Data Management Project

SQLITE VS NEO4J

Antonio Grieco 1715585
Eleonora Beatrice Rossi 18*****

GOALS

01 Compare efficency of the two Database Management System based on Time Analysis

02 Show how and why the two DBMS performs better in what situation.

03 Evaluation performed using different queries, some that performs better in Relational Systems and some that performs better in NoSQL systems.

TOOLS

SQLite

We used SQLite as our Relational DBMS.

Neo4j

We used Neo4j as our NoSQL DMBS.

Python

We used Python to convert the DataSet in a Graph Database and to perform queries on both the Databases.

CypherShell

We used CypherShell to interact with Neo4j remotely.

DATASET

For the DataSet we chosed the following one :

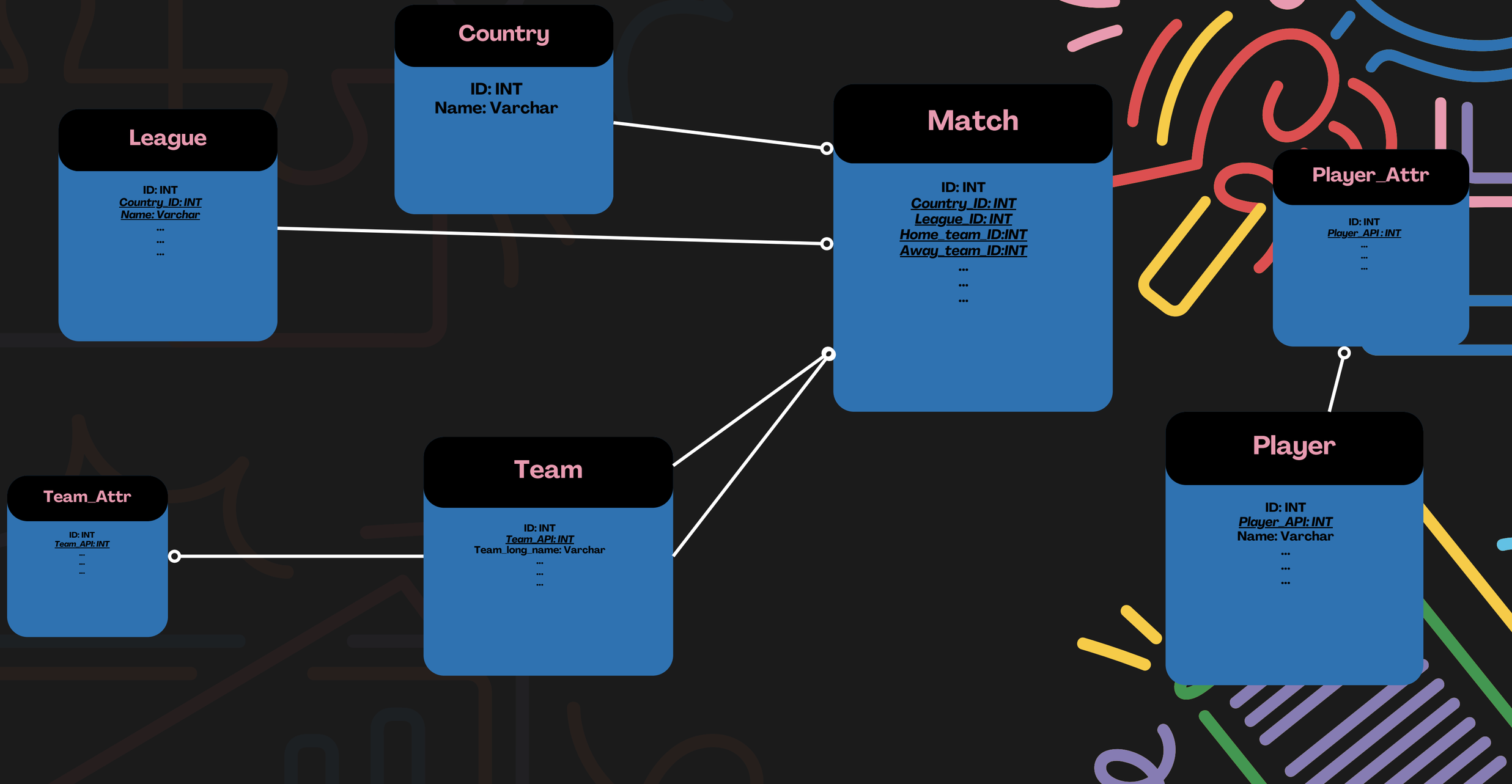
https://www.kaggle.com/datasets/hugomathien/soccer

It is a DataSet that contains information on Soccer Teams and the matches that they played in Europe.

