



implémentation d'une BD graphe dans Neo4j

Réalisé par :

- Akli Yacine
- NAIT ALAMARA Ouali

Le choix d'un jeu de données



En se basant sur les données de Netflix.

Source : Kaggle

Lien : [Dataset Netflix](#)

- 6235 lignes
- 12 colonnes regroupant les informations sur les films, acteurs

Pré-traitement des données



à partir de la colonne **date_added** (ex : 08-sept-2018), nous avons créé 3 nouvelles colonnes :

- ❖ year (Integer between 2012-2019)
- ❖ month (Integer between 1-12)
- ❖ day (Integer between 1-31)

Outil utiliser : Power Bi

Note : Les dates dans la colonne **date_added** ne sont pas formater à un format unique, donc à partir des 3 nouvelles colonnes on a créé une nouvelle colonne **date_added** avec des dates de meme formats, pour pouvoir extraire correctement (year, month, day) a partir de la colonne **date_added**

Implémentation des données dans une base de données graphe Neo4J

Importer le nouveau fichier Csv dans le dossier dbms/import.

- Création des noeuds Movie :

```
LOAD CSV WITH HEADERS FROM "file:/netflix3.csv" as F FIELDTERMINATOR ','
CREATE (m:Movie{id:toInteger(F.show_id), title:F.title})
SET
m.type = F.type,
m.director = F.director,
m.cast= F.cast,
m.country = F.country,
m.date_str = F.date_added,
m.release_year = F.release_year,
m.rating = F.rating,
m.duration = F.duration,
m.listed_in = F.listed_in,
m.description = F.description,
m.year = apoc.date.field(apoc.date.parse(F.date_added, "ms", "MM/dd/yyyy"), "year"),
m.month = apoc.date.field(apoc.date.parse(F.date_added, "ms", "MM/dd/yyyy"), "month"),
m.day = apoc.date.field(apoc.date.parse(F.date_added, "ms", "MM/dd/yyyy"), "day");
```

Implémentation des données dans une base de données graphe Neo4J



- Création des noeuds Persons et Categories

```
MATCH (m:Movie)
WHERE m.cast IS NOT NULL
WITH m
UNWIND split(m.cast, ',') AS actor
MERGE (p:Person {name: trim(actor)})
MERGE (p)-[r:ACTED_IN]->(m);
```

```
MATCH (m:Movie)
WHERE m.listed_in IS NOT NULL
WITH m
UNWIND split(m.listed_in, ',') AS category
MERGE (c:Category {name: trim(category)})
MERGE (m)-[r:IN_CATEGORY]->(c);
```

Implémentation des données dans une base de données graphe Neo4J



- Création des noeuds Directors et Types

```
MATCH (m:Movie)
WHERE m.director IS NOT NULL
WITH m
MERGE (d:Person {name: m.director})
MERGE (d)-[r:DIRECTED]->(m);
```

```
MATCH (m:Movie)
WHERE m.type IS NOT NULL
WITH m
MERGE (t:Type {type: m.type})
MERGE (m)-[r:TYPED_AS]->(t);
```

Implémentation des données dans une base de données graphe Neo4J



- Création des noeuds Countries

```
MATCH (m:Movie)
WHERE m.country IS NOT NULL
MERGE (c:Country {name:
trim(m.country)})
MERGE (m)-[:WHERE]->(c);
```

Implémentation des données dans une base de données graphe Neo4J



- Suppression des Propriétés non utiles dans Movie (Car on a créé à chaque une de ces propriétés des noeuds)

MATCH(m:Movie)

SET m.country = null, m.category = null, m.type = null , m.director = null , m.cast = null

Implémentation des données dans une base de données graphe Neo4J



- Création de la relation *WORK_WITH* entre les acteurs qui ont joué dans le même film

```
MATCH (p:Person)-[rel:ACTED_IN]->(m:Movie)<-[:ACTED_IN]-(p2:Person)  
MERGE (p)-[r:WORK_WITH]-(p2);
```

Implémentation des données dans une base de données graphe Neo4J



- Création des noeuds Years entre 2012 et 2019

```
WITH range(2012, 2019) AS years, range(1,12) AS months  
FOREACH(year IN years | CREATE (y:Year {value: year}));
```

Implémentation des données dans une base de données graphe Neo4J



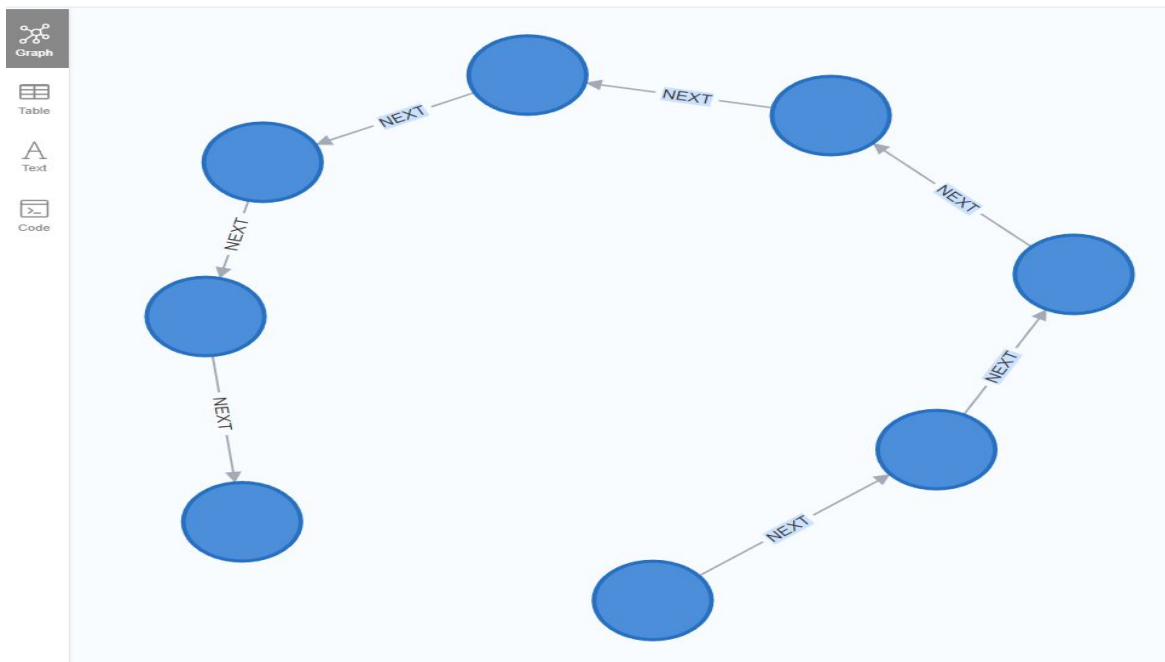
- Connecter chaque année à l'année suivante avec la relation NEXT

```
MATCH (year:Year)
WITH year
ORDER BY year.value
WITH collect(year) AS years
FOREACH(i in RANGE(0, size(years)-2) |
  FOREACH(year1 in [years[i]] |
    FOREACH(year2 in [years[i+1]] |
      MERGE (year1)-[:NEXT]->(year2))));
```

Implémentation des données dans une base de données graphe Neo4J

- Donc on aura ce graph pour les nœuds Years

```
neo4j$ MATCH (y:Year) RETURN y
```



Implémentation des données dans une base de données graphe Neo4J



- Création de la relation CREATED_ON entre les noeuds Movie et Year, et creation d'un index pour le noeuds Year pour accélérer cette operation (Création de la relation CREATED_ON)

```
CREATE INDEX yearIndex  
FOR (m:Year)  
ON m.value
```

```
MATCH (m:Movie)  
WITH m, m.year AS y  
MATCH (year:Year {value: y})  
MERGE (m)-[:CREATED_ON]->(year)  
RETURN m,year;
```

Le graphe Neo4j finale de nos données Netflix:

```
neo4j$ call db.schema.visualization
```

