

# KATHMANDU UNIVERSITY

DHULIKHEL, NEPAL

Department of Computer Science & Engineering (DoCSE)



[NoSQL Lab Assignment](#)

COMP-232

Submitted By:

**Sambeg Shrestha (45)**

Computer Engineering

2<sup>nd</sup> Year, 2<sup>nd</sup> Semester

Submitted to:

**Mr. Santosh Khanal**

7 Aug, 2020

## Acknowledgment

I would like to express my deepest gratitude to all those who provided me with the possibility to complete this report. A special gratitude to our subject teacher, Mr. Santosh Khanal, whose elucidating lectures on the database system and practical classes in the application of Neo4j NoSQL DBMS, helped me a lot in understanding and implementation of non-relational graph database language Neo4j. Furthermore, I would like to acknowledge my peers who helped me in understanding the topic and guidance they imparted me throughout implementation of the different laboratory works.

## Abstract

Database Management System (DBMS) is a software package designed to define, manipulate, retrieve and manage data in a database. As computer science students, learning the fundamentals of DBMS is most paramount. The Lab study done under COMP 202 was in order get acquainted with the fundamentals of one of the most popular and upcoming Non-relational DBMS technology: Graph Database using Neo4j, and to understand the operations done in a database. Using basic Neo4j cypher language queries we learned the Data Definition, Manipulation and Querying. In conclusion, data storage and manipulation is the most fundamental requirement of a computing technology and technologies like Neo4j are the most structured, systematic and convenient method to manage a database

# Chapter 1 Introduction

## 1.1 Database

A database is an organized collection of data, generally stored and accessed electronically from a computer system. Complex databases are often developed using formal design and modeling techniques.

## 1.2 Database Management System (DBMS)

DBMS is the software that interacts with end users, applications, and the database itself to capture and analyze the data. The sum total of the database, the DBMS and the associated applications can be referred to as a "database system". Often the term "database" is also used to loosely refer to any of the DBMS, the database system or an application associated with the database.

## 1.3 NoSQL Database

NoSQL in simple colloquial terms means “non SQL” or “not only SQL” and is used to refer to any non-relational database. These database store data in any other form other than table. A non-relational database is a database that does not use the tabular schema of rows and columns found in most traditional database systems. Instead, non-relational databases use a storage model that is optimized for the specific requirements of the type of data being stored. For example, data may be stored as simple key/value pairs, as JSON documents, or as a graph consisting of edges and vertices.

### 1.3.1 Advantages

Its known advantages are:

- **Elastic scalability:** These databases are designed for use with low-cost commodity hardware.
- **Big Data Applications:** Massive volumes of data are easily handled by NoSQL databases.

- **Economy:** Relational Databases require installation of expensive storage systems and proprietary servers, while NoSQL databases can be easily installed in cheap commodity hardware clusters as transaction and data volumes increase. This means that you can process and store more data at much less cost.
- **Dynamic schemas:** NoSQL databases need no schemas to start working with data. In Relational Database you have to define a schema first, making things more difficult because you have to change the schema everytime the requirements change.
- **Auto-sharding:** Relational Databases scale vertically, which means you often have a lot of databases spread across multiple servers because of the disk space they need to work. NoSQL databases usually support auto-sharding, meaning that they natively and automatically spread data across an arbitrary number of servers, without requiring the application to even be aware of the composition of the server pool.
- **Replication:** Most NoSQL databases also support automatic database replication to maintain availability in the event of outages or planned maintenance events. More sophisticated NoSQL databases are fully self-healing, offering automated failover and recovery, as well as the ability to distribute the database across multiple geographic regions to withstand regional failures and enable data localization.
- **Integrated caching:** Many NoSQL technologies have excellent integrated caching capabilities, keeping frequently-used data in system memory as much as possible and removing the need for a separate caching layer

### 1.3.2 NoSQL Types

Some of the different types of NoSQL database are: document databases, key-value-databases, wide-column stores, and graph databases.

- **Document databases** store data in documents similar to JSON (JavaScript Object Notation) objects. Some popular document based database are *MongoDB*, *Elasticsearch*, *MarkLogic*, *OrientDB*, etc
- **Key-value databases** are a simpler type of database where each item contains keys and values *Redis* and *DynanoDB* are popular key-value databases.
- **Wide-column stores** store data in tables, rows, and dynamic columns. *Cassandra* and *HBase* are two of the most popular wide-column stores.
- **Graph databases** store data in nodes and edges. *Neo4j* and *JanusGraph* are examples of graph databases.

