



PROJET BDS

---

## Impémentation d'une base de données graphe dans Neo4J

---

*Présenté par :*

HENNI AMAR

RSMOUKI ABDELLAH

Année universitaire : 2021/2022

# Introduction

Notre projet est composé en 4 parties :

- Partie 1 : Choix et import d'un jeu de données, dans cette partie on va expliquer les étapes des traitements fait à notre dataset choisi sur Kaggle et on va détailler l'importation de ces données vers Neo4J et PostgreSQL
- Partie 2 : Réalisation de requêtes Cypher afin d'intéroger notre base de données précédemment créée
- Partie 2(partie complémentaire) : Creation de notre propre base de données afin de tester les requêtes cypher récursive et essayer de trouver une requête SQL plus efficace qu'une requête Cypher à expressivité équivalente
- Partie 3 : Analytique de graphe. Dans cette partie on va faire tournée deux algorithme de graphe datascience sur nos ensembles de données à savoir l'algorithme Closeness et K-Nearest Neighbors et nous allons détailler comme l'interface de Bloom nous a aidé dans notre projet.
- Partie 4 : Partie libre : Dans cette partie libre on a intégré notre base de donnée Neo4J dans une application Web React.js au moyen du driver **use-neo4j**

## Partie 1 : Choix et import d'un jeu de données

Dans notre projet on a choisie de travailler sur les matches de football disputé des **Championnat d'Europe de football**, afin de réalisée cela on a choisi un dataset sur **Kaggel**.

### Informations générale sur notre dataset :

La structure du football européen est complexe. Bien que l'instance dirigeante de l'UEFA supervise les compétitions continentales, les nations individuelles sont en mesure de gérer leurs ligues et coupes nationales de la manière la mieux adaptée au pays. Cela a créé une gamme de ligues différentes. Dans notre Dataset il y'a tout les matchs jouées entre 2008 et 2016 dans les différentes leagues européennes et donc dans différents pays européens.

### Information générale sur notre Dataset utilisée

- Lien : <https://www.kaggle.com/hugomathien/soccer>  
Contenu de notre Dataset :
  - Le dataset est sous format d'une base de donnée sqlite qui contient les tableaux suivant : Match, Team, League, Country ...  
Pour la réalisation de notre travail on a extrèses ces données sous forme de fichiers .CSV pour faire nos traitement dessus (Match.csv, Team.csv, League.csv, Country.csv)

— Nombre de lignes :

- Dans Match.csv : **25979** lignes
- Dans Team.csv : **299** lignes
- Dans Country.csv : **11** lignes
- Dans League.csv : **11** lignes

— Nombre de colonnes :

- Dans Match.csv : **116 colonnes** dans ce fichier on utilisera un sous ensemble de colonnes seulement. (on verra pourquoi cette désision dans la partie traitement et Importation des données)
  - Dans Team.csv : **5 colonnes**
  - Dans Country.csv : **2 colonnes**
  - Dans League.csv : **3 colonnes**
- Nombre de seanson jouées présents dans notre Dataset : (entre 2008 to 2016)

## Traitement et Importation des données

Notre dataset est sous format d'une base de données **sqlite**, afin de les importer dans PostegreSQL et Neo4J et à la même occasion de traiter nos données avec python on les à extrait vers un format **CSV**.

Pour ce faire on a utilisés les librairies : pandas, sqlite3

**Etape 1 : Connexion à la base de données sqlite et import des données sous forme de dataframes**

```
import sqlite3
import pandas as pd

# Create our connection.
cnx = sqlite3.connect('/content/drive/MyDrive/M2-DATA/Neo4J/Final-Project/database.sqlite')

df_Country = pd.read_sql_query("SELECT * FROM Country", cnx)
df_League = pd.read_sql_query("SELECT * FROM League", cnx)
df_Match = pd.read_sql_query("SELECT * FROM Match", cnx)
df_Team = pd.read_sql_query("SELECT * FROM Team", cnx)
```

## Etape 2 : Conversion des dataframes en fichiers csv

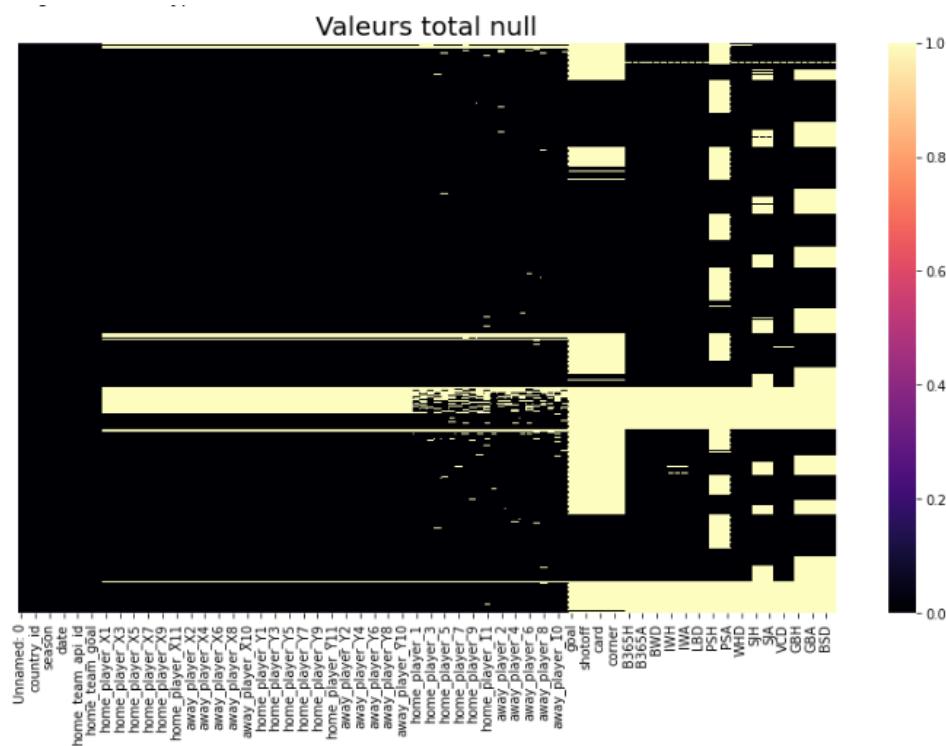
```
df_Country.to_csv('/content/drive/MyDrive/M2-DATA/Neo4J/Final-Project/Country.csv')
df_League.to_csv('/content/drive/MyDrive/M2-DATA/Neo4J/Final-Project/League.csv')
df_Match.to_csv('/content/drive/MyDrive/M2-DATA/Neo4J/Final-Project/Match.csv')
df_Team.to_csv('/content/drive/MyDrive/M2-DATA/Neo4J/Final-Project/Team.csv')
```

**Remarque :** On a pas choisi de modéliser les données des joueurs (players) car le dataset n'est pas complet à ce sujet. C'est à dire qu'au niveaux des matchs il y'a trop de données manquantes (vides au niveau des joueurs qui ont disputé les matchs. D'autres informations comme la possession dans les matchs sont erronées.

De plus, on n'a pas pris en compte les colonnes qui contient les joueurs et les clubs de l'api FIFA car justement vu qu'ont utilise pas les joueurs dans notre modélisation cela ne servirai à rien par rapport au types de requêtes que nous envisageons.

Afin que l'ont puisse bien visualisé le nombre de données manquantes des matches on a réliser le plot suivant qui repesente le nombre total de valeurs null dans le fichier **Match.csv** :

(voir les fichiers joins afin de regarder le code pour faire le plot **nom du fichier : plot\_emplacement\_vide.py**)



Aprés discution on a fait notre choix de colonnes à garder pour notre modélisation :

<b>country_id</b>	0
<b>league_id</b>	0
<b>season</b>	0
<b>stage</b>	0
<b>date</b>	0
<b>match_api_id</b>	0
<b>home_team_api_id</b>	0
<b>away_team_api_id</b>	0
<b>home_team_goal</b>	0
<b>away_team_goal</b>	0

Cette figure représente les colonnes qu'on a gardés et les valeurs associées représentent le nombre totale de valeurs null de ces colonnes dans notre dataset. Les valeurs sont toutes à 0 ce qui veut dire qu'il ne manque aucune données sur ces colonnes.