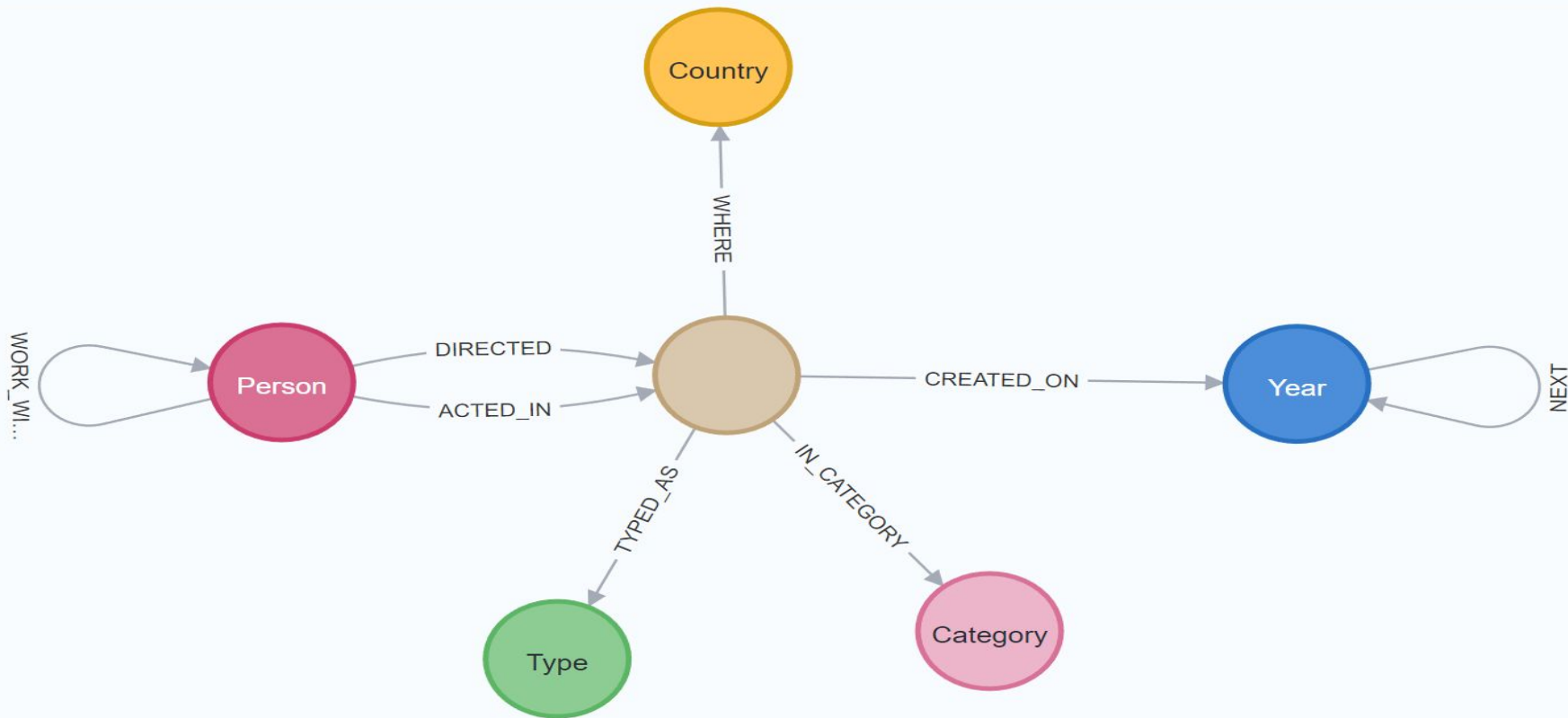


Graph

Table

Text

Code



Implémentation des données dans une base de données relationnel PostgreSQL

- Concernant la modélisation SQL, on a choisi de créer seulement deux tables afin d'éviter plus de jointures et éviter la complexité des requêtes :

Les deux table sont :

```
CREATE TABLE movie (  
  show_id BIGINT,  
  type_movie VARCHAR(20),  
  title TEXT,  
  country TEXT,  
  date_added DATE,  
  release_year VARCHAR(4),  
  rating VARCHAR(50),  
  duration TEXT,  
  listed_in TEXT,  
  description TEXT,  
  PRIMARY KEY (show_id)  
);
```

```
CREATE TABLE persons (  
  id SERIAL PRIMARY KEY,  
  name TEXT,  
  is_director BOOLEAN,  
  show_id BIGINT  
);
```

Implémentation des données dans une base de données relationnel PostgreSQL

- Création de la table movie à partir de fichier netflix3.csv : (On exécute tous ces requêtes à la fois)

```
create temporary table t (  
  show_id BIGINT,  
  type_movie VARCHAR(20),  
  title TEXT, director TEXT,  
  cast_member TEXT,  
  country TEXT,  
  date_added DATE,  
  release_year VARCHAR(4),  
  rating VARCHAR(50),  
  duration TEXT, listed_in TEXT,  
  description TEXT,  
  year_added VARCHAR(4),  
  month_added VARCHAR(2),  
  day_added VARCHAR(2),  
  PRIMARY KEY (show_id)  
);
```

```
copy t (show_id, type_movie,  
  title, director, cast_member,  
  country, date_added,  
  release_year, rating,  
  duration, listed_in,  
  description, year_added,  
  month_added, day_added)  
from  
  'C:\Users\yacin\Desktop\Neo4jProject\netflix3.csv'  
DELIMITER ','  
CSV HEADER;
```

```
insert into movie( show_id, type_movie,  
  title, country, date_added, release_year,  
  rating, duration, listed_in, description  
)  
select  
  t.show_id, t.type_movie,  
  t.title, t.country ,  
  t.date_added , t.release_year,  
  t.rating, t.duration,  
  t.listed_in, t.description  
from t  
  
drop table t;
```


Implémentation des données dans une base de données relationnel PostgreSQL

- Création de la table persons à partir de fichier netflix3.csv : (On exécute tous ces requêtes à la fois)

```
create temporary table t (  
  show_id BIGINT,  
  type_movie VARCHAR(20),  
  title TEXT, director TEXT,  
  cast_member TEXT,  
  country TEXT,  
  date_added DATE,  
  release_year VARCHAR(4),  
  rating VARCHAR(50),  
  duration TEXT, listed_in TEXT,  
  description TEXT,  
  year_added VARCHAR(4),  
  month_added VARCHAR(2),  
  day_added VARCHAR(2),  
  PRIMARY KEY (show_id)
```

```
);
```

```
copy t (show_id,  
  type_movie, title,  
  director, cast_member,  
  country, date_added,  
  release_year, rating,  
  duration, listed_in,  
  description, year_added,  
  month_added, day_added)  
from  
'C:\Users\yacin\Desktop\Neo4jProject\netflix3.csv'  
DELIMITER ','  
CSV HEADER;
```

```
WITH actor_cte AS (  
  SELECT trim(n1.x) as actor, n1.y as show_id  
  FROM (SELECT  
    unnest(string_to_array(cast_member, ',')) x,  
    t.show_id as y  
    FROM t) n1  
)  
  
insert into persons(name, is_director, show_id)  
select actor_cte.actor, false, actor_cte.show_id  
from actor_cte;  
  
drop table t;
```

Implémentation des données dans une base de données relationnel PostgreSQL

- Insérer les réalisateurs des films dans table persons à partir de fichier netflix3.csv : (On exécute tous ces requêtes à la fois)

```
create temporary table t (  
  show_id BIGINT,  
  type_movie VARCHAR(20),  
  title TEXT, director TEXT,  
  cast_member TEXT,  
  country TEXT,  
  date_added DATE,  
  release_year VARCHAR(4),  
  rating VARCHAR(50),  
  duration TEXT, listed_in TEXT,  
  description TEXT,  
  year_added VARCHAR(4),  
  month_added VARCHAR(2),  
  day_added VARCHAR(2),  
  PRIMARY KEY (show_id)
```

```
copy t (show_id, type_movie,  
  title, director, cast_member,  
  country, date_added,  
  release_year, rating,  
  duration, listed_in,  
  description, year_added,  
  month_added, day_added)  
from  
'C:\Users\yacin\Desktop\Neo4jPro  
ject\netflix3.csv'  
DELIMITER ','  
CSV HEADER;
```

```
insert into persons(  
  name,  
  is_director,  
  show_id  
)  
select  
  t.director,  
  true,  
  t.show_id  
from t  
ON CONFLICT (id) DO NOTHING;  
  
drop table t;
```

```
);
```

Requêtes (Neo4j)

Une requête avec un filtre négatif sur certains types de patterns

```
MATCH (m:Movie)
WHERE NOT (m)-[:TYPED_AS]-> (:Type{type: 'TV Show'})
AND NOT (m)-[:WHERE]-> (:Country{name: "United States"})
AND NOT toInteger(split(m.duration, ' ')[0]) > 90
RETURN m.title, m.duration;
```

Dans cette requête on cherche tous les films et non pas les Séries et qui ne sont pas des films des Etats Unis et qui ont une durée au plus 90 min, donc on aura 740 films qui ont ce critère

Started streaming 740 records after 1 ms and completed after 38 ms.