## 1.4 Neo4j Graph Database

Neo4j is a graph database management system developed by Neo4j, Inc. Described by its developers as an ACID-compliant transactional database with native graph storage and processing, Neo4j is available in a GPL3-licensed open-source "community edition", with online backup and high availability extensions licensed under a closed-source commercial license. Neo also licenses Neo4j with these extensions under closed-source commercial terms.

Neo4j is implemented in Java and accessible from software written in other languages using the Cypher query language through a transactional HTTP endpoint, or through the binary "bolt" protocol.

## Chapter 2 Methodology

The Lab Study was performed using the following elucidated chronological structure.

### 2.1 Installation and Configuration

#### 2.1.1 System

Linux (Ubuntu 20.04)

#### 2.1.2 Steps

1. Getting the executable sourcefile:

The Appimage, a portable executable, file for linux system was obtained from <https://neo4j.com/download/> .

2. Setting up executable:

The appimage upon download cannot be run as an app, for this we execute the following bash command

A terminal window with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The command `chmod -x neo4j-desktop-offline-1.3.3-x86_64.AppImage` is displayed in a light green monospace font.

```
chmod -x neo4j-desktop-offline-1.3.3-x86_64.AppImage
```

*Figure 1: Bash code for making appimage executable*

This make the file executable and the app can be launched by simply double clicking. The following window then appears

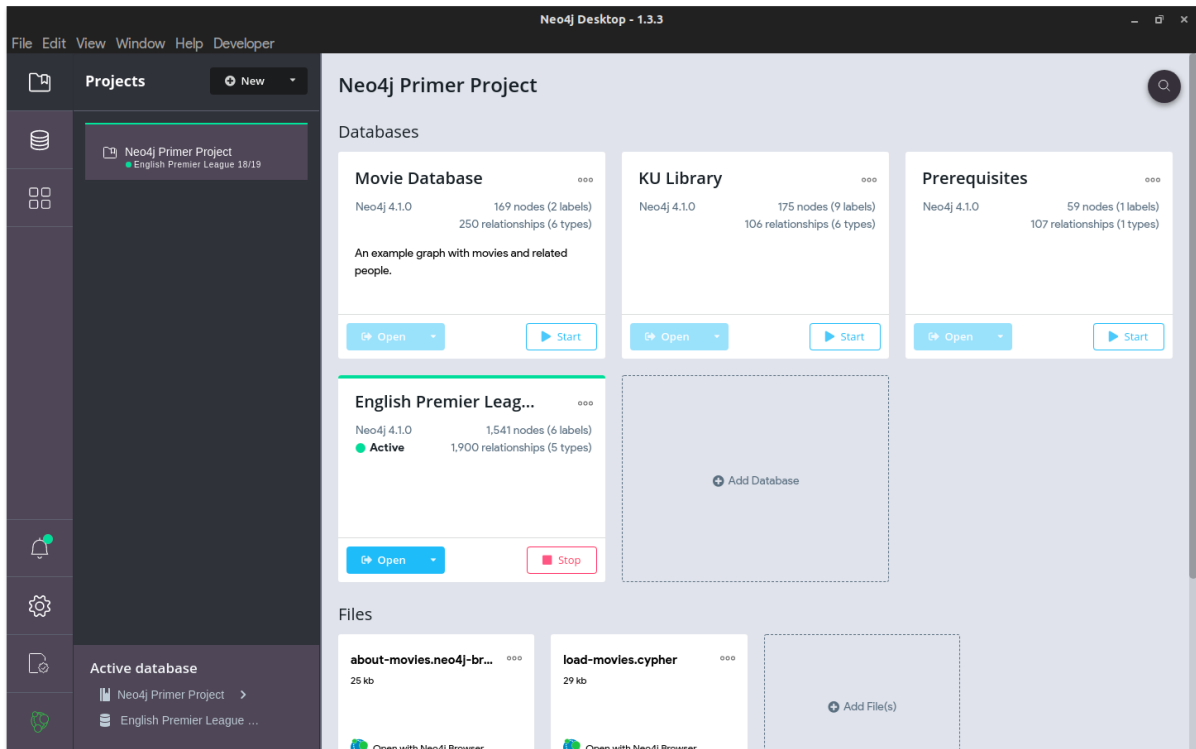
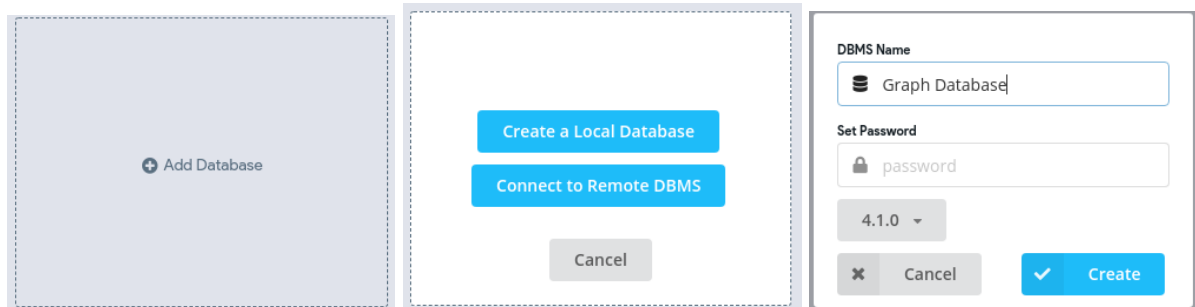