## Importation des données sur Neo4J

### Définitions de contraintes :

Avant d'importer nos données, on doit veiller à ce que ces dernières soient uniques dans notre base de donnée Neo4J et afin de garantir cela, ont à mis en place les contraintes suivantes :

```
CREATE CONSTRAINT ON (matches:Matches) ASSERT matches.id_match IS UNIQUE
CREATE CONSTRAINT ON (team:Team) ASSERT team.id_team IS UNIQUE
CREATE CONSTRAINT ON (league:League) ASSERT league.id_league IS UNIQUE
CREATE CONSTRAINT ON (country:Country) ASSERT country.id_country IS UNIQUE
```

On vérifie en suite que nos contraintes ont été bien prises en compte :

	name	description
1	"constraint_8adfd242"	"CONSTRAINT ON ( matches:Matches ) ASSERT (matches.id_match) IS UNIQUE"
2	"constraint_cf35e12b"	"CONSTRAINT ON ( team:Team ) ASSERT (team.id_team) IS UNIQUE"
3	"constraint_e29f44a4"	"CONSTRAINT ON ( league:League ) ASSERT (league.id_league) IS UNIQUE"
4	"constraint_ebc9cbd6"	"CONSTRAINT ON ( country:Country ) ASSERT (country.id_country) IS UNIQUE"

### L'importation des données

La démarche réalisé est expliquée pas à pas via les commentaire présents de le code de l'importation :

#### Part 1 : Utilisation du fichier Match.csv créée

- Creation des noeuds matches et leurs informations
- Creation des team (club) et leurs relationships avec les matches
- Creation des pays (country) et leurs relationships avec les matches
- Creation des leagues et leurs relationships avec les matches

```

//Insérer les données par lot de 10000 à la fois car toutes les données du dataset ne peut être chargé en mémoire directement
:auto USING PERIODIC COMMIT 10000

//Chargement du dataset avec les en-têtes
LOAD CSV WITH HEADERS FROM 'file:///Match.csv' AS line

//Split la date délimitée des matches par '-' et transmission de la variable de ligne
WITH split(line.date,'-') as date,line

//Affectation d'alias pour l'index 1 du tableau qui représente une année, le 2 représentant un mois et le 3 représentant un jour
//(on a utilisé substring pour garder que le jour sans les 00:00:00 à la fin)
WITH date[0] as year, date[1] as month, SUBSTRING(date[2],0,2) as day,line

//Collecte de toutes les données de matches pertinentes dans une liste avec la date dévisée en morceaux gérables
WITH collect([day,month,year,line.stage,line.home_team_api_id,line.away_team_api_id,line.home_team_goal,line.away_team_goal,
line.match_api_id,line.season,line.league_id,line.country_id]) as data

//Transformation de la liste en lignes individuelles
UNWIND data as allData

//Et creation d'un tableau avec tout les numéros des mois (2 caractères)
with allData, ["01","02","03","04","05","06","07","08","09","10","11","12"] as allMonthNames

//Fusion des noeuds leurs données en extrayants les données de la liste au dessus créé: (Si les noeud existent déjà,
//ils ne seront pas créés à nouveau)
MERGE (matches:Matches{id_match:allData[8],
season:allData[9], day:toInteger(allData[0]),
month:toInteger([i IN RANGE(0,SIZE(allMonthNames)-1) WHERE allMonthNames[i] = allData[1]][0]+1),
year:allData[2],
stage:allData[3],
homeTeam:allData[4],
awayTeam:allData[5],
homeTeamGoal:allData[6],
awayTeamGoal:allData[7],
league:allData[10],
country:allData[11]})  

WITH allData as allData, matches as matches

//Fusionner également les deux noeuds d'équipes relatifs à ce match (si des noeuds existent déjà, il ne seront pas créés à nouveau)
MERGE (team1:Team{id_team:allData[4]})  

MERGE (team2:Team{id_team:allData[5]})



//Fusionner les deux noeuds des leagues relatifs à ce match (si des noeuds existent déjà, il ne seront pas créés à nouveau)
MERGE (league:League{id_league:allData[10]})

//Fusionner les deux noeuds des pays relatifs à ce match (si des noeuds existent déjà, il ne seront pas créés à nouveau)
MERGE (country:Country{id_country:allData[11]})

//Définition des variables qui prennent en charge le nombre de but marquées pour les équipe à domicile et la variable qui prend le
//nombre de but marquées à l'extérieur
WITH toInteger(allData[6]) AS goalsScoredFullTimeHome,toInteger(allData[7]) AS goalsScoredFullTimeAway, matches AS matches,
team1 AS team1, team2 AS team2, league AS league, country AS country

//Les relationship suivantes entre les noeuds sont créées, leurs type c'est soit (WIN,LOST,EQUAL1,EQUAL2)
//en fonction des variables de score(buts) qui ont été créer précédemment
//Les but marquées à domicile et à l'extérieurs sont stockés respectivement dans la relation de chaque équipe avec le match comme
// indiqué dans la conception que l'ont détaillera plus tard
FOREACH (allData in CASE WHEN (goalsScoredFullTimeHome > goalsScoredFullTimeAway) THEN [1] ELSE [] END |
MERGE (team1)-[r1:WON{goalsScoredFullTime:goalsScoredFullTimeHome}]->(matches)<-[r2:LOST{goalsScoredFullTime:goalsScoredFullTimeAway}]->(team2))

FOREACH (allData in CASE WHEN (goalsScoredFullTimeHome < goalsScoredFullTimeAway) THEN [1] ELSE [] END |
MERGE (team1)-[r1:LOST{goalsScoredFullTime:goalsScoredFullTimeHome}]->(matches)<-[r2:WON{goalsScoredFullTime:goalsScoredFullTimeAway}]->(team2))

FOREACH (allData in CASE WHEN (goalsScoredFullTimeHome = goalsScoredFullTimeAway) THEN [1] ELSE [] END |
MERGE (team1)-[r1:EQUAL2{goalsScoredFullTime:goalsScoredFullTimeHome}]->(matches)<-[r2:EQUAL1{goalsScoredFullTime:goalsScoredFullTimeAway}]->(team2))

//On définit aussi les relations entre les matches et les leagues et entre les matches et les pays
MERGE (matches)-[r3:DISPUTED_IN]->(league)
MERGE (matches)-[r4:DISPUTED_INTO]->(country)

```

## Part 2 :

- Ajout des propriétés présentes dans le fichier **Team.csv** aux Teams (clubs)
- Ajout des propriétés présentes dans le fichier **Country.csv** aux country (pays)
- Ajout des propriétés présentes dans le fichier **League.csv** aux leagues

```

//Insert longname team property for all teams
LOAD CSV WITH HEADERS FROM 'file:///Team.csv' AS line
MERGE (team:Team{id_team:line.team_api_id})
SET team.long_name = line.team_long_name
return team

//Insert shortname team property for all teams
LOAD CSV WITH HEADERS FROM 'file:///Team.csv' AS line
MERGE (team:Team{id_team:line.team_api_id})
SET team.short_name = line.team_short_name
return team

//Insert country names
LOAD CSV WITH HEADERS FROM 'file:///Country.csv' AS line
MERGE (country:Country{id_country:line.id})
SET country.country_name = line.name
return country

//Insert League names
LOAD CSV WITH HEADERS FROM 'file:///League.csv' AS line
MERGE (league:League{id_league:line.country_id})
SET league.league_name = line.name
return league

```

## Importation des données sur Postgresql

Avant d'entamer Cette partie, on a voulu mentionner le travail qu'on a fait pour transformer un fichier csv en une base de données postgres qui contient trois tables(transfers, players et teams). On a décidé de ne pas continuer le travail avec ces données car on a pris le dataset de la documentation de neo4j ([https://guides.neo4j.com/football\\_transfers](https://guides.neo4j.com/football_transfers)). La partie d'importation des données dans postgres n'existe pas dans cette documentation.

On a séparer le dataset pour avoir une structure de données relationnelle. Après avoir analyser les colonnes qui existent dans le fichier csv, on a essayé de déviser les données en trois tables :

```

postgres=# -- list all tables:
postgres=# \d
              List of relations
 Schema |     Name      | Type | Owner
-----+-----+-----+-----
 public | clubs       | table | postgres
 public | players     | table | postgres
 public | transfers   | table | postgres
(3 rows)

```

On a déviser chaque ligne du fichier source en quatres lignes dans notre base de données relationnelle :

- **Players** : Une ligne dans la table players avec playerUri comme clés primaire.
- **Clubs** : deux ligne dans la table clubs( une pour buyerClub et l'autre pour sellerClub). De même, la clé primaire est le clubUri.
- **Transfers** : Une ligne pour la table transfers. Pour cette table, on a pris transferUri comme clé primaire et les colonnes playerUri, sellerClubUri et buyerClubUri comme des clés étrangères.

