER_ID=p.PLAYER_ID
GROUP BY gd.PLAYER_ID
HAVING N_MATCHES>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. A left join is used since some players present in the "games_details" table are not present in the "players" table. There is also an aggregation condition expressed with the HAVING clause. As you can see from the results of the query, there are some NULL values in the PLAYER_NAME given by the left join.

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

```
SELECT CITY, NICKNAME
FROM teams t
WHERE TEAM_ID IN (SELECT TEAM_ID
                  FROM players
                  WHERE PLAYER_ID IN (SELECT PLAYER_ID
                                      FROM games_details
                                      WHERE PTS>60))
ORDER BY CITY
```

We select the CITY and the NICKNAME of the teams. The selection condition is expressed through a double nested query in the WHERE clause where the IN operator is used twice. We select first the players who have scored more than 60 points and then the corresponding teams.

6) *return the names of Western Confederation teams that never scored more than 150 points in a home game*

```
SELECT CITY, NICKNAME
FROM teams t
WHERE EXISTS (SELECT *
      FROM ranking r
      WHERE t.TEAM_ID=r.TEAM_ID AND CONFERENCE='West')
      AND NOT EXISTS (SELECT *
          FROM games g
          WHERE TEAM_ID=HOME_TEAM_ID AND PTS_home>150)
```

This query is similar to the previous one, but here we use the EXISTS operator. Specifically, we use this operator to select the teams that belong to the western confederation. This information can be found in the "ranking" table. This query is also a correlated one because the TEAM_ID variable in the inner block is defined in the outer block. The NOT EXISTS operator is then used to not select teams that have scored more than 150 points in a game.

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

```
SELECT NICKNAME AS OPPONENT, GAME_DATE_EST, PTS_home, PTS_away
FROM games g, teams t
WHERE (HOME_TEAM_ID, VISITOR_TEAM_ID) IN (SELECT t1.TEAM_ID, t2.TEAM_ID
                                FROM teams t1 CROSS JOIN teams t2
                                WHERE t1.TEAM_ID!=t2.TEAM_ID AND t1.NICKNAME='Lakers'
                                          AND HOME_TEAM_WINS=1 AND SEASON=2019)
        AND g.VISITOR_TEAM_ID=t.TEAM_ID
```

We select the NICKNAME of the opposing Lakers team, the date of the game and the result. In the WHERE clause there is a condition on multiple attributes: on the home team and on the visitor team. The subquery has a cross join between two copies of the table "teams" with the condition that the home team must be the Lakers and must differ from the visitor team. Finally, the SEASON must be 2019 and the home team must have won; this last condition is expressed by the boolean variable HOME_TEAM_WINS set equal to 1.

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

```
SELECT PLAYER_NAME, ROUND(AVG(PTS),3) AS AVERAGE_PTS
FROM players p JOIN games_details gd ON p.PLAYER_ID=gd.PLAYER_ID
GROUP BY p.PLAYER_ID
HAVING AVG(PTS)>=ALL(SELECT AVG(PTS) FROM players p JOIN games_details gd ON
                                p.PLAYER_ID=gd.PLAYER_ID GROUP BY gd.PLAYER_ID)
```

I select the PLAYER_NAME and the average points per game using these two tables. Here you can see that there is a nested query in the HAVING clause and there is also the ALL operator. This is used to select among all the players in the database, only the one with the highest average points per game. The execution time of this query is about 5 and a half seconds.