Figure 2: Neo4j Startup window dashboard

From here one can easily create Databases

## 2.2 Creating a Database

Following the GUI, we can easily create a Database



We then setup the English Premier League Database.

## 2.3 Populating the Database

We use the data set provided in CSV format for our database. To do this first we need to copy the CSV file to the import folder of the created neo4j database.

Location of import folder in my system:



`/home/sambeg/.config/Neo4j Desktop/Application/neo4jDatabases/database-7625f7ff-4914-4d92-8975-a2d0c9ef6cc2/installation-4.1.0/import`

So we copy the dataset to this location in order to import it in our database.

```
cp epl.csv /home/sambeg/.config/Neo4j Desktop/Application/neo4jDatabases/database-7625f7ff-4914-4d92-8975-a2d0c9ef6cc2/installation-4.1.0/import
```

Figure 3: Command for copying file

### 2.3.1 Loading CSV

The CSV file was imported with headers into a *RawData* Node in order to make things simpler and to not import the CSV multiple times.

```
LOAD CSV
WITH HEADERS FROM 'file:///epl.csv' AS row
MERGE (n:RawData {
  Div:row.Div,
  Date: row.Date,
  HomeTeam: row.HomeTeam,
  AwayTeam: row.AwayTeam,
  FullTimeHomeGoal: row.FTHG,
  FullTimeAwayGoal: row.FTAG,
  FullTimeResult: row.FTR,
  HalfTimeHomeGoal: row.HTHG,
  HalfTimeAwayGoal: row.HTAG,
  HalfTimeResult: row.HTR,
  Referee: row.Referee,
  HomeShot:row.HS,
  AwayShot:row. AS,
  HomeShotTarget:row.HST,
  AwayShotTarget:row.AST,
  HomeFoul: row.HF,
  AwayFoul:row.AF,
  HomeCorner:row.HC,
  AwayCorner: row.AC,
  HomeYellow:row.HY,
  AwayYellow:row.AY,
  HomeRed: row.HR,
  AwayRed:row.AR
});
```

Figure 4: Loading CSV content into RawData

Here, we load only the required data from the CSV with their headers.

### 2.3.2 Cleaning String Data into Date

Since the Date element is string data and we require Temporal Date data readable by neo4j, we clean the data as follows

```
// UPDATE THE STRING DATE OF RAW DATA TO NEO4J READABLE DATE
MATCH (n:RawData)
WITH [item IN split(n.Date, "/") | toInteger(item)] AS dateComponents, n
SET n.Date = date({day: dateComponents[0], month: dateComponents[1], year: dateComponents[2]})
RETURN n;
```

Figure 5: Cleaning string to date data type

Here, the split command splits the raw data into different part based on the “/” character. So, for example it splits “2018/11/13” into three parts “2018”, “11”, “13” and stores it into an array *dateComponents*. Then we create a *date* object and assign *year*, *month* and *day* values to the object using the previous array and save it as *Date* in the *RawData*.

### 2.3.3 Creating Nodes and Relationships

#### 2.3.3.1 Division

It is the division of the football league as we know there are different division in club football in England, for instance first division, second division, Premier League.

```
//Division of League
MATCH (n:RawData)
WITH DISTINCT n.Div AS Div
CREATE (d:Division {
  name: Div,
  season: "2018/2019"
});
```

Figure 6: Creating Division

Here, since the dataset is of only one division, we extract the division from distinct *Div* property of *RawData* nodes which we aliased as “*n*” and assign it as property *name* to a new node *Division* along with property *season* as 2018/2019 since the dataset is of said season.