On a trouvé des doublants dans les deux tableaux Clubs et Players, donc on les a supprimer pour avoir une base de données cohérentes. On a aussi supprimer les lignes qui contient des clés primaires null.

```
COPY players FROM '/home/abdel/Documents/neo4j/players_dataset.csv' WITH (FORMAT csv);
COPY transfers FROM '/home/abdel/Documents/neo4j/transfers_dataset.csv' WITH (FORMAT csv);
COPY clubs FROM '/home/abdel/Documents/neo4j/clubs_dataset.csv' WITH (FORMAT csv);
```

Après cette configuration, on a essayé de créer des requêtes de jointures pour évaluer la rapidité de ce gestionnaire de base de données.

Une simple requête dans cette base de données est la sélection du nom du joueur et du nom du club acheteur pour un joueur spécifique. Comme montre la figure ci-dessous, on a fait une trois jointure pour arriver au résultat attendu :

```
SELECT pl.playerName, cl_bu.clubName
FROM transfers AS tr
INNER JOIN players AS pl
ON tr.playerUri = pl.playerUri
INNER JOIN clubs AS cl_se
ON cl_se.clubUri = tr.sellerClubUri
INNER JOIN clubs AS cl_bu
ON cl_bu.clubUri = tr.buyerClubUri
WHERE tr.playerUri = '/douglas-costa/profil/spieler/75615';
```

Le résultat de cette requête est le suivant :

1	Douglas Costa	Shakhtar D.

Et le temps d'exécution :

```
Successfully run. Total query runtime: 68 msec.
1 rows affected.
```

On a encore essayé la même requête sans spécifier un joueur :

```

SELECT pl.playerName, cl_bu.clubName
FROM transfers AS tr
INNER JOIN players AS pl
ON tr.playerUri = pl.playerUri
INNER JOIN clubs AS cl_se
ON cl_se.clubUri = tr.sellerClubUri
INNER JOIN clubs AS cl_bu
ON cl_bu.clubUri = tr.buyerClubUri
ORDER BY tr.timestamp
LIMIT 5;

```

1	Florent Sinama-Pongolle	Sporting CP
2	Keisuke Honda	CSKA Moscow
3	Younès Kaboul	Spurs
4	Alessandro Budel	Brescia
5	Douglas Costa	Shakhtar D.

Successfully run. Total query runtime: 128 msec.  
5 rows affected.

On voit clairement que la complexité des jointures affecte gravement le temps de réponse à la requête.

Pour ce qui concerne notre jeu de données actuel, on va tout d'abord séparer le dataset pour avoir une structure de données relationnelle. Après avoir analyser les columns qui existent dans les fichiers csv, on a essayé de déviser les données en quatre tables :

```

postgres=# -- list all tables;
postgres=# \d
      List of relations
 Schema |   Name    | Type  | Owner
-----+-----------+-----+-----
 public | countries | table | postgres
 public | leagues   | table | postgres
 public | matchs    | table | postgres
 public | teams     | table | postgres
(4 rows)

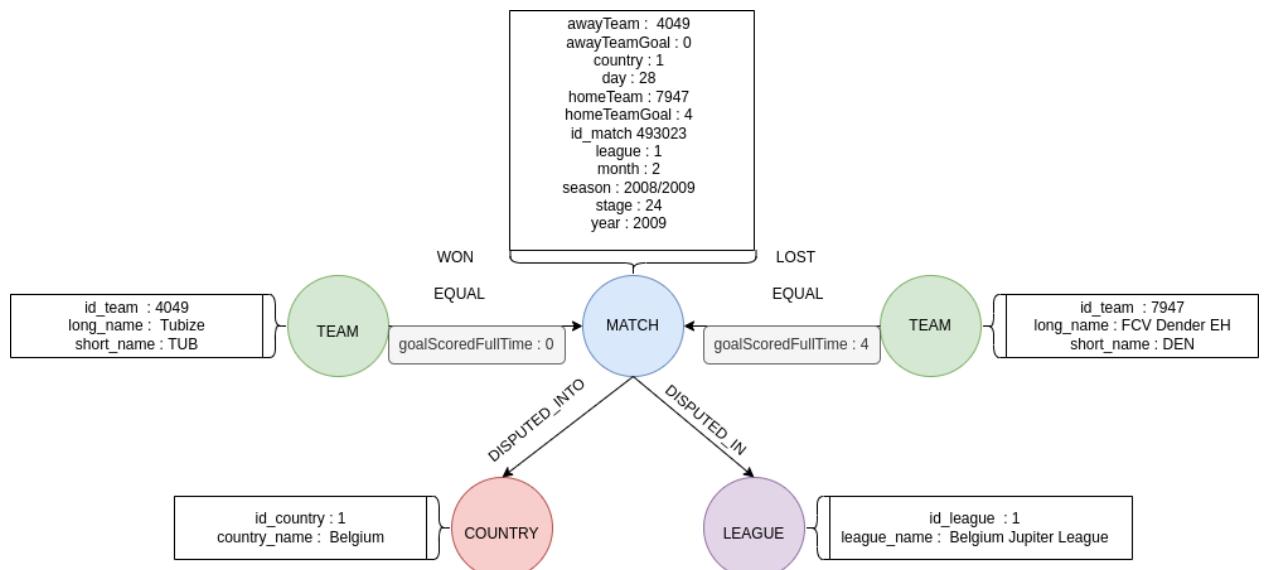
```

Pour cette modélisation, on a essayé de s'inspirer de la modélisation(en sqlite) pour pouvoir importer les données dans notre base de données postgres :

- **countries** : La table countries avec id comme clé primaire.
- **leagues** : La table leagues avec id comme la clé primaire et countryId comme clé étrangère.
- **matchs** : La table matchs avec la clé primaire id. Cette table contient une clé étrangère vers la table countries, une clé étrangère vers la table leagues et deux autres clés étrangères vers la table teams chacune spécifie le homeTeam/awayTeam.
- **teams** : La table teams avec id comme clé primaire. On a aussi ajouté une contrainte d'unicité sur la colonne teamApiId car c'est une clé étrangère dans la table matchs.

Comme on a fait pour la partie d'importation dans neo4j, on a éliminé toutes les colonnes qu'on ne va pas utiliser dans notre projet. Dans les ressources, les fichiers(ipynb) qui commencent par data\_preparation\_\* sont les fichiers qui contiennent le code qu'on a utilisé pour nettoyer ces données avant l'importation dans postgres.

Avant de passer à la deuxième partie et réaliser nos requêtes Cypher voici une représentation générale de nos noeuds et les relationship entre eux :



## Partie 2 : Requêtes Cypher

Durant cette partie on va réalisé 10 requêtes Cypher sur la base de données Neo4J Crée (des matches de football disputé **des Championnat d'Europe de football** entre 2008 et 2016).

### Requête 1 :

```
2 //La première requête est pour avoir la valeur maximal de victoire
3 MATCH (team:Team)-[r:WON]→(matches:Matches{month:0o1})
4 //Compter les victoires
5 WITH team, COUNT(r) as count
6 //Garder la valeur maximale
7 WITH MAX(count) as max
8 //Nouvelle requête
9 MATCH (team:Team)-[r:WON]→(matches:Matches{month:0o1})
10 WITH team, COUNT(r) as count, max
11 //Ne garder que les résultats dont le nombre de victoires est égal au max
12 WHERE count = max
13 RETURN team.long_name as `Long Team Name`, team.short_name as `Short Team Name`, count as `Goals On January`
```

"Long Team Name"	"Short Team Name"	"Goals On January"
"Real Madrid CF"	"REAA"	29

### Requête 2 :

```
1 //Requête 2 : Affichage des top 5 équipes qui à marqué le plus de but
2 //Correspondre tous les matchs et les équipes
3 MATCH (team:Team)-[r]→(matches:Matches)
4 //retourner le nom de l'équipe et la somme totale des but de chaque équipe
5 //Le score total est une somme des buts marqué présent dans les relationship avec l'attribut "goalsScoredFullTime"
6 WITH sum(r.goalsScoredFullTime) AS allGoals, team AS team
7 RETURN team.long_name AS `Team Name`, allGoals AS `All Goals`
8 //Ordonner par rapport au nombre de buts marqué du score le plus élevé au plus bas et limiter les résultats à 5
9 ORDER BY allGoals DESC LIMIT 5
```

"Team Name"	"All Goals"
"FC Barcelona"	849
"Real Madrid CF"	843
"Celtic"	695
"FC Bayern Munich"	653
"PSV"	652