Requêtes (PostgreSQL équivalente)

Celle-ci est la requête PostgreSQL équivalente et on obtient 740 films ce qui prouve que les données de Neo4j et PostgreSQL sont bien synchronisés

```
SELECT title, duration
FROM movie m
WHERE type_movie != 'TV Show'
AND (country != 'United States' OR country IS NULL )
AND CAST (SPLIT_PART(m.duration, ' ', 1) AS INTEGER) <= 90;
```

Voici le résultat de cette requête

Total rows: 740 of 740

Query complete 00:00:00.114

Requêtes (Neo4j)



Une requête avec OPTIONAL MATCH, dans cette requête on aura les films ou les séries qui ont été créés en 2015 avec leurs nombres d'acteurs, avec OPTIONAL MATCH il va afficher même les films qui n'ont pas d'acteurs (peut-être c'est des documentaires)

```
MATCH (m:Movie)
OPTIONAL MATCH (p:Person) -[:ACTED_IN]-> (m)
WITH m, p
MATCH (m) -[:CREATED_ON]-> (y:Year)
WHERE y.value = 2015
RETURN DISTINCT m.title, count(p)
ORDER BY count(p) DESC
```

Résultats:

Started streaming 90 records in less than 1 ms and completed after 66 ms.

Requêtes (PostgreSQL équivalente)

Celle-ci est la requête PostgreSQL équivalente et on obtient les mêmes résultats avec Neo4j

```
SELECT DISTINCT m.title, COUNT(p)
FROM movie m
LEFT JOIN persons p ON p.show_id = m.show_id
WHERE EXTRACT(YEAR FROM m.date_added) = 2015
GROUP BY m.title
ORDER BY COUNT(p) DESC;
```

Voici le résultat de cette requête

Total rows: 90 of 90

Query complete 00:00:00.059

Requêtes

Requête avec 'OPTIONAL MATCH'

```
MATCH (u:Person)
OPTIONAL MATCH (u)-[:DIRECTED]→(m:Movie)
RETURN u.name, m.title
```

	u.name	m.title
	Rebecca Madsen	nan
167	"Kyle Rideout"	"Adventures in Public School"
168	"Riley Murdock"	null
169	"Vincent Tong"	null

Cette requête va d'abord trouver tous les nœuds de type **Person**. Ensuite, elle utilise l'opérateur **OPTIONAL MATCH** pour trouver les nœuds de type **Movie** auxquels chaque nœud Person a été lié par une relation **DIRECTED**. Si aucun nœud Movie n'est trouvé pour un nœud Person donné, la valeur de la colonne m.title sera mise à null pour cette ligne de résultat.

Enfin, la requête utilise l'opérateur RETURN pour renvoyer le nom de chaque Person et le titre du film qu'il a dirigé (s'il a été directeur de ce film). Le résultat de cette requête sera une table avec deux colonnes : la première colonne contiendra le nom de chaque Personnes, et la seconde colonne contiendra le titre du film qu'il a dirigé (ou null s'il n'a pas été directeur de film). Chaque ligne de la table représentera une personne et le film qu'il a été directeur

Requêtes (Neo4j)



deux utilisations de **WITH** différentes, dans cette requête on cherche tous les réalisateurs avec le nombre de séries ou films qui ont réalisé et ce nombre doit être supérieur à 5

```
MATCH (m:Movie) <-[:DIRECTED]- (d:Person)
WITH d, COUNT(m) as n_movies
WHERE n_movies > 5
RETURN d.name, n_movies;
```

Résultats:

Started streaming 27 records after 6 ms and completed after 27 ms.

Requêtes (PostgreSQL équivalente)



Celle-ci est la requête PostgreSQL équivalente et on obtient les mêmes résultats avec Neo4j

```
SELECT d.name, COUNT(m) as n_movies
FROM persons d
JOIN movie m ON m.show_id = d.show_id
WHERE d.is_director = TRUE AND d.name IS NOT NULL
GROUP BY d.name
HAVING COUNT(m) > 5;
```

Voici le résultat de cette requête

Total rows: 27 of 27	Query complete 00:00:00.050
----------------------	-----------------------------

Requêtes (Neo4j)

Deuxième utilisations de WITH, dans cette requête on cherche la catégorie qui a plus de films ou séries puis on affiche tous les acteurs de cette catégorie

```
MATCH (m:Movie) -[:IN_CATEGORY]-> (c:Category)
WITH c, count(m) as mv
order by mv asc
limit 1
MATCH (c) -- (:Movie) -[:ACTED_IN]- (p:Person)
return p.name;
```

Résultats:

Started streaming 40 records after 9 ms and completed after 16 ms.

Requêtes (Neo4j)



Une requête explorant à la fois les données et la topologie du graphe

```
MATCH p=(:Person) -[:WORK_WITH*2..3]-> (:Person{name:'Leonardo  
DiCaprio'})  
WHERE any(n in nodes(p) WHERE n.name = 'Hugh Jackman')  
RETURN p
```

Ici on cherche les path avec la relation WORK_WITH de longueur 2 ou 3 et qui pointe vers le noeud **Leonardo DiCaprio** et ce path doit contenir au moins un seul noeud **Hugh Jackman**

Requêtes (Neo4j)

une requête utilisant COLLECT et UNWIND, dans cette requête on cherche à avoir tous les films de l'acteur qui a le plus de films

```
MATCH (p:Person) -[:ACTED_IN]-> (m:Movie)
WITH p, COLLECT(m.title) AS movie_titles
ORDER BY size(movie_titles) DESC
LIMIT 1
UNWIND movie_titles as mv
RETURN mv ORDER BY mv ASC
```

mv	
1	"A Family Man"
2	"A Wednesday"
3	"Aiyaary"
4	"C Kkompany"
5	"Chaahat"
6	"Chashme Buddoor"
7	

Started streaming 33 records in less than 1 ms and completed after 27 ms.

Requêtes (PostgreSQL équivalente)

Celle-ci est la requête PostgreSQL équivalente et on obtient les mêmes résultats avec Neo4j

```
SELECT unnest(n.movies) FROM  
(SELECT p.name, array_agg(m.title) AS movies  
FROM persons p  
JOIN movie m ON m.show_id = p.show_id  
WHERE p.name IS NOT NULL AND p.is_director = FALSE  
GROUP BY p.name  
ORDER BY array_length(array_agg(m.title), 1) DESC  
LIMIT 1) AS n  
ORDER BY unnest(n.movies) ASC;
```

	unnest text	
1	A Family Man	
2	A Wednesday	
3	Aiyaary	
4	C Kkompany	
5	Chaaahat	
6	Chashme Buddoor	
7	Game	
8	Hamara Dil Aapke Paas Hai	

Total rows: 33 of 33

Query complete 00:00:00.194

Requêtes (Neo4j)

une requête manipulant des listes avec **reduce**, dans cette requête on cherche à avoir les 5 premier acteurs qui ont joué le plus d'heure dans les films qui ont été créé entre 2015 et 2017 et des films qui ne sont pas Indian

```
MATCH (p:Person) -[:ACTED_IN]-> (m:Movie)
-[:TYPED_AS]-> (t:Type{type: 'Movie'}),
(m) -[:CREATED_ON]-> (y:Year),
(m) -[:WHERE]-> (c:Country)
WHERE y.value >= 2015 AND y.value <=2017
AND c.name <> 'India'
WITH p, collect(m.duration) AS duration_list
WITH p, reduce(total_duration=0,
  i IN duration_list | total_duration+toInteger(split(i, '
')[0])) AS total_duration
RETURN p.name, round(toFloat(total_duration)/60, 2)
ORDER BY total_duration DESC
LIMIT 5
```

	p.name	round(toFloa
1	"Demet Akbağ"	15.38
2	"Samuel West"	11.48
3	"Junko Takeuchi"	11.05
4	"Chie Nakamura"	11.05
5	"Kazuhiko Inoue"	9.63

Started streaming 5 records after 1 ms and completed after 34 ms.