### 2.3.3.2 Team

It is the particular instance of a group of sportsmen and staff under one management who function as a unit.

```
MATCH (n:RawData)
WITH DISTINCT n.HomeTeam AS HT
CREATE (d:Team {
  name: HT,
  matches_played: 0,
  won: 0,
  loss: 0,
  draw: 0,
  points: 0,
  home:0,
  away:0
});
```

Figure 7: Creating Team

### 2.3.3.3 Matchday

It is the particular instance of a matchup that takes place between two teams

```
//Matchday
MATCH (n:RawData)
CREATE (d:Matchday {
  title:n.HomeTeam+" vs. "+n.AwayTeam,
  Date:n.Date,
  HT:n.HomeTeam,
  AT:n.AwayTeam,
  REF:n.Referee
});
```

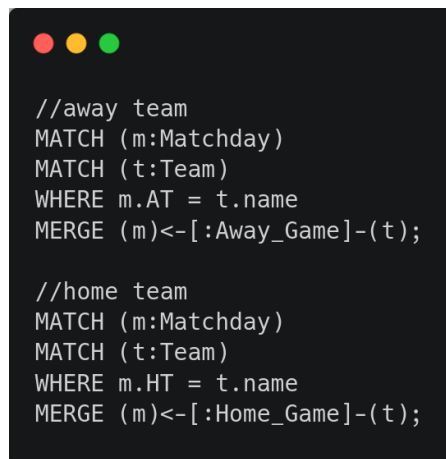
Figure 8: Creating Matchday

Here the property *HomeTeam* of *RawData* consists of teams who play matches at home. So it contains all the possible teams playing matches. So, we extract **distinct** values from this property as with alias HT and create new Node *Team* with property *name* as the team name and assign it HT. Other properties include *matches\_played*, *won*, *loss*, *draw*, *points*, *home*, and *away* which are used for ranking purposes.

#### 2.3.3.4 Assigning Teams to Matchdays

Since particular match consists of participation of two teams: home and away teams, we create relationship between team and matchday.

Relationship graph pattern:  $(:Matchday) \leftarrow [ :Home\_Game / Away\_Game ] - (:Team)$



```
//away team
MATCH (m:Matchday)
MATCH (t:Team)
WHERE m.AT = t.name
MERGE (m) <- [ :Away_Game ] - (t);

//home team
MATCH (m:Matchday)
MATCH (t:Team)
WHERE m.HT = t.name
MERGE (m) <- [ :Home_Game ] - (t);
```

Figure 9: Creating relationship of team and matchday

Here, we create relationships named 'Away\_Game' and 'Home\_Game' from *Team* to *Matchday* nodes for home and away teams by using WHERE condition.

#### 2.3.3.5 Creating Result

Each match will produce certain outcomes like wins and losses. We show this by creating a new node and making a relation to Matchday.

Pattern:  $(:Matchday) - [ :has\_Results ] \rightarrow (:Results)$

```

MATCH (n:RawData)
MATCH (m:Matchday)
WHERE n.Date = m.Date AND n.HomeTeam = m.HT AND n.AwayTeam = m.AT
MERGE (m)-[:has_Results]->(d:Results {
  FTHG:n.FullTimeHomeGoal,
  FTAG:n.FullTimeAwayGoal,
  FTR:n.FullTimeResult,
  HTHG:n.HalfTimeHomeGoal,
  HTAG:n.HalfTimeAwayGoal,
  HTR:n.HalfTimeResult
});

```

Figure 10: Creating Results

Here, we create a relationship 'has\_Results' from *Matchday* to new node *Results* with different properties like FTHG, FTAG, FTR, etc by using WHERE clause to specify exact results from *RawData* corresponding to said matchday.

#### 2.3.3.6 Creating Statistics

Like Results, Each match will consist of certain stats like fouls and cards. We show this by creating a new node and making a relation to Matchday.

Pattern: (:Matchday)-[:has\_Statistics]->(:Stats)

```

//Statistics
MATCH (n:RawData)
MATCH (m:Matchday)
WHERE n.Date = m.Date AND n.HomeTeam = m.HT
MERGE (m)-[:has_Statistics]->(d:Stats {
  HS:n.HomeShot,
  AS :n.AwayShot,
  HST:n.HomeShotTarget,
  AST:n.AwayShotTarget,
  HF:n.HomeFoul,
  AF:n.AwayFoul,
  HC:n.HomeCorner,
  AC:n.AwayCorner,
  HY:n.HomeYellow,
  AY:n.AwayYellow,
  HR:n.HomeRed,
  AR:n.AwayRed
});

```

Figure 11: Creating statistics

Here, we create a relationship 'has\_Results' from *Matchday* to new node *Stats* with different properties like HS, AS, etc by using WHERE clause to specify exact results from *RawData* corresponding to said matchday.