### Requête 3 :

```
1 //Requête 3 : Affichage des 5 top équipes avec la pire défense.
2 //Faire correspondre les matchs et les deux équipes en même temps connectées à ce match
3 MATCH (team1:Team)-[r1]-(matches:Matches)-[r2]-(team2:Team)
4 //Puisque nous voulons afficher la pire défense, c'est-à-dire les équipes qui ont reçu le plus de buts
5 //nous devons additionner les buts de la team2 qui résident dans r2
6 //mais nous montrerons le nom de la team1
7 WITH sum(r2.goalsScoredFullTime) AS allGoals, team1 AS team1, team2 AS team2
8 RETURN team1.long_name AS `Team Name`, allGoals AS `All Goals`
9 ORDER BY allGoals DESC LIMIT 5
```

"Team Name"	"All Goals"
"Dundee United"	78
"FC Zürich"	73
"Aberdeen"	72
"Motherwell"	67
"Kilmarnock"	64

### Requête 4 :

```
1
2 //Requête 4 : Faire correspondre toutes les équipes et tous les matchs
3 //((une équipe à la fois) et n'obtenir le nombre de points des victoires et des nuls
4 //des matches des 10 première équipes (avec le plus grand score) à la saison 2015/2016
5 MATCH
6 (team:Team)-[r:WON|EQUAL1|EQUAL2]-(matches:Matches)
7 //collecter les types des relations rassemblées dans un tableau
8 //et transmettre également le nom de l'équipe
9 WITH collect(type(r)) as matchResult, team.long_name as teamName, matches.season as season
10 //parcourir le tableau des types de relations
11 WHERE matches.season = '2015/2016'
12 WITH reduce(data = {points: 0, teamName: teamName}, rel IN matchResult |
13 //quand une victoire est trouvée
14 CASE WHEN rel = "WON"
15 THEN {
16     //augmenter la variable "points" de 3
17     points: data.points + 3
18 }
19 ELSE {
20     //si (EQUAL1 ou EQUAL2) est trouvé augmenter la variable "points" de 1
21     points: data.points + 1
22 }
23 END
24 ) AS result, teamName, season
25 //Construire le tableau
26 UNWIND result.points as points
27 RETURN points, teamName ,season
28 ORDER BY points DESC
29 LIMIT 10
```

```
neo4j$ //Requête 4 : Faire correspondre toutes les équipes et tous les matchs
```

"points"	"teamName"	"season"
96	"Paris Saint-Germain"	"2015/2016"
91	"Juventus"	"2015/2016"
91	"FC Barcelona"	"2015/2016"
90	"Real Madrid CF"	"2015/2016"
88	"FC Bayern Munich"	"2015/2016"
88	"SL Benfica"	"2015/2016"
88	"Atlético Madrid"	"2015/2016"
86	"Sporting CP"	"2015/2016"
86	"Celtic"	"2015/2016"
84	"PSV"	"2015/2016"

## Requête 5 :

```
1 //Requête 5 : Afficher les équipes qui ont subit des défaites avec le score le plus bas
2 //La première requête consiste uniquement à obtenir la valeur maximale des pertes
3 MATCH (team:Team)-[r:LOST]→(matches:Matches)
4
5 //Compter les victoires
6 WITH team, COUNT(r) as count
7
8 //Garder la valeur maximale
9 WITH MAX(count) as max
10
11 //Nouvelle requête
12 MATCH (team:Team)-[r:LOST]→(matches:Matches)
13 WITH team, COUNT(r) as count, max
14 //Ne garder que les résultats dont le nombre de pertes est égal au max
15 WHERE count = max
16 RETURN team.long_name as `Team Name`, count as `Losses`
```

```
neo4j$ //Requête 5 : Afficher l'équipe qui a subit le plus de défaites avec le score le plus bas
```

"Team Name"	"Losses"
"Kilmarnock"	142

## Requête 6 :

```
1 //Requête 6: Afficher l'équipe avec le plus de victoire consécutives
2 //obtenir les victoires, les défaites et les nuls de chaque équipe
3 MATCH(team:Team)-[r:WON|LOST|EQUAL1|EQUAL2]-(matches:Matches)
4
5 //traiter les types de relations, transmettre le nom et
6 //créer un tableau à partir du mois et du jour (les deux nombres)
7 // cela est nécessaire car les résultats DOIVENT être classés en fonction de la date
8 WITH type(r) as matchResult, team.long_name as teamName, collect([matches.month, matches.day]) as list
9
10 //créer une liste de dates (mois , moisJour)
11 UNWIND list AS datesList
12
13 //transmettre les variables pour un traitement ultérieur
14 WITH teamName, matchResult, datesList
15
16 //Ordonnée tout par les dates
17 ORDER BY datesList
18 //REDUCE effectuera une action pour chaque élément de la liste des types de relations
19 RETURN teamName, REDUCE(s = {bestStreak: 0, currentStreak: 0}, result IN COLLECT(matchResult) |
20 //if WON is found
21 CASE WHEN result = "WON"
22 THEN {
23
24 //la variable bestStreak représente le plus grand nombre de victoires consécutives pour une équipe, tout au long de la boucle
25 //le currentStreak compte le nombre de victoires consécutives entre les dates
26 //une fois que le currentStreak + 1 est plus grand que le score élevé précédent, le meilleurStreak est mis à jour avec le nouveau score élevé
27 //sinon ça reste pareil
28 bestStreak: CASE WHEN s.bestStreak < s.currentStreak + 1 THEN s.currentStreak + 1 ELSE s.bestStreak END,
29 currentStreak: s.currentStreak + 1
30 }
31
32 //si c'est autre chose qu'une victoire, bestStreak est enregistré et la séquence actuelle est annulée car cette séquence est interrompue
33 ELSE {bestStreak: s.bestStreak, currentStreak: 0}
34 END
35
36 //Return le bestStreak
37 //ordonnée par lui et limiter par 1 pour obtenir une seul équipe
38 ).bestStreak AS result
39
40 ORDER BY result DESC LIMIT 1;
```

"teamName"	"result"
"Real Madrid CF"	17

## Requêtes avec index :

Nous avons tournés quelques requêtes similaire en ajoutant l'index et nous avons remarqué une différence au niveau du temps d'exécution, après l'ajout d'index les requêtes sont plus rapides.

## Partie 2 (complémentaire) : Requêtes récursives en Cypher et en PostgreSQL

Pour la suite des requêtes notamment pour les requêtes récursives nous considérons que notre base de données n'est pas adapté pour ce type de requête car il n'y à pas trop de chemins de longeurs variables.

Afin d'utiliser ce type de requêtes et afin de bien comprendre on a pris l'initiative de crée une base de données nous même sous forme de fichier **CSV**

### Description de notre BDD :

On a pensez à illustrez le concept de descendance dans les familles c'est à dire les lien Parent - Enfant

Afin de réaliser cela on a utilisées des prénoms et des identifiants des parents et des enfants au hasard.

**Remarque :** On a pas pris un grand jeu de données car on voulait bien constater les choses et qu'ont voulait pas se perdre, car le but est de comprendre le fonctionnement générale et les principale différence entre les base de données relationnels et graphes sur ce type de requêtes.

## Implémentation de notre Base de données sur Neo4J

Avant toute chose on va procéder à l'importation du fichier csv sur notre espace de travail.

### Part 1 : Creation des noeuds

```
1 WITH 'file:///ourData.csv' AS url
2 LOAD CSV WITH HEADERS FROM url AS line
3 MERGE (personne:Personne{id_personne:line.id})
4 ON CREATE SET personne.nom = line.Name, personne.id_fils = line.id_fils,
5 personne.id_parents = line.id_parents;
```

Added 30 labels, created 30 nodes, set 120 properties, completed after 101 ms.