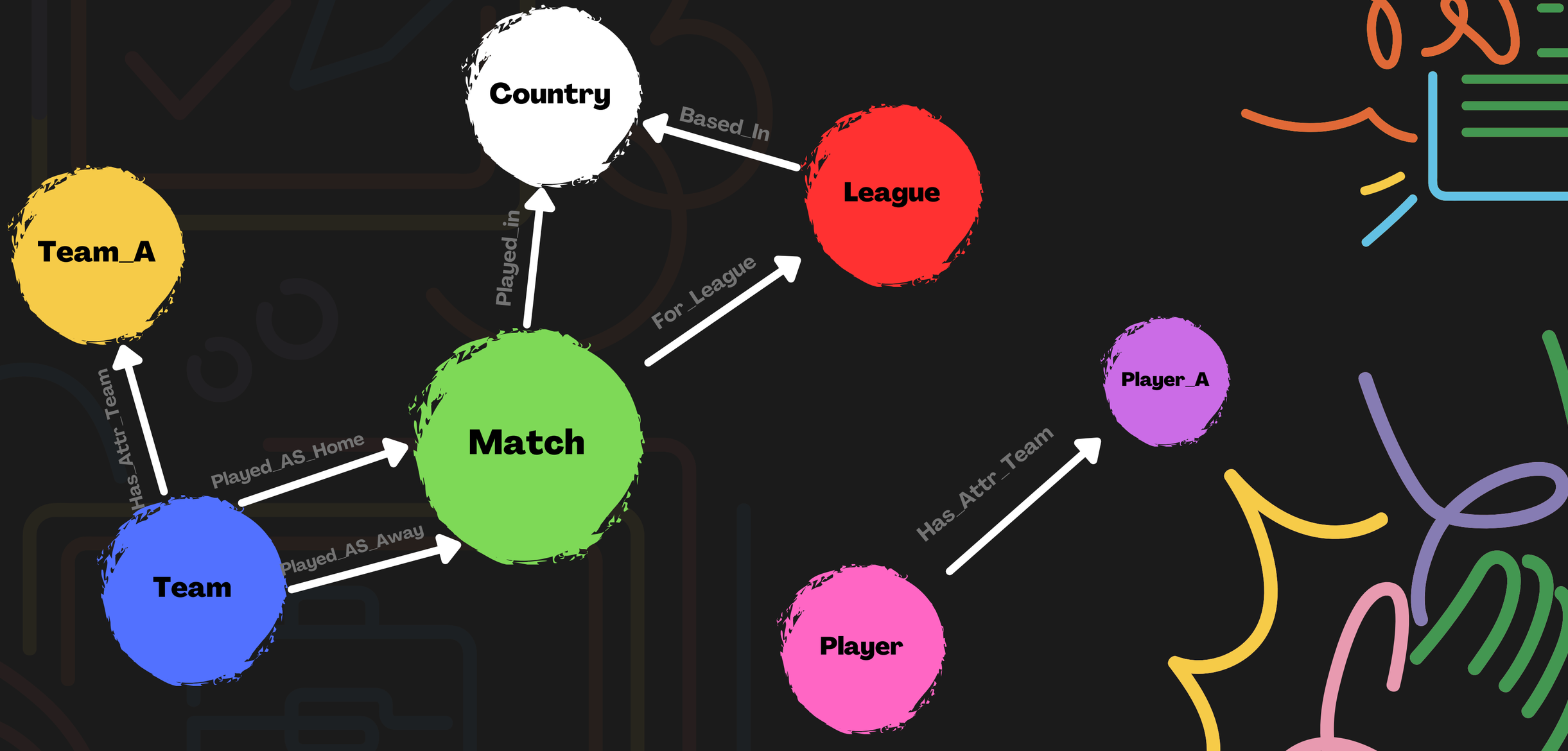
More in detail:

- **Country**
- **League**
- **Match**
- **Player**
- **Player_Attributes**
- **Team**
- **Team_Attributes**

ENTITY RELANTIONSHIP SCHEMA



GRAPH DATABASE



GRAPH DATABASE CREATION

01 **To create the Graph Database we used a Python Script to read all the Tables of the Database and creating a Node for each Row.**

02 **Once all the nodes has been created it’s time for the relationships:**

- To achieve that we matched the values associated at the Foreign Keys of the relevant Tables and then created an edge between the two nodes.
- In our case we have just Directed Edges, but this factor depends on the structure of the DB.