## 2.4 Queries

### 2.4.1 Show all the EPL teams involved in the season

Cypher Query:

```

//1
MATCH (n:RawData)
WITH DISTINCT n.HomeTeam AS Team
RETURN Team ORDER BY Team ASC;

```

Here, using MATCH we get all the data from Node RawData, and using WITH clause we alias all the distinct teams filtered by DISTINCT from HomeTeam property of RawData node as Team and display them in Ascending order using RETURN and ORDER BY ASC.

**Result:**



The image shows a software interface with a sidebar on the left containing three icons: a table icon labeled 'Table', a large letter 'A' icon labeled 'Text' (which is highlighted), and a code icon labeled 'Code'. To the right of the sidebar is a table with 18 rows. The first row has a header 'Team'. The subsequent 17 rows list football teams in ascending alphabetical order: Arsenal, Bournemouth, Brighton, Burnley, Cardiff, Chelsea, Crystal Palace, Everton, Fulham, Huddersfield, Leicester, Liverpool, Man City, Man United, Newcastle, Southampton, Tottenham, Watford, and West Ham.

Team
"Arsenal"
"Bournemouth"
"Brighton"
"Burnley"
"Cardiff"
"Chelsea"
"Crystal Palace"
"Everton"
"Fulham"
"Huddersfield"
"Leicester"
"Liverpool"
"Man City"
"Man United"
"Newcastle"
"Southampton"
"Tottenham"
"Watford"
"West Ham"

Figure 12: Query 1 Result

## 2.4.2 How many matches were played on Mondays

Cypher Query:

```
//2.  
MATCH (n:Matchday)  
WHERE n.Date.DayOfWeek = 1  
RETURN COUNT(n) as Toatal_matches_on_Monday;
```

Here, we select Matchday node using MATCH, then using WHERE clause filter the nodes where DayOfWeek is 1 ie. Node Temporal object Date consists of a function named 'DayOfWeek' which returns the weekday numbering from 1 – 7, with 1 being Monday and 7 being Sunday. So by filtering the node with weekday 1 we get all matches played on Monday. Using Return and aggregate function COUNT we return the exact number of matches.

**Result:**



The screenshot shows the Neo4j Cypher query interface. At the top, the query is entered: `neo4j$ MATCH (n:Matchday) WHERE n.Date.DayOfWeek = 1 RETURN COUNT(n) as Toatal_matches_on_Monday;`. Below the query, there are three tabs: 'Table', 'Text', and 'Code'. The 'Table' tab is selected, and it displays a table with one column named 'Toatal\_matches\_on\_Monday' and one row with the value '17'.

"Toatal_matches_on_Monday"
17

Figure 13: Query 2 Result



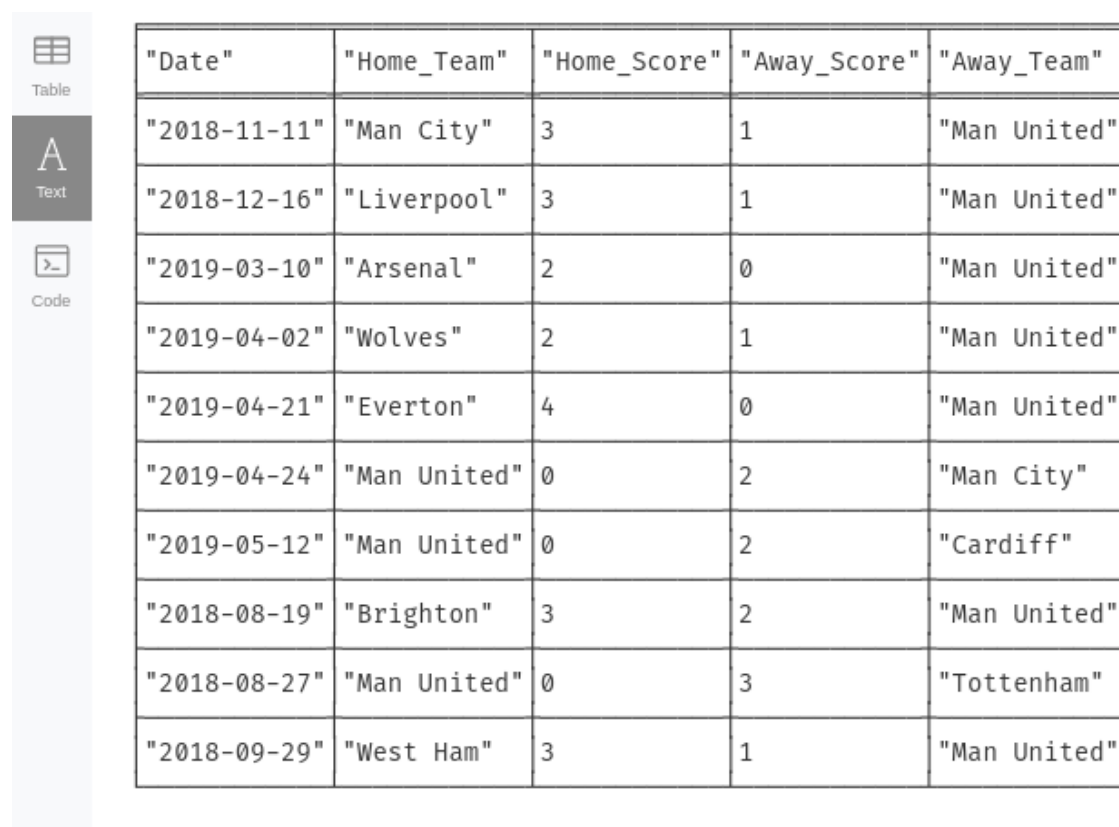
### 2.4.3 Display all the matches where Manchester United Lost