Table  
Code

### Part 2 : Création des relationship entre les noeuds

On va crée la relation parents suivant les id\_parents par rapport aux id\_fils

```
1 MATCH (parent:Personne),(fils:Personne)
2 WHERE parent.id_parents = fils.id_fils
3 CREATE (parent)-[r:PARENT_OF]-(fils)
```

Après l'importation des données, on peut avoir le graphe des descendant des personne réalisé avec nos données : (sous-graphe)



On utilisant la requêtes :

```
1 MATCH (personne:Personne)
2 RETURN personne
3 LIMIT 20
```

On exprime la même requête avec son équivalent en SQL :

Importation des données sur postgresql

Part 1 : Création de la table

```

postgres=# CREATE TABLE "personne"
postgres-# (
postgres(# "id_personne" INTEGER UNIQUE,
postgres(# "nom_personne" TEXT,
postgres(# "id_fils" INTEGER,
postgres(# "id_parents" INTEGER
postgres(# );
CREATE TABLE

```

## Part 2 : Importation des données depuis notre jeu de données our-Data.csv

```

postgres=# COPY personne(id_personne, nom_personne, id_fils, id_parents)
FROM '/home/amar/Images/Neo4J-RSS/Dataset/ourDataset/ourData.csv'
DELIMITER ','
CSV HEADER;
COPY 30
postgres=# SELECT * FROM personne;

```

Notre base de données est comme suit :

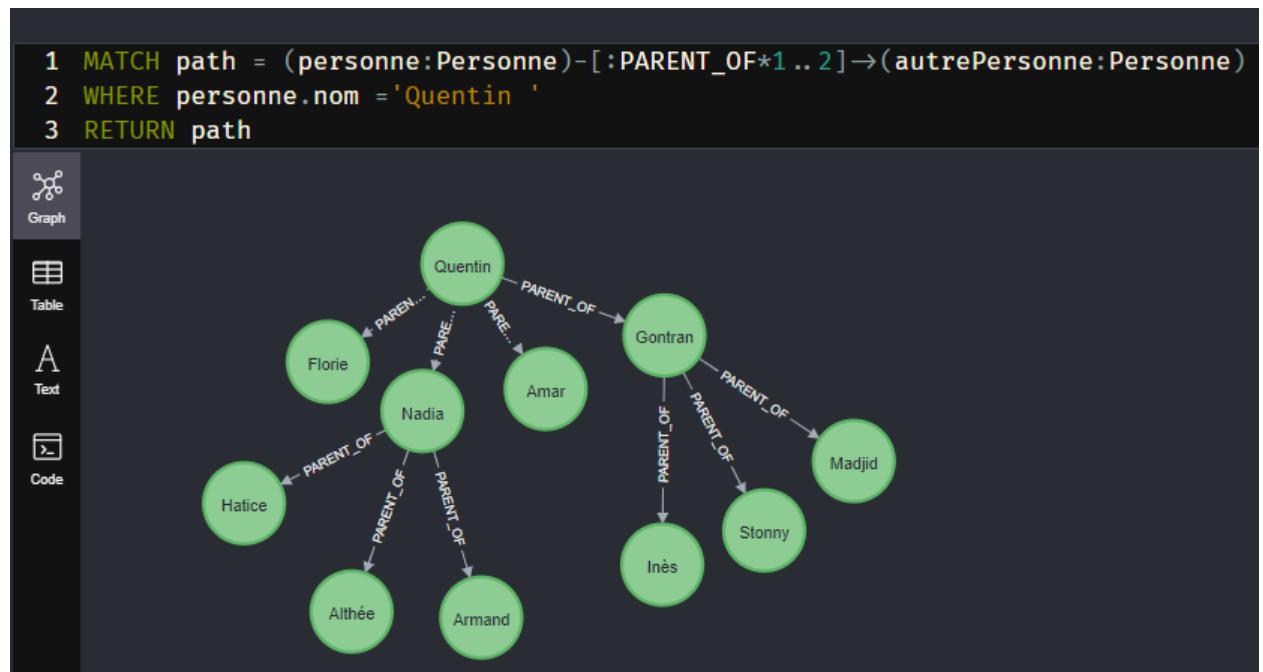
<b>id_personne</b>	<b>nom_personne</b>	<b>id_fils</b>	<b>id_parents</b>
1	Frédéric	1	2
2	Louis	2	0
3	Fabien	3	1
4	Armance	4	1
5	Quentin	5	3
6	Henriette	6	3
7	Simon	7	2
8	Ariane	8	7
9	Mireille	9	8
10	Nadia	10	5
11	Florie	11	5
12	Gontran	12	5
13	Amar	13	5
14	Armand	14	10
15	Althée	15	10
16	Hatrice	16	10
17	Agénor	17	2
18	Gladys	18	2
19	Blandine	19	18
20	Serge	20	18
21	Romuald	21	19
22	Nassima	22	20
23	Madjid	23	12
24	Stonny	24	12
25	Inès	25	12
26	Angeline	26	24
27	Abdellah	27	24
28	Ali	28	26
29	Fatiha	29	6
30	Jean	30	6

(30 rows)



Une seconde requête que l'on peut faire c'est de retourner les descendants d'une personne en particulier : (de la personne dans le prénom est Quentin)

On obtient le sous graphe ci-dessous



On exprime la même requête avec son équivalent en SQL :

```

postgres=# WITH RECURSIVE req(id_personne, id_fils, id_parents, descendants) as
(SELECT person.id_personne, person.id_fils, person.id_parents, person.nom_personne
FROM personne person
WHERE person.nom_personne = 'Quentin'
UNION ALL
SELECT person.id_personne, person.id_fils, person.id_parents, descendants ||' '|| person.nom_personne
FROM personne person
INNER JOIN req as rq ON rq.id_fils = person.id_parents )
SELECT * FROM req;
    
```

On a comme résultats :

<code>id_personne</code>	<code>id_fils</code>	<code>id_parents</code>	<code>descendants</code>
5	5	3	Quentin
10	10	5	Quentin Nadia
11	11	5	Quentin Florie
12	12	5	Quentin Gontran
13	13	5	Quentin Amar
14	14	10	Quentin Nadia Armand
15	15	10	Quentin Nadia Althée
16	16	10	Quentin Nadia Hatice
23	23	12	Quentin GontranMadjid
24	24	12	Quentin GontranStonny
25	25	12	Quentin GontranInès
26	26	24	Quentin GontranStonnyAngeline
27	27	24	Quentin GontranStonnyAbdellah
28	28	26	Quentin GontranStonnyAngeline Ali
(14 rows)			

Une autre requête consiste à retourner le nombre de générations entre une personne et un prédecesseur :

## Au niveau de neo4J

Ici on cherche le nombre de générations entre Armand et son grand parent Louis

```
1 //Requête 3: Affichage du nombre de générations entre 2 personnes (Armand) et (Louis)
2 MATCH path = (personne:Personne)-[:PARENT_OF*1..10]-(predecesseur:Personne)
3 WHERE personne.nom = 'Louis' AND predecesseur.nom = 'Armand'
4 RETURN length(path)
```

length(path)
5

En SQL :

On exprime la même requête avec son équivalent en SQL :

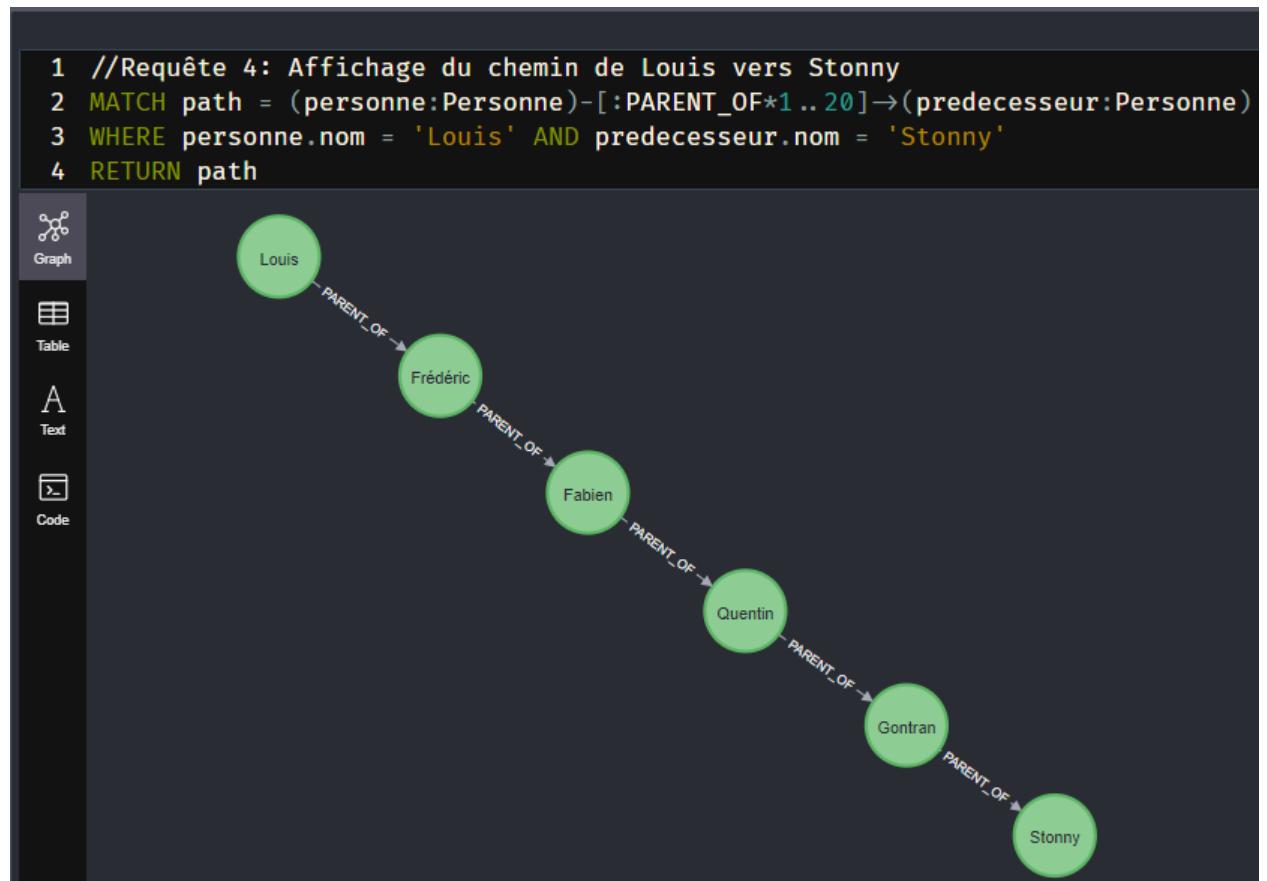
```
postgres=# WITH RECURSIVE req(id_personne, id_fils, id_parents, descendants, nombre_de_generations) as
(SELECT person.id_personne, person.id_fils, person.id_parents, person.nom_personne, 0 as nombre_de_generations
FROM personne person
WHERE person.id_parents = 0
UNION ALL
SELECT person.id_personne, person.id_fils, person.id_parents, descendants ||'''|| person.nom_personne, nombre_de_generations+1
FROM personne person
INNER JOIN req as rq ON rq.id_fils = person.id_parents )
SELECT * FROM req WHERE id_fils=14 and id_parents=10;
```