9) *for Western Confederation teams, return the players and corresponding 2021 games in which they scored more than 20% of the points compared to the total points scored by the two teams during the game*

```
SELECT PLAYER_NAME, t1.NICKNAME AS HOME_TEAM, t2.NICKNAME AS
            VISITOR_TEAM, g.SEASON, PTS_home+PTS_away AS PTS_MATCH, PTS AS
            PTS_PLAYER, ROUND(PTS/(PTS_home+PTS_away),3) AS P_PTS
FROM games_details gd, games g, teams t1, teams t2, players p, ranking r
WHERE gd.GAME_ID=g.GAME_ID AND HOME_TEAM_ID=t1.TEAM_ID
            AND VISITOR_TEAM_ID=t2.TEAM_ID AND p.PLAYER_ID=gd.PLAYER_ID
            AND (t1.TEAM_ID=r.TEAM_ID OR t2.TEAM_ID=r.TEAM_ID)
            AND CONFERENCE='West' AND g.SEASON=2021
GROUP BY gd.GAME_ID, p.PLAYER_NAME
HAVING P_PTS>0.2
```

This query requires information present in all 5 tables of the database. We select information about the players and the teams. In particular, we select the total number of points scored by the two teams, the points scored by the player and the corresponding percentage on the total number of points.

A first naïve formulation of this query would be to insert all 5 tables in the FROM clause, then express the joins and the conditions in the WHERE clause. Finally, we use the aggregation condition on the percentage of points in the HAVING clause. This solution turns out however extremely inefficient because there are too many joins. This query took about 260 seconds.

3) OPTIMIZATION

8.1) A possible alternative to run the query number 8 and avoid a nested query in the HAVING clause is to use views. Accordingly, to try to optimize this query, I created a view with all the player names and their respective average points per game:

```
CREATE VIEW view1(PLAYER_NAME,AVERAGE_PTS) AS
            SELECT PLAYER_NAME, ROUND(AVG(PTS),3)
            FROM players p JOIN games_details gd ON p.PLAYER_ID=gd.PLAYER_ID
            GROUP BY p.PLAYER_NAME
```

Then I declared a query where I select the player in the view with the highest average points per game through a nested query in the WHERE clause. This query takes about 7 seconds and is more inefficient than the previous one:

```
SELECT *
FROM view1
WHERE AVERAGE_PTS=(SELECT MAX(AVERAGE_PTS) FROM view1)
```

So, I have tried to declare another view without any join and I replaced the PLAYER_NAME of the table "players" with the PLAYER_ID of "games_details":

```
CREATE VIEW view2(PLAYER_ID,AVERAGE_PTS) AS
            SELECT PLAYER_ID, ROUND(AVG(PTS),3)
            FROM games_details
            GROUP BY PLAYER_ID
```

The join between the two tables appears in the query itself. Doing so, the query is equivalent to the previous ones but takes less than a second to be executed. It is much more efficient:

```
SELECT PLAYER_NAME, AVERAGE_PTS
FROM players p JOIN view2 v2 ON p.PLAYER_ID=v2.PLAYER_ID
WHERE AVERAGE_PTS=(SELECT MAX(AVERAGE_PTS) FROM view2)
GROUP BY p.PLAYER_ID
```

9.1) A first possible optimization for the query number 9 is to not include the table "ranking" in the FROM clause and to avoid a join with this table. The relationships between the two tables "teams" t1 and t2 with "ranking" are defined through two subqueries with the IN operator. This query takes only 0.25 seconds.

```
SELECT PLAYER_NAME, t1.NICKNAME AS HOME_TEAM, t2.NICKNAME AS
            VISITOR_TEAM, g.SEASON, PTS_home+PTS_away AS PTS_MATCH, PTS AS
            PTS_PLAYER, ROUND(PTS/(PTS_home+PTS_away),3) AS P_PTS
FROM games_details gd, games g, teams t1, teams t2, players p
WHERE gd.GAME_ID=g.GAME_ID AND HOME_TEAM_ID=t1.TEAM_ID
           AND VISITOR_TEAM_ID=t2.TEAM_ID AND p.PLAYER_ID=gd.PLAYER_ID
           AND (t1.TEAM_ID IN (SELECT r.TEAM_ID FROM ranking r WHERE
                   CONFERENCE='West' GROUP BY r.TEAM_ID)
                   OR t2.TEAM_ID IN (SELECT r.TEAM_ID FROM ranking r WHERE
                   CONFERENCE='West' GROUP BY r.TEAM_ID)) AND g.SEASON=2021
GROUP BY gd.GAME_ID, p.PLAYER_NAME
HAVING P_PTS>0.2
```