Cypher Query:

```
//3.
MATCH p=(m:Matchday)-[:has_Results]->(r:Results)
WHERE (m.HT="Man United" AND r.FTR="A") OR (m.AT="Man United" AND r.FTR="H")
RETURN m.Date as Date, m.HT as Home_Team, toInteger(r.FTHG) as Home_Score, toInteger(r.FTAG) as Away_Score, m.AT as Away_Team;
```

Here, we select the digraph (:Matchday)-[:hasResult]→(:Results) for all the matches and its results. Using WHERE we filter node matchday where the team is “Man United” and results where they lost. This is done by tallying if Home Team (HT) is Man Utd and for them to lose Full Time Result (FTR) is Away win (A), and similar with Man United as Away Team. Then we return the Match info.

Result:



The screenshot shows a database interface with a sidebar on the left containing icons for 'Table', 'Text', and 'Code'. The 'Table' view is selected, displaying a table with 5 columns: 'Date', 'Home\_Team', 'Home\_Score', 'Away\_Score', and 'Away\_Team'. The table contains 11 rows of match data where Manchester United lost.

"Date"	"Home_Team"	"Home_Score"	"Away_Score"	"Away_Team"
"2018-11-11"	"Man City"	3	1	"Man United"
"2018-12-16"	"Liverpool"	3	1	"Man United"
"2019-03-10"	"Arsenal"	2	0	"Man United"
"2019-04-02"	"Wolves"	2	1	"Man United"
"2019-04-21"	"Everton"	4	0	"Man United"
"2019-04-24"	"Man United"	0	2	"Man City"
"2019-05-12"	"Man United"	0	2	"Cardiff"
"2018-08-19"	"Brighton"	3	2	"Man United"
"2018-08-27"	"Man United"	0	3	"Tottenham"
"2018-09-29"	"West Ham"	3	1	"Man United"

Figure 14: Query 3 Result

## 2.4.4 Display all matches that “Liverpool” won but were down in the first half.

Cypher Query:

```
//4.
MATCH p=(m:Matchday)-[:has_Results]->(r:Results)
WHERE
(m.HT="Liverpool" AND r.HTR="A" AND r.FTR="H" )
OR
(m.AT="Liverpool" AND r.HTR="H" AND r.FTR="A")
RETURN m.Date AS Date, m.AT AS Away_Team , toInteger(r.HTAG) AS HT_Away_Goal, toInteger(r.FTAG) AS FT_Away_Goal,
m.HT AS Home_Team, toInteger(r.HTHG) AS HT_Home_Goal, toInteger(r.FTHG) AS FT_Home_Goal
```