On a le résultat suivant :

id_personne	id_fils	id_parents	descendants	nombre_de_generations
14	14	10	Louis FrédéricFabien Quentin Nadia Armand	5

Une 10 ème et dernière requête c'est de retourner un chemin de Louis jusqu'à Stonny :

### Au niveau de neo4J



En SQL :

```

postgres=# WITH RECURSIVE req(id_personne, id_fils, id_parents, descendants) as
(SELECT person.id_personne, person.id_fils, person.id_parents, person.nom_personne
FROM personne person
WHERE person.id_parents = 0
UNION ALL
SELECT person.id_personne, person.id_fils, person.id_parents, descendants ||'||| person.nom_personne
FROM personne person
INNER JOIN req as rq ON rq.id_fils = person.id_parents )
SELECT * FROM req WHERE id_fils=24 and id_parents=12;
    
```

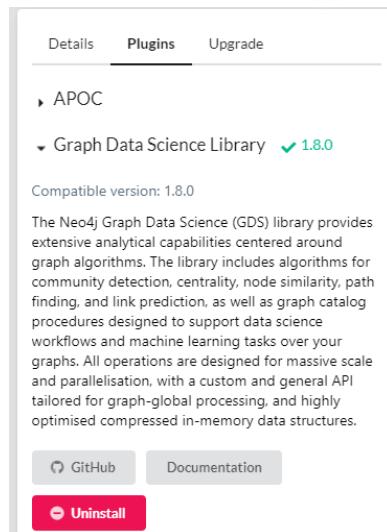
On a le résultat suivant :

id_personne	id_fils	id_parents	descendants
(1 row)	24	24	12   Louis   FrédéricFabien Quentin GontranStonny

## Partie 3 : Analytique de graphe

Dans cette partie, on va faire tourne en autres des algorithmes de Graphe DataScience Library sur nos données.

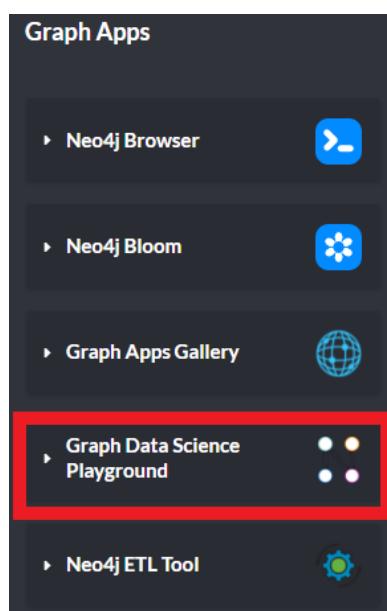
Au début on a Installer le Plugin associé à la **Graphe DataScience Library**, Cela peut se faire via l'interface de Neo4J.



Pour nous faciliter la tâche on a installer **NEuler** qui est une interface utilisateur sans code qui nous aide à intégrer la bibliothèque Neo4j Graph Data Science. Il prend en charge l'exécution de chacun des algorithmes graphiques de la bibliothèque, l'affichage des résultats, et il fournit également des requêtes Cypher pour reproduire ses résultats.

On suit les étapes de l'installation du site de Neo4J :

<https://neo4j.com/developer/graph-data-science/neuler-no-code-graph-algorithms/>



L'interface de la NEuler pour la Graph Data Science Playground se présente ainsi comme suit :

The screenshot shows the NEuler interface. At the top, there's a navigation bar with 'Home' (highlighted in blue), 'Run Single Algorithm', and 'Run Algorithm Recipe'. Below it, a banner says 'WELCOME TO NEULER - THE GRAPH DATA SCIENCE PLAYGROUND'. A 'Database Connection' section shows 'Username: neo4j', 'Server: bolt://localhost:7687', and 'Database: neo4j', with a 'Configure Database' button. The main area has a 'Getting Started' heading. It says 'NEuler supports running graph algorithms in two ways:' and shows two icons: one for 'Run single algorithm' (a graph with arrows) and one for 'Run Algorithm Recipe' (a fork and spoon).

## Algorithm 1 : Closeness

On a choisi de faire tourner l'algorithme **Closeness** qui est efficace pour savoir quels sont les noeuds qui peuvent diffuser au mieux l'information sur le dataset que l'ont a crée car, il est le plus adapté. C'est plus représentatif d'analyser la diffusion d'information sur des personnes que sur notre dataset de match de football et clubs etc

L'algorithme est le suivant :

```
:use neo4j;

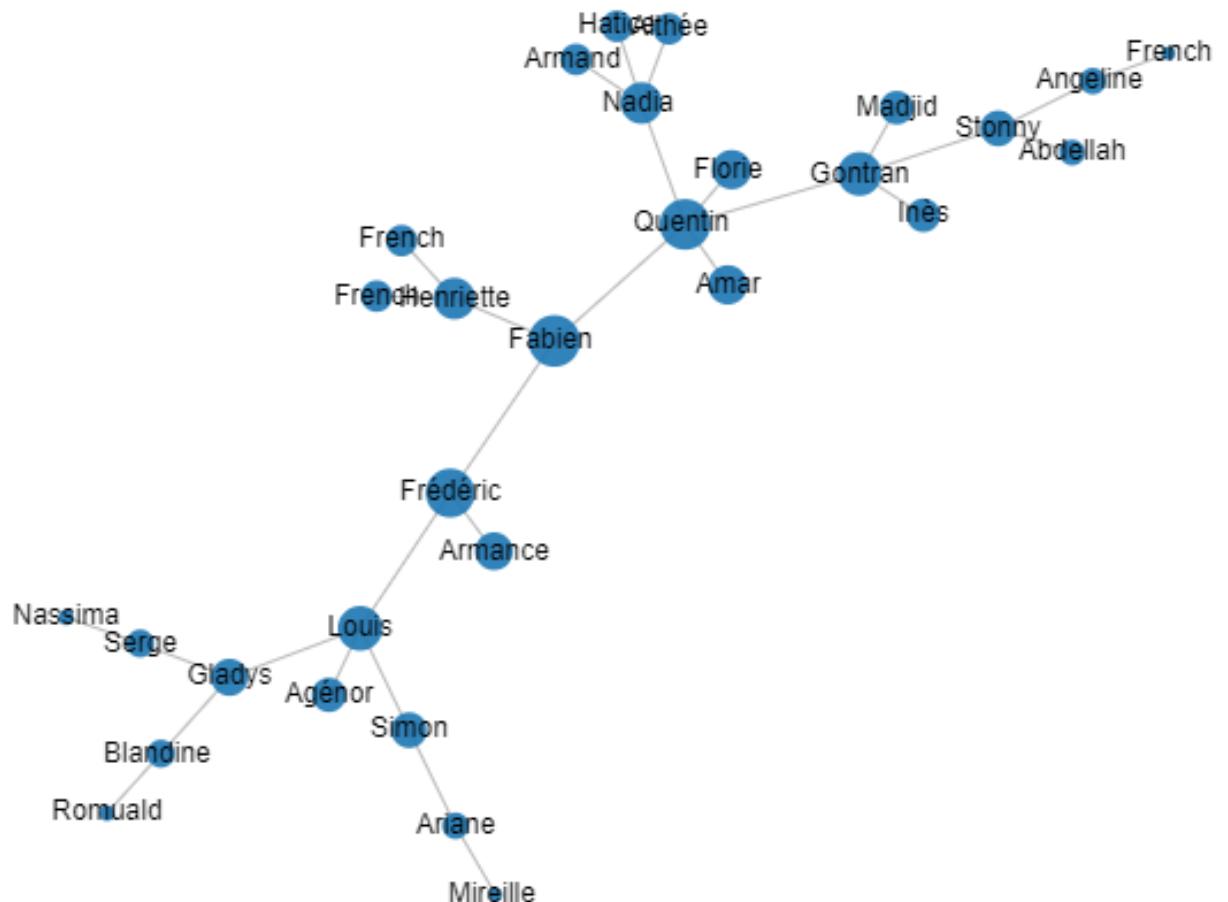
:param limit => ( 45 );
:param config => ({
  nodeProjection: 'Personne',
  relationshipProjection: {
    relType: {
      type: 'PARENT_OF',
      orientation: 'NATURAL',
      properties: {}
    }
  },
  writeProperty: 'closeness'
});
:param communityNodeLimit => ( 10 );

CALL gds.alpha.closeness.write($config);

MATCH (node:`Personne`)
WHERE exists(node.`closeness`)
RETURN node, node.`closeness` AS score
ORDER BY score DESC
LIMIT toInteger($limit);
```