9.2) Another alternative is to keep the query as it was but instead of selecting the entire table "ranking" and creating joins on that, you select a table aggregated by TEAM_ID. As you can see this solution is slightly faster than the previous one (0.17 seconds). This difference is even more evident if we eliminate the SEASON condition: the first one would take 11 seconds, while the second one would take around 5 seconds.

```
SELECT PLAYER_NAME, t1.NICKNAME AS HOME_TEAM, t2.NICKNAME AS
            VISITOR_TEAM, g.SEASON, PTS, PTS_home+PTS_away AS PTS_MATCH, PTS
            AS PTS_PLAYER, ROUND(PTS/(PTS_home+PTS_away),3) AS P_PTS
FROM games_details gd, games g, teams t1, teams t2, players p, (SELECT r.TEAM_ID
         FROM ranking r WHERE CONFERENCE='West' GROUP BY r.TEAM_ID) r
WHERE gd.GAME_ID=g.GAME_ID AND HOME_TEAM_ID=t1.TEAM_ID
          AND VISITOR_TEAM_ID=t2.TEAM_ID AND p.PLAYER_ID=gd.PLAYER_ID
          AND (t1.TEAM_ID=r.TEAM_ID OR t2.TEAM_ID=r.TEAM_ID) AND
g.SEASON=2021
GROUP BY gd.GAME_ID, p.PLAYER_NAME
HAVING P_PTS>0.2
```

Lastly, we want to see what happens to two of the queries if we remove the constraints on the primary and foreign keys. To do this I declared a database identical to the one used so far but without keys and called it nba2.

The query I have tested is the third one, which took only 0.25 seconds with the keys. In the not optimized version it takes around 2 seconds:

```
SELECT DISTINCT PLAYER_NAME
FROM players2 p, games2 g,
         (SELECT PLAYER_ID, gd.GAME_ID FROM games_details2 gd JOIN games2 g ON
         gd.GAME_ID=g.GAME_ID WHERE SEASON>=2009 AND SEASON<=2018) AS gd1
WHERE p.PLAYER_ID=gd1.PLAYER_ID AND g.GAME_ID=gd1.GAME_ID AND
           PTS_home-PTS_away>30
```

The second query is the 9.2, that passes from about 0.2 seconds to around 3 seconds:

```
SELECT PLAYER_NAME, t1.NICKNAME AS HOME_TEAM, t2.NICKNAME AS
            VISITOR_TEAM, g.SEASON, PTS, PTS_home+PTS_away AS PTS_MATCH,
            PTS, ROUND(PTS/(PTS_home+PTS_away),3) AS P_PTS
FROM games_details2 gd, games2 g, teams2 t1, teams2 t2, players2 p, (SELECT
          r.TEAM_ID FROM ranking2 r WHERE CONFERENCE='West' GROUP BY
          r.TEAM_ID) r
WHERE gd.GAME_ID=g.GAME_ID AND HOME_TEAM_ID=t1.TEAM_ID
          AND VISITOR_TEAM_ID=t2.TEAM_ID AND p.PLAYER_ID=gd.PLAYER_ID
          AND (t1.TEAM_ID=r.TEAM_ID OR t2.TEAM_ID=r.TEAM_ID) AND
g.SEASON=2021
GROUP BY gd.GAME_ID, p.PLAYER_NAME
HAVING P_PTS>0.2
```

To finish, in the excel file that I leave attached, I have reported the times of execution of all the queries in their not optimized version and in their optimized version with the keys.

In all cases you can observe a significant improvement in performance. Only for two queries the performances are worse. This is probably due to the fact that these two queries use joins between variables that are not declared as foreign keys in the first database.