Requêtes (PostgreSQL équivalente)

Celle-ci est la requête PostgreSQL équivalente et on obtient les mêmes résultats avec Neo4j

```
WITH my_table AS (  
  SELECT p.name, m.duration  
  FROM persons p  
  JOIN movie m ON m.show_id = p.show_id  
  WHERE m.type_movie = 'Movie'  
        AND EXTRACT(YEAR FROM m.date_added) >= 2015  
        AND EXTRACT(YEAR FROM m.date_added) <= 2017  
        AND m.country != 'India'  
        AND p.is_director = FALSE  
)  
SELECT name, ROUND(SUM(CAST(SPLIT_PART(duration, '-', 1)  
  AS INTEGER) / 60.0)::numeric, 2) AS total_duration  
FROM my_table  
WHERE name IS NOT NULL  
GROUP BY name  
ORDER BY total_duration DESC  
LIMIT 5;
```

	name text	total_duration numeric
1	Demet Akbağ	15.38
2	Samuel West	11.48
3	Chie Nakamura	11.05
4	Junko Takeuchi	11.05
5	Kazuhiko Inoue	9.63

Requêtes (Neo4j)

Un filtre post UNION avec CALL, ici on cherche à avoir l'acteur qui a joué le plus de Séries avec le réalisateur qui a fait le plus de Séries

```
CALL {  
  MATCH (p:Person) -[:ACTED_IN]-> (m:Movie) -[:TYPED_AS]-> (t:Type{type: 'TV Show'})  
  WITH p, collect(m.duration) AS seasons  
  RETURN p.name as actor, "Actor" AS act, size(seasons) AS seasons_number  
  ORDER BY seasons_number DESC LIMIT 1  
  UNION  
  MATCH (d:Person) -[:DIRECTED]-> (m:Movie) -[:TYPED_AS]-> (t:Type{type: 'TV Show'})  
  WITH d, collect(m.duration) AS seasons  
  RETURN d.name as actor, "Director" AS act, size(seasons) AS seasons_number  
  ORDER BY seasons_number DESC LIMIT 1  
}  
RETURN actor, seasons_number, act;
```

actor	seasons_number	act
"Takahiro Sakurai"	18	"Actor"
"Alastair Fothergill"	3	"Director"

Requêtes (PostgreSQL équivalente)

Celle-ci est la requête PostgreSQL équivalente et on obtient les mêmes résultats avec Neo4j

```
WITH t1 AS (  
  SELECT p.name, count(*) AS seasons_number, 'Actor' AS act  
  FROM persons p  
  JOIN movie m ON m.show_id = p.show_id  
  WHERE m.type_movie = 'TV Show' AND p.name IS NOT NULL  
  AND p.is_director = FALSE  
  GROUP BY p.name  
  ORDER BY seasons_number DESC  
  LIMIT 1  
, t2 AS (  
  SELECT d.name, count(*) AS seasons_number, 'Director' AS act  
  FROM persons d  
  JOIN movie m ON m.show_id = d.show_id  
  WHERE m.type_movie = 'TV Show' AND d.name IS NOT NULL  
  AND d.is_director = TRUE  
  GROUP BY d.name  
  ORDER BY seasons_number DESC  
  LIMIT 1  
)  
SELECT * FROM t1 UNION SELECT * FROM t2;
```

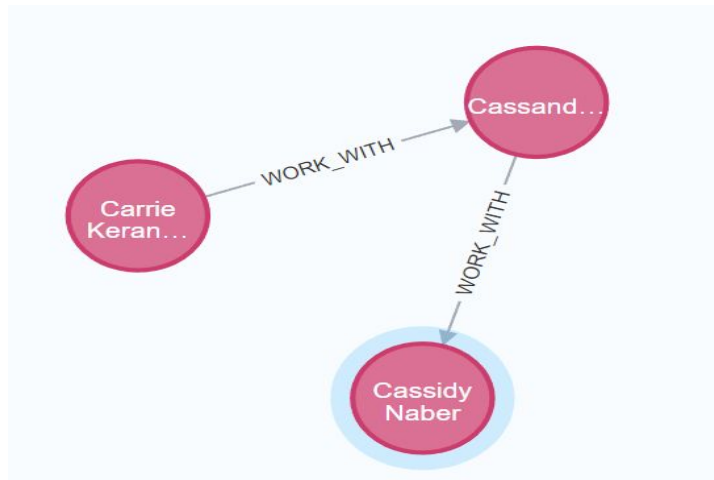
	name text	seasons_number bigint	act text
1	Alastair Fothergill	3	Director
2	Takahiro Sakurai	18	Actor

Requêtes (Neo4j)

Une requête utilisant des fonctions de prédicat du type `all()`, dans cette requête on cherche tous les path de la relation `WORK_WITH` de longueur entre 1 et 3 et dont tous les noeuds contient un nom qui commence avec 'CA'

```
MATCH path=(p1:Person) -[:WORK_WITH*1..3]-> (p2:Person)
WHERE ALL(n in nodes(path) WHERE n.name STARTS WITH "Ca" )
RETURN path
```

On aura des paths comme ca

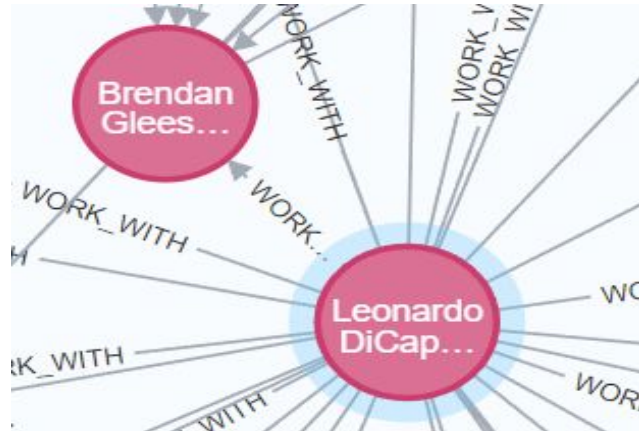


Requêtes (Neo4j)

Une requête utilisant des fonctions de prédicat du type `none()`, dans cette requête on cherche tous les path de la relation `WORK_WITH` de longueur 1 et qui commence avec le noeud de nom 'Leonardo DiCaprio' et dont tous les noeuds ne contient pas un nom qui commence avec 'A'

```
MATCH path=(p1:Person{name: 'Leonardo DiCaprio'}) -[:WORK_WITH*1]-> (p2:Person)
WHERE NONE(n in nodes(path) WHERE n.name STARTS WITH "A")
RETURN path
```

On aura des paths comme ça



Requêtes (Neo4j)



Une requête utilisant des fonctions de prédicat du type exists(), dans cette requête on cherche à avoir tous les réalisateurs qui sont aussi acteurs

```
MATCH path=(p:Person) -[:DIRECTED]-> (m:Movie)
WHERE exists((p)-[:ACTED_IN]->(:Movie))
RETURN DISTINCT p.name;
```

Requêtes (Neo4j)



Une requête utilisant des fonctions de prédicat du type any(), dans cette requête on cherche à avoir les films ou séries non Indienne et qui contient le nom 'horror' dans leurs description

```
MATCH path= (m:Movie) -[:WHERE]->(c:Country)
WHERE NOT(c.name CONTAINS "India")
AND any(n in nodes(path) WHERE n.description CONTAINS
"horror")
RETURN DISTINCT m.title, c.name, m.description;
```

Requêtes (Neo4j)



Une requête utilisant des fonctions de prédicat du type single(), dans cette requête on cherche à avoir tous les acteurs qui ont joué dans une seule catégorie de films ou séries

```
MATCH path=(p:Person)-[:ACTED_IN]->(:Movie)-[:IN_CATEGORY]->(c:Category)
WHERE single(r in relationships(path) WHERE type(r)="IN_CATEGORY" )
RETURN DISTINCT p.name, c.name;
```

dans cette deuxième requête on veut avoir les acteurs qui ont joué dans un seul pays de films ou séries, et qui ont jamais joué dans un autre pays

```
MATCH path=(p:Person)-[:ACTED_IN]->(:Movie)-[:WHERE]->(c:Country)
WHERE single(r in relationships(path) WHERE type(r)="WHERE" )
RETURN DISTINCT p.name, c.name;
```

Contraintes



```
CREATE CONSTRAINT movie_id  
FOR (m :Movie)  
REQUIRE m.id IS UNIQUE
```

```
CREATE CONSTRAINT person_name  
FOR (p :Person)  
REQUIRE p.name IS UNIQUE
```

```
CREATE CONSTRAINT category_value  
FOR (c:Category)  
REQUIRE c.name IS UNIQUE
```

```
CREATE CONSTRAINT country_name  
FOR (c :Country)  
REQUIRE c.name IS UNIQUE
```

```
CREATE CONSTRAINT year_value  
FOR (y:Year)  
REQUIRE y.value IS UNIQUE
```

```
CREATE CONSTRAINT type_value  
FOR (t:Type)  
REQUIRE t.type IS UNIQUE
```

Indexes



```
CREATE FULLTEXT INDEX m_description_index  
FOR (m:Movie)  
ON EACH [m.description]
```

```
CREATE INDEX m_title_index  
FOR (m:Movie)  
ON m.title
```

```
CREATE INDEX p_name_index  
FOR (p:Person)  
ON p.name
```


Requêtes (Neo4j)