On a les résultats suivant :

Personne	
Node	Score
1, Fabien	0.3372093023255814
3, Quentin	0.32954545454545453
2, Frédéric	0.31521739130434784
0, Louis	0.28431372549019607
5, Gontran	0.27884615384615385
3, Henriette	0.2636363636363636
5, Nadia	0.2636363636363636



On remarque on utilisant cette algorithme que **Fabien** à le score le plus élevé et donc c'est lui qui est le plus apte à diffuser l'information.

## Algorithme 2 : K-Nearest Neighbors

Cette algorithme permet de classifier les données suivant une valeur K (distance) donnée, C'est un algorithme simple et facile à mettre en oeuvre.

On va utiliser KNN sur notre ensemble de données des matchs de football, on va classifier grâce à cette algorithme les matchs suivant les nombres de buts marquées à domicile. On va mettre la valeur du K à 5 car c'est la valeur qui est souvent la plus optimale (mais réellement il faut bien évidemment faire des test des différentes valeurs de différents K et voir la précision pour chaque K et ensuite faire de la "cross validation" afin de choisir le meilleur).

L'algorithme KNN est le suivant :

```
:use neo4j;
```

```
:param limit => ( 42 );

:param config => ({
  nodeProjection: 'Matches',
  relationshipProjection: {
    relType: {
      type: '*',
      orientation: 'NATURAL',
      properties: {}
    }
  },
  nodeWeightProperty: 'homeTeamGoal',
  topK: 5,
  randomJoins: 10,
  sampleRate: 0.5,
  deltaThreshold: 0.001,
  nodeProperties: [
    'homeTeamGoal'
  ]
});
:param communityNodeLimit => ( 10 );
```

```
CALL gds.beta.knn.stream($config)
YIELD node1, node2, similarity
WITH node1, collect({node: gds.util.asNode(node2), similarity: similarity}) AS to
RETURN gds.util.asNode(node1) AS from, to
LIMIT toInteger($limit);
```

Résultats :

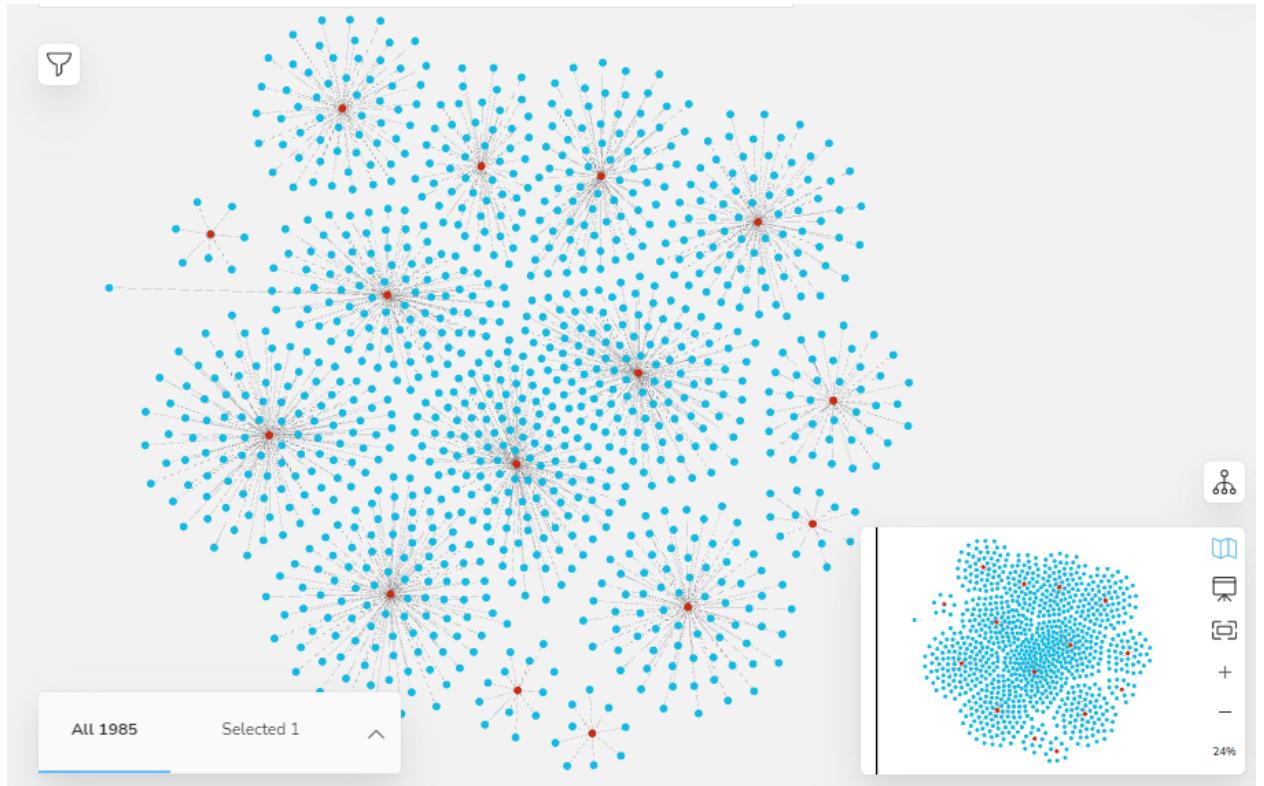
Matches

From	Nodes							
492473, 1	888321, 1	1	1726194, 1	1	1474239, 1	1	1468565, 1	1
492474, 0	2016014, 0	1	2216651, 0	1	506630, 0	1	838734, 0	1
492475, 0	1216874, 0	1	1722205, 0	1	876010, 0	1	704742, 0	1
492476, 5	654353, 5	1	530573, 5	1	836699, 5	1	1051758, 5	1
492477, 1	523831, 1	1	838859, 1	1	1709786, 1	1	1748798, 1	1
492478, 1	1256842, 1	1	1677188, 1	1	530128, 1	1	1030352, 1	1
492479, 2	875906, 2	1	1726114, 2	1	1227889, 2	1	506752, 2	1
492480, 1	530250, 1	1	1222705, 1	1	1474935, 1	1	1778105, 1	1
							2016079, 1	1

Les matchs (Sur graphique c'est les id des matchs) sont classé selon le nombre de buts marqués à domicile avec une distance de 5

## L'interface Bloom

Durant nos différentes parties du projet, on a utilisé bloom afin d'avoir facilement et rapidement une vue global de nos données importées, ce qui nous a grandement aidé à avoir des requêtes en tête à exécuter sous Cypher :



En affichant les liens entre les nœuds team et matchs suivant une relation WON, on peut facilement remarqué que il y'a des équipes qui n'ont pas gagnées plusieurs matchs par exemple ou au contraire voir les équipes qui ont gagnés beaucoup de matchs.