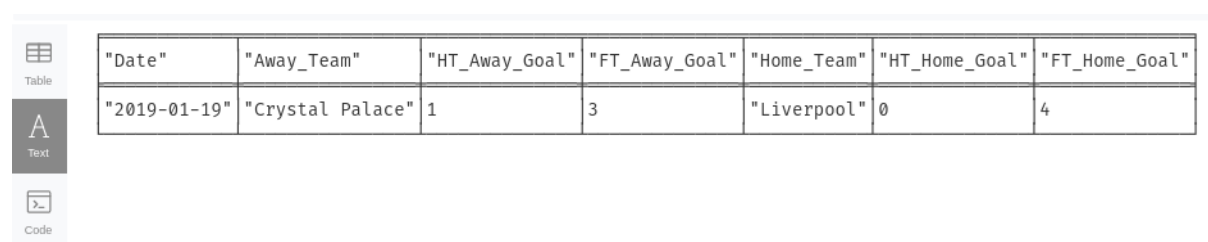
Here, we select the digraph (:Matchday)-[:hasResult]→(:Results) for all the matches and its results. Using WHERE we filter node matchday where the team is “Liverpool” and results where they lost in Half-time however then won in Full-time.

From (m.HT="Liverpool" AND r.HTR="A" AND r.FTR="H" ) filter we get matches where Liverpool are home team and result where home team lost at halftime but home team won at full time.

Similarly, From (m.AT="Liverpool" AND r.HTR="H" AND r.FTR="A" ) filter we get matches where Liverpool are away team and result where away team lost at halftime but away team won at full time.

Then, we return match information accordingly.

**Result:**

A screenshot of a database interface showing a table with 7 columns: Date, Away\_Team, HT\_Away\_Goal, FT\_Away\_Goal, Home\_Team, HT\_Home\_Goal, and FT\_Home\_Goal. The table contains one row of data. On the left side of the interface, there are three icons: a table icon labeled 'Table', a text icon labeled 'Text', and a code icon labeled 'Code'.

"Date"	"Away_Team"	"HT_Away_Goal"	"FT_Away_Goal"	"Home_Team"	"HT_Home_Goal"	"FT_Home_Goal"
"2019-01-19"	"Crystal Palace"	1	3	"Liverpool"	0	4

Figure 15: Query 4 Result.

## 2.4.5 Write a query to display the final ranking of all the teams based on their total points.

Cypher Query:

- Set Win/Loss/Draw record

```
//5.

//SET win/loss home
//home win
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "H" AND t.name = m.HT
SET t.won = (t.won+1), t.points = (t.points+3), t.matches_played = (t.matches_played+1), t.home=(t.home+1);
//away loss
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "H" AND t.name = m.AT
SET t.loss = (t.loss+1), t.matches_played = (t.matches_played+1);

//SET win/loss away
//away win
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "A" AND t.name = m.AT
SET t.won = (t.won+1), t.points = (t.points+3), t.matches_played = (t.matches_played+1), t.away=(t.away+1);
//home Loss
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "A" AND t.name = m.HT
SET t.loss = (t.loss+1), t.matches_played = (t.matches_played+1);

//SET draw home
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "D" AND t.name = m.HT
SET t.draw = (t.draw+1), t.points = (t.points+1), t.matches_played = (t.matches_played+1);
//SET away draw
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "D" AND t.name = m.AT
SET t.draw = (t.draw+1), t.points = (t.points+1), t.matches_played = (t.matches_played+1);
```

Summarizing here we simply check win, loss, and draws in each matchday and update properties of team according to the result.

In a match, Either a team wins then other team loses or both teams draw.

According to our data set, this is further specialized as if a home team wins, the away team loses and if away team wins, home team loses. Also, if home team draws away team draws and vice versa.

Hence, according to the number of wins and draws, a team is rewarded with points. 3 for winning and 1 for drawing.

So, according to our modeling, breaking down this process as below:

### Home Team Win, Away Team Loss

```
//SET win/loss home
//home win
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "H" AND t.name = m.HT
SET t.won = (t.won+1), t.points = (t.points+3), t.matches_played = (t.matches_played+1), t.home=(t.home+1);
//away loss
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "H" AND t.name = m.AT
SET t.loss = (t.loss+1), t.matches_played = (t.matches_played+1);
```

Here, in this code we set win/loss record with condition of home win and away loss.

Firstly for team that won,

#### 1. *MATCH (m:Matchday)-->(r:Results)*

- First we select nodes Matchday and corresponding results,

#### 2. *MATCH (m1:Matchday)-[:Away\_Game|:Home\_Game]-(t:Team)*

- then select teams who play home or away game for matches.