NODE CREATIONS

Below we have an example of how the script works for the creation of the nodes:

```
for table in table_names:
    data = cur.execute("SELECT * FROM " + table)
    col_names = [col[0] for col in cur.description]
    print("Parsing table " + table + ". Column names: " + str(col_names))
    print("\n\n")
    for row in data:
        row_dict = {}
        for i, att in enumerate(col_names):
            row_dict[att] = row[i]
        #Creare un nodo per ogni riga e inserirlo nel database Neo4j
        query = f"CREATE (n:{table} $props)"
        session.run(query, props=row_dict)
    print(f"Finished inserting data from the {table} table.")
```

RELATIONSHIPS

Below we can see an example for the creation of the Relationships:

```
#Creare la relazione HAS_ATTRIBUTES_PLAYER tra Player e Player_Attributes
 query = """
 MATCH (p:Player), (pa:Player_Attributes)
 WHERE p.player_api_id = pa.player_api_id
 CREATE (p)-[:HAS_ATTRIBUTES_PLAYER]->(pa)
 """
 session.run(query)
 print("Created HAS_ATTRIBUTES_PLAYER relationships.")
```

QUERIES

We performed 12 queries, some easy ones and some more complex, we chosed those to highlight strenghtness of the two systems so we will find some that performs in similar ways and some that are much more faster in one system with respect to the other.

Criterion for the queries are based on the nature of the two DBMS:

- SQLite: Aggregation, Join on multiple connections.
- GraphDatabase: Multiple Join operation(Traverse the Graph)

QUERY 1

Return number of teams that have played at least one game in a Belgium League.

SQLite

```
SELECT distinct count(*)
FROM match, league, team, country
WHERE country.name = "Belgium" AND country.id = league.country_id AND league.id = match.league_id
AND (match.home_team_api_id = team.team_api_id OR match.away_team_api_id = team.team_api_id)
GROUP BY country.name
```

Neo4j

```
MATCH (t:Team)-->(:Match)-[:FOR_LEAGUE]->(:League)-[:BASED_IN]->(:Country {name:'Belgium'})
RETURN count(t);
```

QUERY 2

Return name of the team that won more matches

SQLite

```
SELECT team.team_long_name, count(*) as vinte
FROM match, team
WHERE (match.home_team_api_id = team.team_api_id AND match.home_team_goal > match.away_team_goal) OR
(match.away_team_api_id = team.team_api_id AND
match.away_team_goal > match.home_team_goal)
GROUP BY team.team_long_name
ORDER BY vinte desc
LIMIT 1;
```

Neo4j

```
MATCH (s:Team)-[r]->(m:Match)
WHERE (type(r) = 'PLAYED_AS_HOME' AND m.home_team_goal>m.away_team_goal)
OR (type(r) = 'PLAYED_AS_AWAY' AND m.away_team_goal > m.home_team_goal)
RETURN s.team_long_name, count(s) as vinte
ORDER BY vinte DESC LIMIT 1;
```

QUERY 3

Return number of matches played by every team in Italy ordered by highest to lowest

SQLite

```
SELECT team.team_long_name, count(*)
FROM country, match, team
WHERE country.id = match.country_id AND country.name = "Italy"
AND (match.home_team_api_id = team.team_api_id OR match.away_team_api_id = team.team_api_id)
GROUP BY team.team_long_name
ORDER BY count(*) DESC
```

Neo4j

```
MATCH (t:Team)-[r]->(:Match)-[:PLAYED_IN]->(:Country {name:'Italy'})
WHERE type(r) = 'PLAYED_AS_AWAY' OR type(r) = 'PLAYED_AS_HOME'
RETURN t.team_long_name,count(t) ORDER BY count(t) DESC
```

QUERY 4

Return the chain of home-away team (Casa - Trasferta --> Trasferta - Casa) ordered by date of the first match

SQLite

```
SELECT DISTINCT m1.date,t1.team_long_name, t2.team_long_name
FROM team as t1, team as t2, match as m1, match as m2
WHERE  (m1.away_team_api_id = m2.home_team_api_id) AND t2.team_api_id = m2.home_team_api_id
AND t1.team_api_id = m1.home_team_api_id
ORDER BY m1.date
```