Des requêtes Cypher a priori équivalentes mais utilisant des syntaxes différentes : ces requête pour afficher les acteurs qui ont jouer seulement dans les Etats Unis

```
MATCH (p) -[:ACTED_IN]-> (:Movie) -[:WHERE]->
(c2:Country) WHERE not(c2.name CONTAINS
'United States')
WITH p
MATCH
path2=(p)-[:ACTED_IN]->(:Movie)-[:WHERE]->(c3:Co
untry) WHERE any(n in nodes(path2) WHERE
n.name ='United States' ) RETURN p.name,
collect(c3.name)
```

```
MATCH (p) -[:ACTED_IN]-> (:Movie)
-[:WHERE]-> (c2:Country)
WHERE not(c2.name CONTAINS 'United
States')
WITH p
MATCH
path2=(p)-[:ACTED_IN]->(:Movie)-[:WHERE]->(c
3:Country) WHERE c3.name ='United States'
RETURN p.name, collect(c3.name)
```

on remarque que cette requête il est un peu plus rapide que l'autre

Requêtes (Neo4j)



Des requêtes Cypher a priori équivalentes mais utilisant des syntaxes différentes : en comparant cette requête avec la requête du slide 19

```
MATCH (m:Movie)
WHERE NOT exists((m)-[:TYPED_AS]-> (:Type{type: 'TV Show'}))
AND NOT exists((m)-[:WHERE]-> (:Country{name: "United States"}))
AND NOT toInteger(split(m.duration, '')[0]) > 90
RETURN m.title, m.duration;
```

on remarque que cette requête il est un peu plus rapide que l'autre en terme d'exécution

Requêtes (Neo4j)



Des requêtes Cypher a priori équivalentes mais utilisant des syntaxes différentes : en comparant cette requête avec la requête du slide 28

```
MATCH (p:Person) -[:ACTED_IN]-> (m:Movie)
WITH p, count(m.title) AS movie_number
ORDER BY movie_number DESC
LIMIT 1
MATCH (p) -[:ACTED_IN]-> (m2:Movie)
RETURN m2.title ORDER BY m2.title ASC
```

on remarque que cette requête il est un peu plus rapide que l'autre en terme d'exécution

Requêtes (Neo4j)

Des requêtes Cypher a priori équivalentes mais utilisant des syntaxes différentes : Ici on voit une différence remarquable entre ces requêtes équivalentes

```
MATCH p=(:Person{name:'Leonardo DiCaprio'}) -[:ACTED_IN]- (:Movie) -[:ACTED_IN*3..7]- (p2:Person)
RETURN distinct p2.name
```

Started streaming 14928 records in less than 1 ms and completed after 1175 ms, displaying first 1000 rows.

```
MATCH p=(:Person{name:'Leonardo DiCaprio'}) -[:WORK_WITH*2..4]- (p2:Person)
RETURN distinct p2.name
```

Started streaming 14928 records after 1 ms and completed after 21 ms, displaying first 1000 rows.

Requêtes (Neo4j)



Dans les deux requêtes précédentes on voit que y a une différence remarquable en termes d'exécution, la première requête se termine après 1175 ms et la deuxième requête se termine après 20 ms, cela dû au nombre de relations utilisées dans le path de chaque requête pour avoir le même résultat.

Dans la première requête, on a utilisé des path avec la relation `ACTED_IN` de longueur entre 3 et 7 et la deuxième requête on a utilisé des path avec la relation `WORK_WITH` de longueur 2 et 4 ce qui veut dire moins de relations avec la première requête.

Conclusion :

Utilisation beaucoup de relations dans notre requête Cypher est coûteux, c'est pourquoi d'ailleurs on a créé la relation `WORK_WITH` pour accélérer les requêtes en utilisant moins de requête

Requêtes (Neo4j et PostgreSQL)

Requête PostgreSQL récursive et son équivalence en Cypher:



```
WITH RECURSIVE t (name, show_id, depth) AS (  
  SELECT p.name, p.show_id, 1  
  FROM persons p  
  WHERE p.name = 'Leonardo DiCaprio'  
  UNION ALL  
  SELECT p2.name, p2.show_id, t.depth + 1  
  FROM t  
  JOIN persons p1 ON p1.show_id = t.show_id  
  JOIN persons p2 ON p2.name = p1.name  
  WHERE p1.is_director = FALSE  
  AND p2.is_director = FALSE AND t.depth < 3  
)
```

```
SELECT DISTINCT name  
FROM t  
WHERE name <> 'Leonardo DiCaprio';
```

	name	
	text	🔒
1	Seandrea Sledge	
2	Chris McGarry	
3	Quvenzhané Wallis	
4	Ashley Park	
5	Harvey Keitel	
6	Ricky Jay	
7	Finbar Lynch	
8	John Rhys-Davies	

Total rows: 1000 of 1503

Query complete 00:00:00.091

Requêtes équivalente en Neo4j



```
MATCH path=(p1:Person{name:'Leonardo  
DiCaprio'}) -[:WORK_WITH*1..2]- (p2:Person)  
RETURN distinct p2.name
```

	p2.name
1	"Nathalie Baye"
2	"Kevin Corrigan"
3	"Mary Kate Schellhardt"
4	"Christopher Walken"
5	"Max Casella"
6	"Tom Hanks"
7	

Started streaming 1503 records after 1 ms and completed after 10 ms, displaying first 1000 rows.

PostgreSQL récursive VS Neo4j

Pour les petites longueurs de la relation `WORK_WITH`, on ne voit pas vraiment de différence entre les deux requêtes PostgreSQL et Neo4j, mais dès que la longueur de la relation est ≥ 4 on remarque que PostgreSQL est vraiment lent.

Démonstration pour la longueur de `WORK_WITH` est entre 1 et 4 en Cypher qui est équivalent à `t.depth < 5` en PostgreSQL :

Total rows: 1000 of 14927 Query complete 00:00:22.767

Résultat de PostgreSQL

Started streaming 14928 records after 1 ms and completed after 12 ms, displaying first 1000 rows.

Résultat de
Neo4j

On voit dans ces résultats que PostgreSQL a pris 22 secondes pour exécuter la requête pendant que Neo4j a pris seulement 12 ms pour l'exécuter

PostgreSQL VS Neo4j



Dans cette requête on voit que PostgreSQL s'exécute mieux que Neo4j parce que la requête Neo4j utilise beaucoup de relation ce qui est coûteux

```
SELECT DISTINCT p.name, m.title, m.country,  
EXTRACT(YEAR FROM m.date_added) as year,  
m.type_movie  
FROM persons p, movie m  
WHERE m.show_id = p.show_id  
AND p.name is not null  
AND m.title is not null  
AND m.country is not null  
AND EXTRACT(YEAR FROM m.date_added) is  
not null  
AND m.type_movie IS NOT NULL  
AND p.is_director = FALSE;
```

```
MATCH (p:Person) -[:ACTED_IN]-> (m:Movie)  
-[:CREATED_ON]-> (y:Year)  
WITH p, m, y  
MATCH (m) -[:WHERE]-> (c:Country)  
WITH m, p, y, c  
MATCH (m) -[:TYPED_AS]-> (t:Type)  
RETURN DISTINCT p.name, m.title, y.value,  
c.name, t.type
```

PostgreSQL VS Neo4j



Aussi lors de la création du noeud Person avec les relation "ACTED_IN" et "DIRECTED" en Neo4j a partir du fichier CSV, cela prend beaucoup de temps, par contre la création de la table "persons" en PostgreSQL est presque instantanée, un autre point pour PostgreSQL, ce qui montre que beaucoup de relation rendre Neo4j lent.

Requêtes

Requête avec le prédicat de type all ()

```
MATCH (c:Category) ←[:IN_CATEGORY] -(m:Movie)-[:CREATED_ON]→(a)
WHERE a.value ≤ 2017 AND ALL(x IN c.name WHERE x = 'Docuseries')
WITH COUNT(m) AS movies,c
RETURN c.name, movies
```

	c.name	movies
1	"Docuseries"	92

Dans cet exemple, la fonction **ALL** vérifie que tous les genres de chaque film sont égaux à "Docuseries". Si c'est le cas, le film est retourné dans les résultats de la requête.