#### 3. *WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "H" AND t.name = m.HT*

- we filter using WHERE clause such that the matchday selected from above two MATCH statements by tallying their *Date*, *HT*, *AT*
- then filter those records where the **home team won** by checking FTR=H.
- From second MATCH we filter the team by checking they are the **home team**. This results in records with only matches, records, teams where home team won

4. *SET t.won = (t.won+1), t.points = (t.points+3), t.matches\_played = (t.matches\_played+1), t.home=(t.home+1);*

- Here we update the properties of the home team for winning. Their match\_played of home winning team is incremented by 1, won is also incremented by 1 and points is incremented by 3 since winning gives 3 points.

Then For Losing Team,

5. *MATCH (m:Matchday)-->(r:Results)*

6. *MATCH (ml:Matchday)-[:Away\_Game|:Home\_Game]-(t:Team)*

- Same as home winning case, we select match, results, team

7. *WHERE (m.Date = ml.Date AND m.HT = ml.HT AND m.AT = ml.AT) AND r.FTR = "H" AND t.name = m.AT*

- Same as home team winning, we filter using WHERE clause such that the matchday selected from above two MATCH statements by tallying their *Date, HT, AT*
- then filter those records where the **away team lost** by checking FTR="H".
- From second MATCH we filter the team by checking they are the **away team**. This results in records with only matches, records, teams where **away team loss**

8. *SET t.loss = (t.loss+1), t.matches\_played = (t.matches\_played+1);*

- Here we update the properties of the away team for losing. The *match\_played* of away losing team is incremented by 1, loss is also incremented by 1 and points is unchanged since losing gives 0 points.

**Away team Win, Home Team Loss**

```
//SET win/loss away
//away win
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "A" AND t.name = m.AT
SET t.won = (t.won+1), t.points = (t.points+3), t.matches_played = (t.matches_played+1), t.away=(t.away+1);
//home Loss
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "A" AND t.name = m.HT
SET t.loss = (t.loss+1), t.matches_played = (t.matches_played+1);
```

Same process as above section where home team won and away team lost, only difference is roles of away and home teams are reversed ie **away team won and home team lost**.

## Both Team Draw

```
//SET draw home
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "D" AND t.name = m.HT
SET t.draw = (t.draw+1), t.points = (t.points+1), t.matches_played = (t.matches_played+1);
//SET away draw
MATCH (m:Matchday)-->(r:Results)
MATCH (m1:Matchday)-[:Away_Game|:Home_Game]-(t:Team)
WHERE (m.Date = m1.Date AND m.HT = m1.HT AND m.AT = m1.AT) AND r.FTR = "D" AND t.name = m.AT
SET t.draw = (t.draw+1), t.points = (t.points+1), t.matches_played = (t.matches_played+1);
```

Similar process for selecting matchdays, teams and results

Filtering with WHERE, this time instead of wins we filter records where FTR=D ie Draw.

Then we set properties for Drawing a match, Here we update the properties of the first the home team and then the away team successively for drawing. The *match\_played* of the teams are incremented by 1, *draw* is also incremented by 1 and points is incremented by 1 since losing gives 1 points.


- **Finally Displaying the Output Point table**

Based on the records updated in above query, we display the points table as

```
//Points Table
MATCH (t:Team)
RETURN t.name AS Team, t.matches_played AS Matches_Played, t.won AS Won, t.loss AS Lost, t.draw AS Draw, t.points AS Points
ORDER BY t.points DESC;
```

Here, we get the Team node which consists of property *points* updated by previous queries. Then we display properties accordingly in descending order of *points*.

### Result:



"Team"	"Matches_Played"	"Won"	"Lost"	"Draw"	"Points"
"Man City"	38	32	4	2	98
"Liverpool"	38	30	1	7	97
"Chelsea"	38	21	8	9	72
"Tottenham"	38	23	13	2	71
"Arsenal"	38	21	10	7	70
"Man United"	38	19	10	9	66
"Wolves"	38	16	13	9	57
"Everton"	38	15	14	9	54
"West Ham"	38	15	16	7	52
"Leicester"	38	15	16	7	52
"Watford"	38	14	16	8	50
"Crystal Palace"	38	14	17	7	49
"Bournemouth"	38	13	19	6	45
"Newcastle"	38	12	17	9	45
"Burnley"	38	11	20	7	40
"Southampton"	38	9	17	12	39
"Brighton"	38	9	20	9	36
"Cardiff"	38	10	24	4	34
"Fulham"	38	7	26	5	26

Figure 16: Query 5 Result

## Chapter 3 Conclusion

Hence, Neo4j is an excellent NoSQL technology for database management. It is effective, convenient and very systematic at organizing and managing a database. The clear command syntaxes make the code easy to read.

Therefore, NoSQL are an excellent and convenient technology and Neo4j resides on the top of such technology especially Graph Database.