Neo4j

```
MATCH (t1:Team)-[:PLAYED_AS_HOME]->(m1:Match)<-[:PLAYED_AS_AWAY]-(t2:Team)-[:PLAYED_AS_HOME]->(m2:Match)
RETURN DISTINCT m1.date,t1.team_long_name,t2.team_long_name ORDER BY m1.date
```

QUERY 5

Add a new Column 'Total_Goals' that count total goals in each match

SQLite

```
ALTER TABLE match
ADD total_goals SMALLINT;
UPDATE match
SET total_goals = home_team_goal + away_team_goal;
```

Neo4j

```
MATCH (m:Match) SET m.total_goals = m.away_team_goal + m.home_team_goal;
```

QUERY 6

Drop 'crossing' column in Player_Attributes

SQLite

```
ALTER TABLE player_attributes
DROP COLUMN crossing
```

Neo4j

```
MATCH (pa:Player_Attributes) REMOVE pa.crossing
```

QUERY 7

Return first 5 Players with the highest 'Overall' stat

SQLite

```
SELECT p.player_name, AVG(pa.overall_rating) AS avg_rating
FROM Player_Attributes pa
JOIN Player p ON pa.player_api_id = p.player_api_id GROUP BY p.player_api_id
ORDER BY avg_rating DESC LIMIT 5;
```

Neo4j

```
MATCH (p:Player)-[:HAS_ATTRIBUTES_PLAYER]->(pa:Player_Attributes)
RETURN p.player_name AS player, AVG(pa.overall_rating) AS avg_rating
ORDER BY avg_rating DESC
LIMIT 5;
```

QUERY 8

Return total number of matches played in each country

SQLite

```
SELECT c.name AS country, COUNT(*) AS match_count
FROM Country c
JOIN MATCH m ON c.id = m.country_id
GROUP BY c.name;
```

Neo4j

```
MATCH (c:Country)
OPTIONAL MATCH (c)<-[:PLAYED_IN]-(:Match)
WITH c, COUNT(*) AS match_count
RETURN c.name AS country, match_count;
```

QUERY 9

Return Average value of (HOME,AWAY) made goals in 2008/2009 season for every league.

SQLite

```
SELECT l.name AS league_name,
AVG(m.home_team_goal) AS avg_home_goals, AVG(m.away_team_goal) AS avg_away_goals
FROM Match m JOIN League l ON m.league_id = l.id
WHERE m.season = '2008/2009'
GROUP BY l.name;
```

Neo4j

```
MATCH (m:Match)-[:FOR_LEAGUE]->(l:League)
RETURN l.name, SUM(m.home_team_goal) AS total_home_goals, SUM(m.away_team_goal)
AS total_away_goals
ORDER BY l.name;
```

ISSUES & DISCOVERY

During the analysis we encountered some strange situation.

- Cache : First execution of each query is way more slow than all the others, thanks to the cached data.
- Execution timings : to correctly measure the actual execution times it is important to remember that if you don’t read the rows returned by the cursor object, then the rest of your statements won’t be executed. The solution was to call the function fetchall().
- Correctness : In the case were we have to execute multiple statements in the same execution, as the add column query, we have to use the function executescript() instead of execute() that can execute only one at time.

TIME ANALYSIS

Query	SQLite	Neo4j
01	84,56 ms	12,04 ms
02	133,92 ms	358,23 ms
03	80,17 ms	32,36 ms
04	5315,48 ms	403,29 ms
05	20826,62 ms	1547,89 ms

TIME ANALYSIS

Query	SQLite	Neo4j
06	2128,18 ms	2453,82 ms
07	573,87 ms	318,80ms
08	90,22 ms	28,20 ms
09	82,32 ms	337,37 ms

FINAL DISCUSSION

SQLite performs better in situations where there is some kind of Data Aggregation,a clear example is the query 9, where we have to calculate an Average Value.

Obviously Neo4j it's perfect and very efficient when the major need is traversing a graph, that is represented in relational systems with multiple joins.

SQLite also performs better in situations where we have Join operations on multiple conditions, a clear example is the query #10 where we have 3 joins and SQLite performs also better than Neo4j. This is given by query planning and SQLite optimization.

Neo4j instead performs MUCH better in situations where we alter some Tables, as we can see in the Query # 7
In query 6 the result is very strange, we was expecting something similar to the situation of Query 7 but Neo4j it's worst in terms of time, SQLite have to purge data after the removing but apparently it not affect the result so much.

THANK YOU!

GitHub

Antonio Grieco
Eleonora Beatrice Rossi