On a utilisé **ALL** dans une clause **WITH** pour effectuer des calculs sur les résultats de la requête. Dans notre exemple, on a voulu obtenir **le nombre total de films** qui ont été produits avant l'année **2017** et qui appartiennent au genre "Docuseries"

Requêtes

Le Top 10 Acteurs ayant le plus grand nombre de films joués (Neo4j)

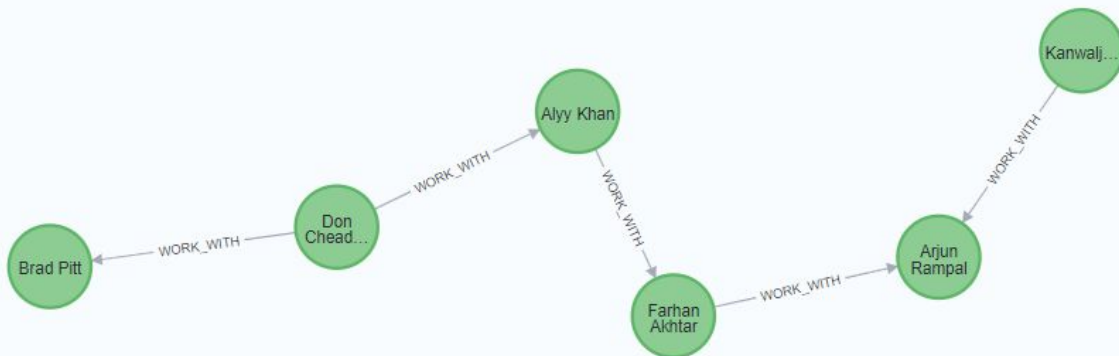
```
1 MATCH (p:Person)-[rel:ACTED_IN]→(m:Movie)
2 WITH p,collect(m.title) as movies,count(*) as total
3 RETURN p.name,total
4 ORDER BY total DESC
5 LIMIT 10
```

	p.name	total
1	"Anupam Kher"	33
2	"Shah Rukh Khan"	30
3	"Naseeruddin Shah"	27
4	"Om Puri"	27
5	"Akshay Kumar"	26
6	"Yuki Kaji"	26
7	"Paresh Rawal"	25
8	"Takahiro Sakurai"	25
9	"Amitabh Bachchan"	24
10	"Boman Irani"	23

Requêtes

La distance entre 2 acteurs (ex: entre Kanwaljeet Singh et Brad Pitt)

```
MATCH (a:Person { name:'Kanwaljeet Singh' }), (b:Person { name:'Brad Pitt' }), p = shortestPath((a)-[r:WORK_WITH*]-(b))
WHERE length(p)> 1
RETURN p
```



Overview

Node labels

* (6)

Person (6)

Relationship types

* (5)

WORK_WITH (5)

Displaying 6 nodes, 5 relationships.

Comparison exécution avec index et sans index



On exécute cette requête avec index : **p_name_index**

```
PROFILE MATCH p=(:Person{name:'Leonardo DiCaprio'}) -[:ACTED_IN]-  
(:Movie) -[:ACTED_IN*3..7]- (p2:Person)  
WHERE any(n in nodes(p) WHERE n.name = 'Hugh Jackman')  
RETURN distinct p2.name
```

Cypher version: , planner: COST, runtime: PIPELINED. 36731996 total db hits in 7672 ms.



▼ NodeIndexSeek@neo4j
anon_0
RANGE INDEX anon_0:Person(name)
WHERE name = \$autostring_0
120 memory (bytes)
38,363,814 pagecache hits
0 pagecache misses
1 estimated rows
2 db hits

1 row

▼ Expand(All)@neo4j
anon_0, anon_1, anon_2
(anon_0)-[anon_1:ACTED_IN]-(anon_2)
1 estimated rows
10 db hits

7 rows

▼ Filter@neo4j
anon_0, anon_1, anon_2
anon_2:Movie
1 estimated rows
44 db hits

On voit que notre requête a utiliser l'index pour s'exécuter et a met 7672 ms pour répondre

Comparison exécution avec index et sans index

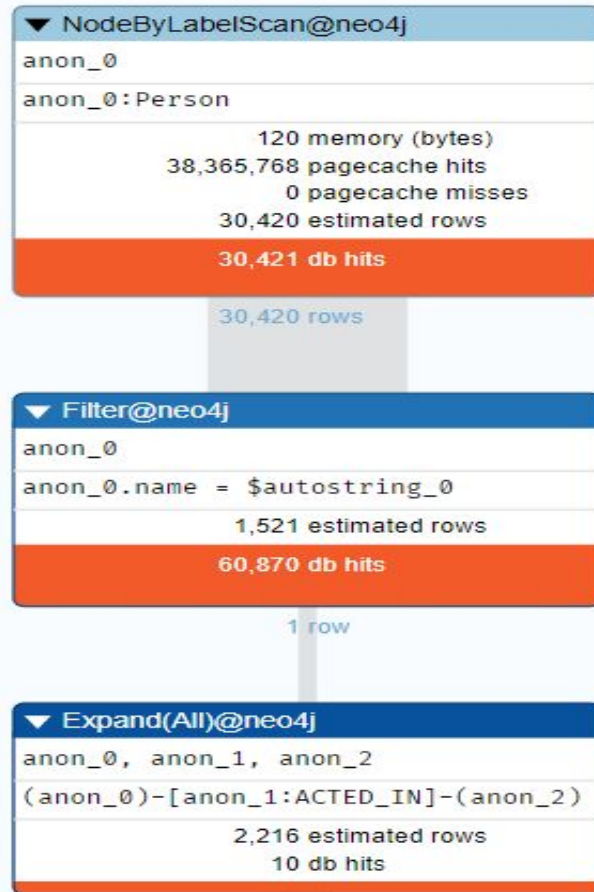
On exécute maintenant la requête sans l'index : `p_name_index`



On aura ça:

Cypher version: , planner: COST, runtime: PIPELINED. 36823285 total db hits in 8401 ms. ⬆ ⬇

en exécutant la requête sans index on remarque que le temps d'exécution est lent sans index, donc l'index accélère l'exécution de notre requête

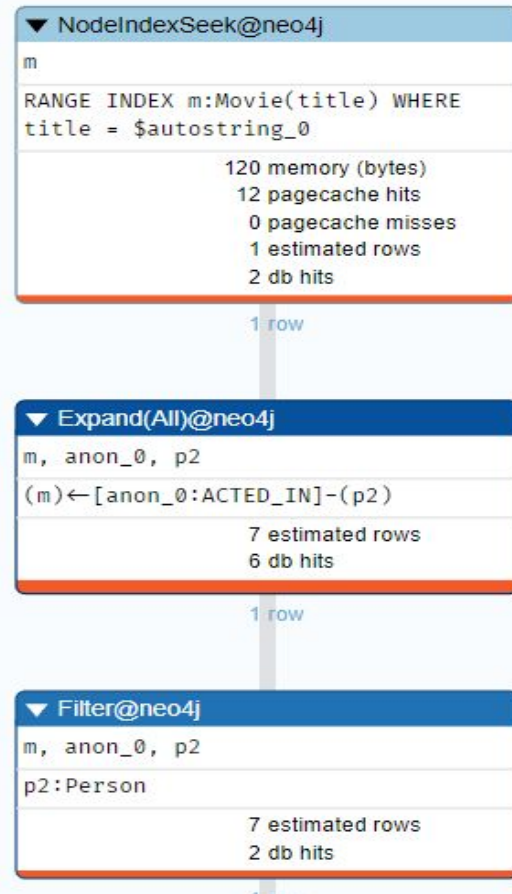


Comparison exécution avec index et sans index

On exécute une autre requête avec index : `m_title_index`

```
PROFILE MATCH p= (m:Movie{title: 'Tidying Up with Marie Kondo'}) <-[:ACTED_IN]- (p2:Person) -[:WORK_WITH*2..5]-> (:Person{name:'Leonardo DiCaprio'})  
RETURN p2.name
```

Cypher version: , planner: COST, runtime: PIPELINED. 14 total db hits in 1 ms.

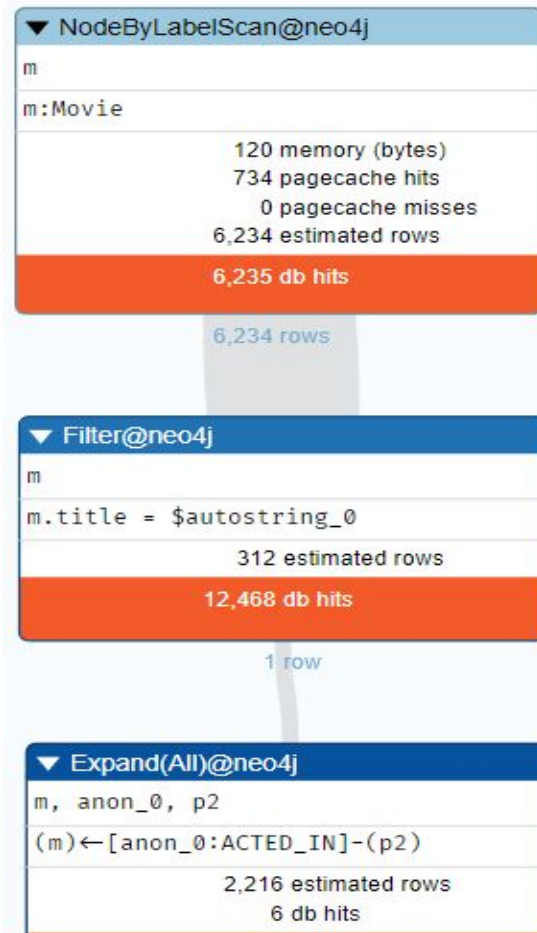


Comparison exécution avec index et sans index

Maintenant on exécute la requête sans index : `m_title_index`, et on aura :