En visualisant nos données avec bloom, nous a permis de mieux comprendre les relations entre nos différents noeuds. Ça nous a même aider nous en tant que binôme afin de bien communiquer sur les différents aspects de nos données.

L'utilisation de bloom nous a également permis de bien inspecté nos éléments de données et voir des traversés de graphes complexes d'une manière simplifié.

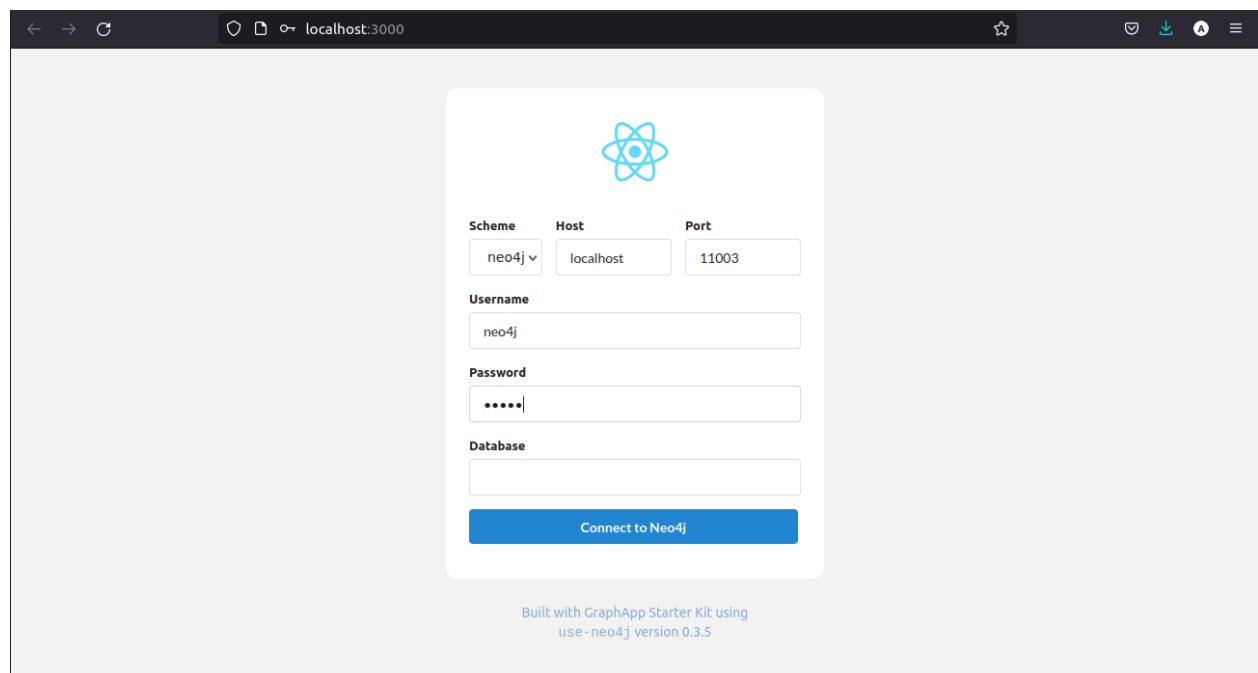
## Partie 4 : Partie libre

Dans cette dernière partie, on a essayé de créer une application web en utilisant la librairie React.js et une base de données neo4j. Pour se connecter au serveur de base de données, on a intégré le packet use-neo4j (<https://github.com/adam-cowley/use-neo4j>) développé par Adam Cowley(<https://twitter.com/adamcowley>). use-neo4j est un packet de npm qui est basé sur des composants et des "Hooks" pour être facilement intégré dans les nouvelles architectures d'applications développées en React.js.

Après, on a pris le app-starter(<https://github.com/adam-cowley/graphapp-starter-react>) développé par Adam Cowley comme point de départ pour le développement de notre application.

Pour atteindre notre objectif, on a développé des composants React(functional components) en utilisant le language **TypeScript**. Aussi, on a utilisé un packet npm "react-awesome-slider" qui nous a facilité l'affichage des statistiques.

Dans la première page, on doit remplir les informations nécessaires pour ce connecter au serveur de base de données. C'est une étape intégrée dans le packet npm mentionné au-dessus.



Après la réussite de la requête de connexion à la base de données, une page s'affiche est une requête Cypher est déjà exécuter pour donner une information sur les données (nombre de matchs).

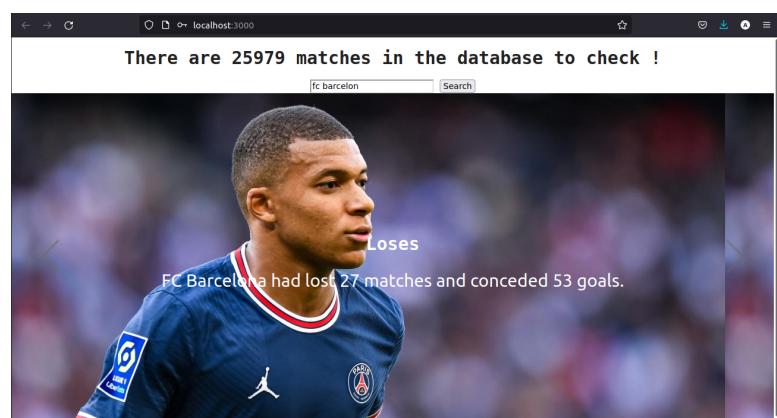


L'utilisateur peut saisir le nom d'une équipe(ou abréviation) pour voir des statistiques générales concernant cette équipe. Il a aussi la possibilité de changer le nom de l'équipe à tout moment et la requête s'execute instantanément :

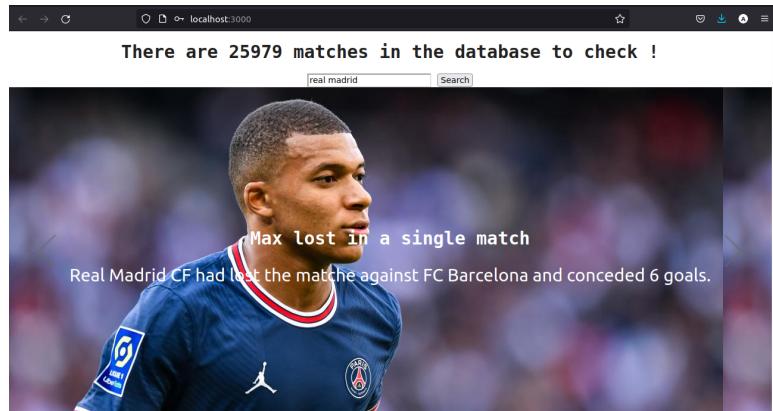
- Le nombre des matchs gagnés avec la somme des buts marqués par l'équipe sélectionnée.



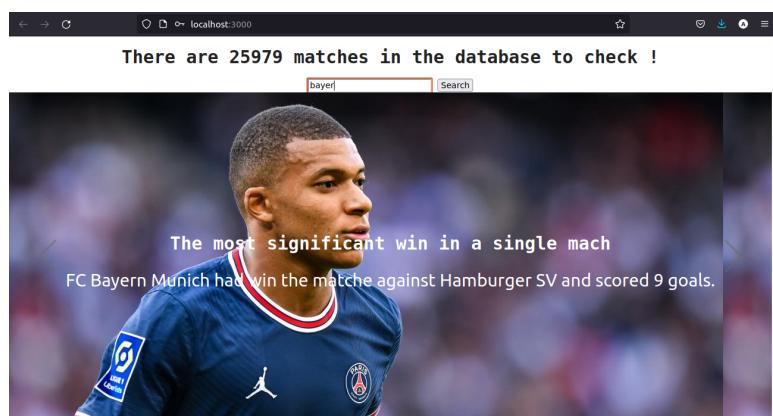
Le nombre des matchs perdus avec la somme des buts encaissé par l'équipe sélectionnée.



Le nombre maximal de buts encaissés par l'équipe sélectionnée dans un seul match.



Le nombre maximal de buts marqués par l'équipe sélectionnée dans un seul match.



Une hiérarchie de nos différents fichiers est disponible sur : <https://gaufre.informatique.univ-paris-diderot.fr/henni/projetcypher>

Pour une description plus approfondi de notre application Web voir :  
<https://gaufre.informatique.univ-paris-diderot.fr/rsmouki/match-statistics.>