Cypher version: , planner: COST, runtime: PIPELINED. 4413062 total db hits in 516 ms. ⬆ ⬇

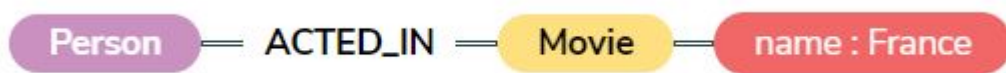
Ici on voit clairement la différence, l'exécution de la requête avec index a met 1 ms pour répondre, sans index a met 516 ms.
Donc l'utilisation de la requête à accélérer l'exécution d la requête



Bloom

On peut analyser nos données avec Bloom visuellement sans utiliser des requêtes cyphers

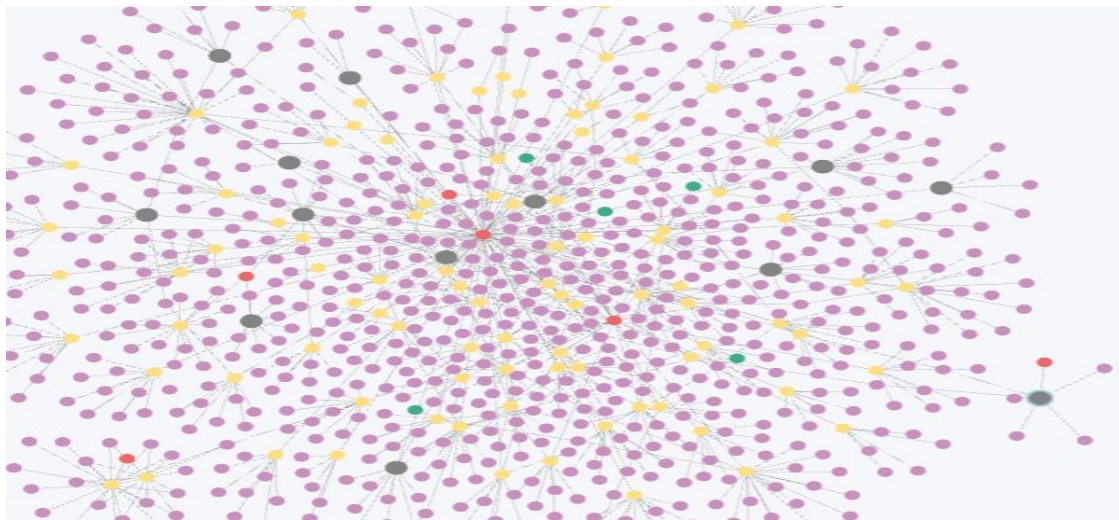
Par exemple on peut avoir le graphe de tous les personnes qui ont joué dans des films Français avec ce path en Bloom



Et on va avoir un graph de tous les acteurs avec les films français.

Remarque :

Ici on a met des rules, par exemple afficher les acteurs qui contient dans leur nom 'Camille' en vert, et afficher les films qui contient dans leur description 'Friend' en gris



Algorithmes de la Graph Data Science Library.

Filtered Node Similarity Algorithms



Cette algorithmes nous permet de savoir les catégories de films et la similarité entre eux, D'abord on crée une projection graph pour les noeuds 'Category', 'Movie', avec la relation **IN_CATEGORY** à partir de notre graph initiale avec cette commande

```
CALL gds.graph.project(  
  'myGraph3',  
  ['Category', 'Movie'],  
  {  
    IN_CATEGORY: {  
      orientation: 'UNDIRECTED'  
    }  
  }  
)
```

Algorithmes de la Graph Data Science Library.

Filtered Node Similarity Algorithms

Puis on exécute l'algorithme sur ce la projection graph créé en Mode "Stream" (pour afficher seulement les résultats de cette algorithmes)

```
CALL gds.alpha.nodeSimilarity.filtered.stream('myGraph3',  
{sourceNodeFilter:'Category' , targetNodeFilter:'Category' } )  
YIELD node1, node2, similarity  
RETURN gds.util.asNode(node1).name AS Category1,  
gds.util.asNode(node2).name AS Category2, similarity  
ORDER BY similarity DESCENDING, Category1, Category2
```

Dans les résultats de cet algorithme on voit les catégories avec le score de similarité entre eux, plus que ce score est grand plus que les catégories sont similaires.

Category1	Category2	similarity
"Dramas"	"International Movies"	0.401500197394394
"International Movies"	"Dramas"	0.401500197394394

Algorithmes de la Graph Data Science Library.

Filtered Node Similarity Algorithms

Exécuter cette algorithme en Mode **stats**, cela peut être utile pour évaluer les performances de l'algorithme en inspectant l'élément **computeMillis**

```
CALL gds.alpha.nodeSimilarity.filtered.stats('myGraph3', {sourceNodeFilter:'Category' ,  
targetNodeFilter:'Category' } )  
YIELD nodesCompared, similarityPairs, computeMillis;
```

nodesCompared	similarityPairs	computeMillis
42	382	10

Algorithmes de la Graph Data Science Library.

Filtered Node Similarity Algorithms

Exécuter cette algorithme en Mode **Mutate**, ce mode nous permet de mettre à jour le graphique nommé qu'on a créé ('myGraph3') avec une nouvelle relation 'SIMILAR' contenant le score de similarité pour cette relation. (It will update the in-memory graph 'myGraph3')

```
CALL gds.alpha.nodeSimilarity.filtered.mutate('myGraph3',{
  mutateRelationshipType: 'SIMILAR',
  mutateProperty: 'score',
  sourceNodeFilter:'Category',
  targetNodeFilter:'Category'
})
YIELD nodesCompared, relationshipsWritten, computeMillis
```

nodesCompared	relationshipsWritten	computeMillis
42	382	12

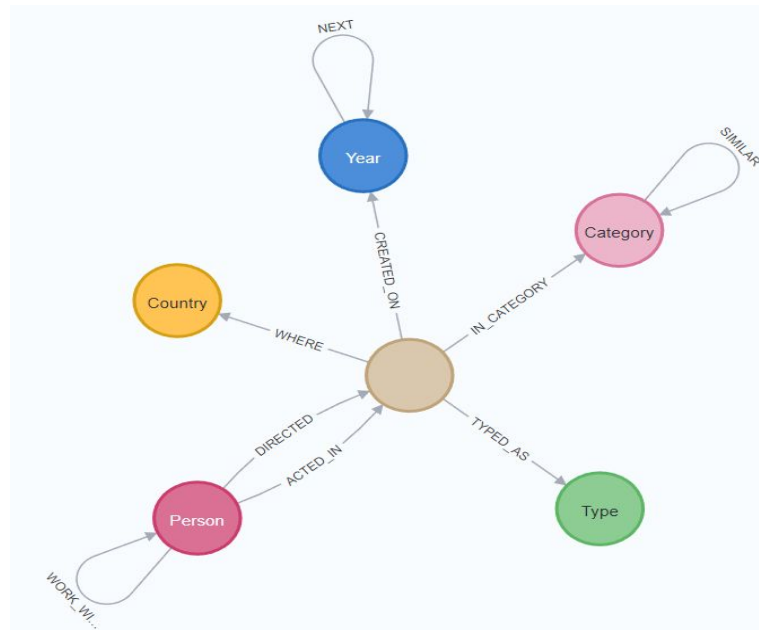
Algorithmes de la Graph Data Science Library.

Filtered Node Similarity Algorithms

Exécuter cet algorithme en Mode **Write**, ce mode nous permet de créer une relation '**SIMILAR**' avec la propriété score de la base de données Neo4j (not '**myGraph3**').

```
CALL gds.alpha.nodeSimilarity.filtered.write('myGraph3',{  
  writeRelationshipType: 'SIMILAR',  
  writeProperty: 'score',  
  sourceNodeFilter:'Category',  
  targetNodeFilter:'Category'  
})  
YIELD nodesCompared, relationshipsWritten
```

On voit qu'une relation SIMILAR est étai créé entre les catégories



Algorithmes de la Graph Data Science Library.

Degree Centrality Algorithms

Cet algorithme va calculer pour chaque personne le nombre de personnes avec qui ont travaillé, puis dans notre requête on s'est intéressé à avoir les 10 premières personnes qui ont travaillé avec plus d'autres personnes . (plus le score es grands plus la personne à travailler avec plus de personne)

```
CALL gds.graph.project(  
  'myGraph4',  
  'Person',  
  {  
    WORK_WITH: {  
      orientation: 'UNDIRECTED'  
    }  
  }  
)
```

```
CALL gds.degree.stream('myGraph4')  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name,  
score AS NumberOfPersonsThatWorkedWith  
ORDER BY NumberOfPersonsThatWorkedWith  
DESC, name DESC  
LIMIT 10
```

Algorithmes de la Graph Data Science Library.

Common Neighbors Algorithms

L'idée de cet algorithme est de calculer le score entre 2 personnes en se basant sur les personnes en commun avec qui ces 2 personnes ont travaillé.

Une valeur de 0 indique que deux nœuds ne sont pas proches, tandis que des valeurs plus élevées indiquent que les nœuds sont plus proches

```
MATCH (p1:Person {name: 'Leonardo DiCaprio'})  
MATCH (p2:Person {name: 'Matt Damon'})  
RETURN gds.alpha.linkprediction.commonNeighbors(p1,  
p2, {relationshipQuery: 'WORK_WITH', direction:  
'BOTH'}) AS score
```

score
8.0

Algorithmes de la Graph Data Science Library.

Common Neighbors Algorithms

Cette requête fait la même chose, mais calculer le score de Common neighbors entre tous les personnes de notre base de données, c'est pour cela cette requête est trop lente

```
MATCH (p1:Person)
MATCH (p2:Person )
RETURN p1.name, p2.name, gds.alpha.linkprediction.commonNeighbors(p1, p2,
{relationshipQuery: 'WORK_WITH', direction: 'BOTH'}) AS score
ORDER BY score DESC
LIMIT 5
```

Algorithmes de la Graph Data Science Library.

Total Neighbors Algorithms



Cet algorithme nous donne le total de personne qui ont travaillé entre 2 personnes données

```
MATCH (p1:Person {name: 'Leonardo DiCaprio'})  
MATCH (p2:Person {name: 'Matt Damon'})  
RETURN gds.alpha.linkprediction.totalNeighbors(p1, p2, {relationshipQuery: 'WORK_WITH',  
direction: 'BOTH'}) AS score
```

score
124.0

Algorithmes de la Graph Data Science Library.

Total Neighbors Algorithms



Cet algorithme nous indique avec un score entre deux personnes, si ces deux personnes sont proches et à quelle degré sont proches.

Une valeur de 0 indique que deux nœuds ne sont pas proches, tandis que des valeurs plus élevées indiquent que les nœuds sont plus proches.

```
MATCH (p1:Person {name: 'Leonardo DiCaprio'})
```

```
MATCH (p2:Person {name: 'Matt Damon'})
```

```
RETURN gds.alpha.linkprediction.adamicAdar(p1, p2, {relationshipQuery: 'WORK_WITH',  
direction: 'BOTH'}) AS score
```

score
2.182653827162529

Algorithmes de la Graph Data Science Library.

Triangle Count Algorithms

Cet algorithme nous donne pour chaque nœud le nombre de relations triangulaire dont ce nœud participe.

```
CALL gds.graph.project(  
  'myGraph5',  
  'Person',  
  {  
    WORK_WITH: {  
      orientation: 'UNDIRECTED'  
    }  
  }  
)
```

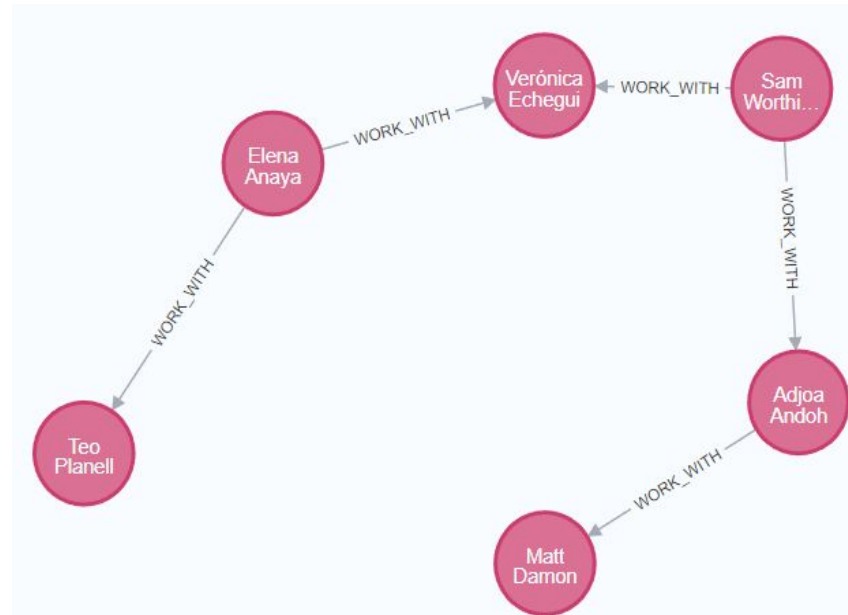
```
CALL gds.triangleCount.stream('myGraph5')  
YIELD nodeId, triangleCount  
RETURN gds.util.asNode(nodeId).name AS  
name, triangleCount  
ORDER BY triangleCount DESC
```

name	triangleCount
"Takahiro Sakurai"	3196
"Yuki Kaji"	2979
"Yuichi Nakamura"	2938

Cypher shortest path

Cette requête pour calculer le plus court chemin entre deux personnes

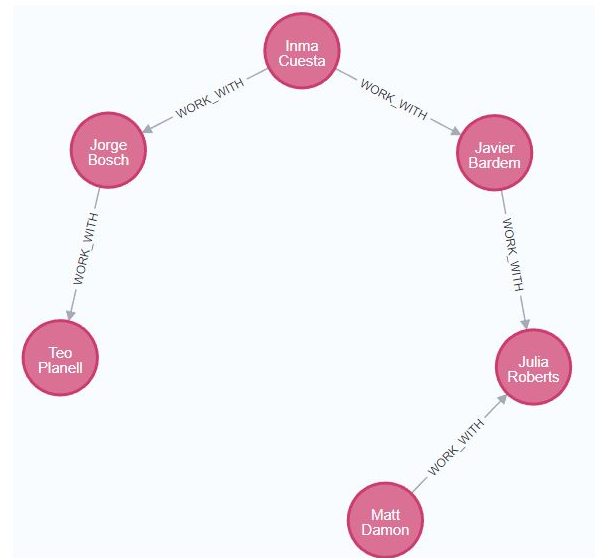
```
MATCH (source:Person{name: 'Matt Damon'  
}), (target:Person{name: 'Teo Planell'}),  
p = shortestPath((source) -[:WORK_WITH*]-  
(target))  
RETURN p
```



Cypher shortest path (Dijkstra pondéré version procedure GDS)

Cette requête pour calculer le plus court chemin entre deux personnes en utilisant Dijkstra pondéré version procedure GDS

```
MATCH (source:Person{name: 'Matt Damon'}), (target:Person{name:
'Teo Planell'})
CALL gds.shortestPath.dijkstra.stream('myGraph5', {
  sourceNode: source,
  targetNode: target
})
YIELD index, sourceNode, targetNode, totalCost, nodeIds, costs, path
RETURN
  index,
  gds.util.asNode(sourceNode).name AS sourceNodeName,
  gds.util.asNode(targetNode).name AS targetNodeName,
  totalCost,
  [nodeId IN nodeIds | gds.util.asNode(nodeId).name] AS
nodeNames,
  costs,
  nodes(path) as path
ORDER BY index
```




PostgreSQL unweighted shortest path



D'abord on a créé une table 'knows' qui contient tous les personnes qui travaille entre eux

```
CREATE TABLE knows AS  
SELECT p1.name as name1, p2.name as name2  
FROM persons p1  
JOIN persons p2 ON p1.show_id = p2.show_id
```



```

WITH RECURSIVE paths(startPerson, endPerson, path, level, endPersonReached) AS (
    SELECT name1 AS startPerson, name2 AS endPerson,
    ARRAY[name1, name2] AS path, 1 AS level,
    bool_or(CASE WHEN p2.name = 'Teo Planell'
    THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
    FOLLOWING) AS
    endPersonReached
    FROM knows
    JOIN persons p1 ON p1.name = knows.name1
    JOIN persons p2 ON p2.name = knows.name2
    WHERE p1.name = 'Matt Damon'
    UNION ALL
    SELECT paths.startPerson AS startPerson, name2 AS endPerson,
    array_append(path, name2) AS path, level + 1 AS level,
    bool_or(CASE WHEN p2.name = 'Teo Planell'
    THEN true ELSE false END) OVER (ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED
    FOLLOWING) AS
    endPersonReached
    FROM paths
    JOIN knows ON paths.endPerson = knows.name1
    JOIN persons p2 ON p2.name = knows.name2
    WHERE p2.name != ALL(paths.path)
    AND NOT paths.endPersonReached
)
SELECT path, level
FROM paths
JOIN persons ON persons.name = paths.endPerson
WHERE persons.name = 'Teo Planell';

```



Merci de votre attention.