Duc Nguyen DUONG M2 MIDS
Inès DARDOURI

Projet : Base de données avancé

I. Choix et import d'un jeu de données

Nous avons décidé de travailler sur les données du TP3 (les fichiers airports.csv, airlines.csv et routes.csv) car nous pensons que ces données sont assez complètes et intéressantes pour travailler. Les trois fichiers CSV qui vont constituer notre base de données sont les suivants:

- Airports.csv : contenant les informations sur les aéroports, y compris les aérodromes et les aéroports fermés
- Airlines.csv : contenant les informations sur les compagnies aériennes qui assurent les vols entres les différents aéroports
- Routes.csv : contenant les informations sur les routes qui relient les aéroports et les compagnies aériennes

A. Cypher

Nous importons nos fichiers CSV directement dans neo4j, grâce aux requêtes d'imports au début du fichier "requests_cypher" joint à ce rapport.
Nous créons donc les nœuds correspondant aux aéroports, aux compagnies aériennes ainsi qu'aux routes.
Nous avons aussi créé les index sur les id de airport, airline et route ainsi que les propriété country, city, IATA de airport et name de route. Ces index vont nous être utiles pour la suite, pour certaines requêtes.
Finalement, nous avons créé une relationship entre les nœuds de airport que nous avons appelé path et qui a pour propriété le nom de airline associé. Cette dernière nous servira pour faire des requêtes plus complexes.

B. Postgresql

Pour faciliter la comparaison entre Neo4j et Postgresql, nous avons décidé d'importer et de modéliser de la même façon pour les deux bases. Les commandes SQL à effectuer sont dans le fichier "requests_postgresql". Nous avons donc créé trois tables : airport, airline et route correspondant aux trois fichiers mentionnés ci-dessus. Ensuite, nous avons mis les index sur les mêmes variables que ceux du Cypher afin de comparer l'efficacité de ces deux types de base de données.

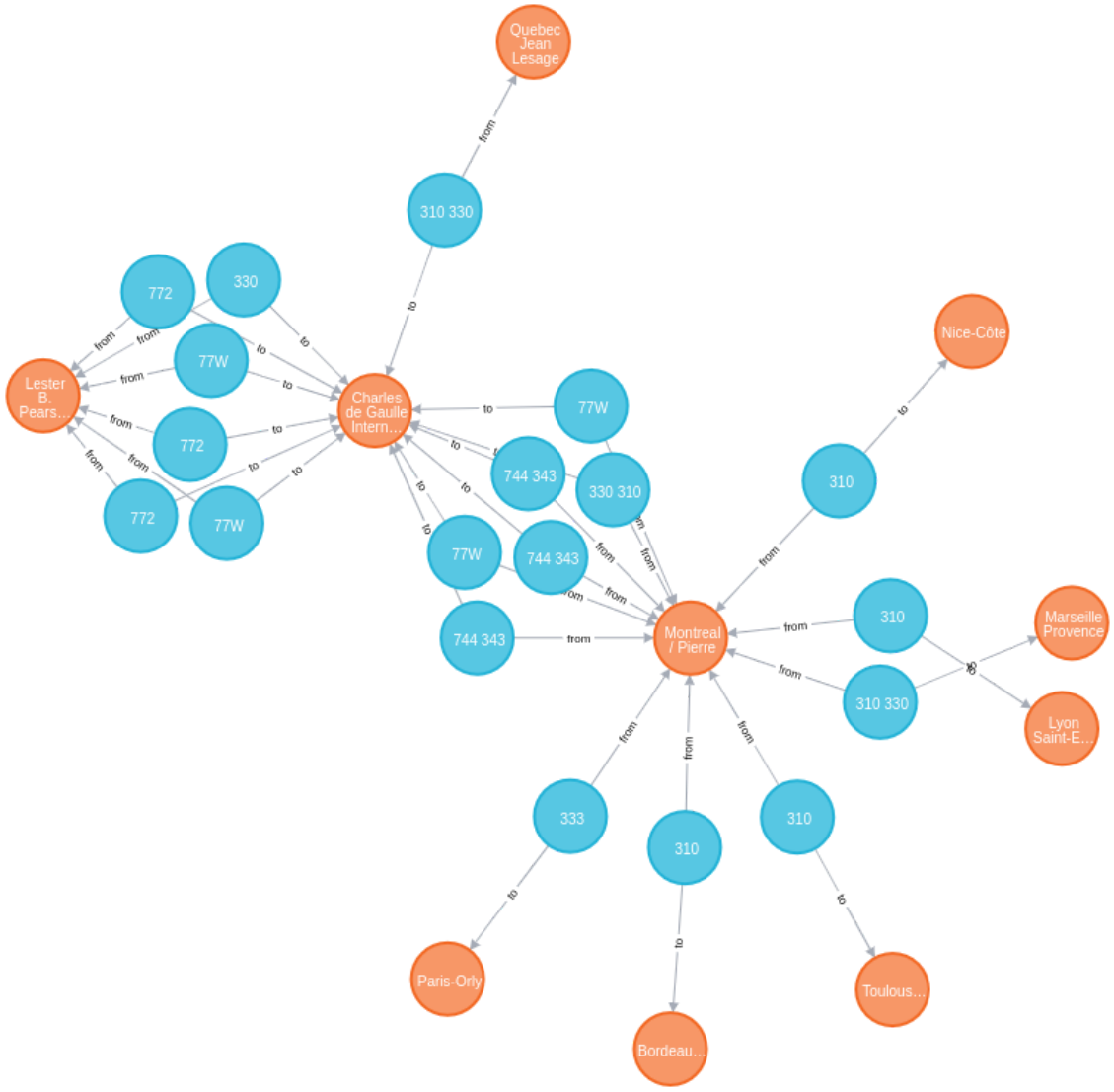
II. Requêtes

Nous avons joint un fichier contenant nos requêtes sur la base de données, avec trois niveaux de difficultés différents, tel que nous avons procédé en TP.

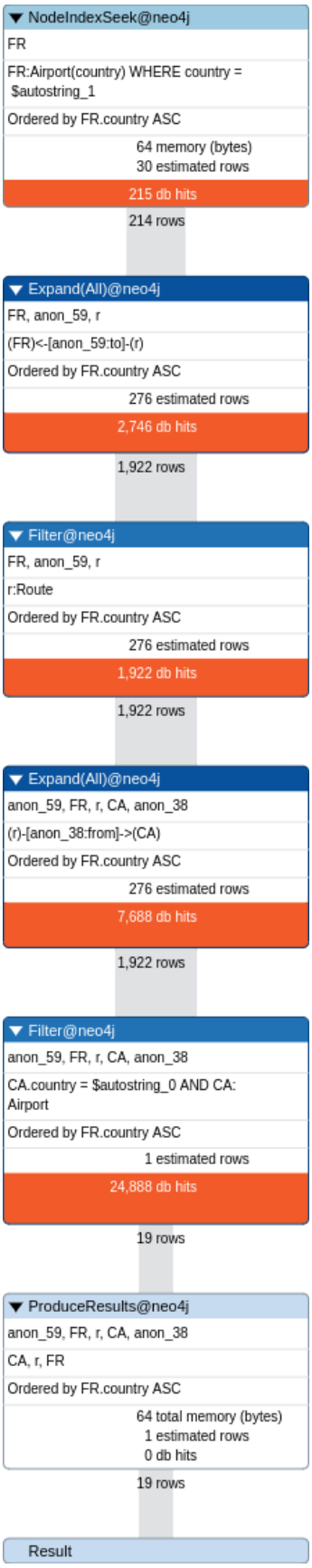
1. Quelques plans d'exécution

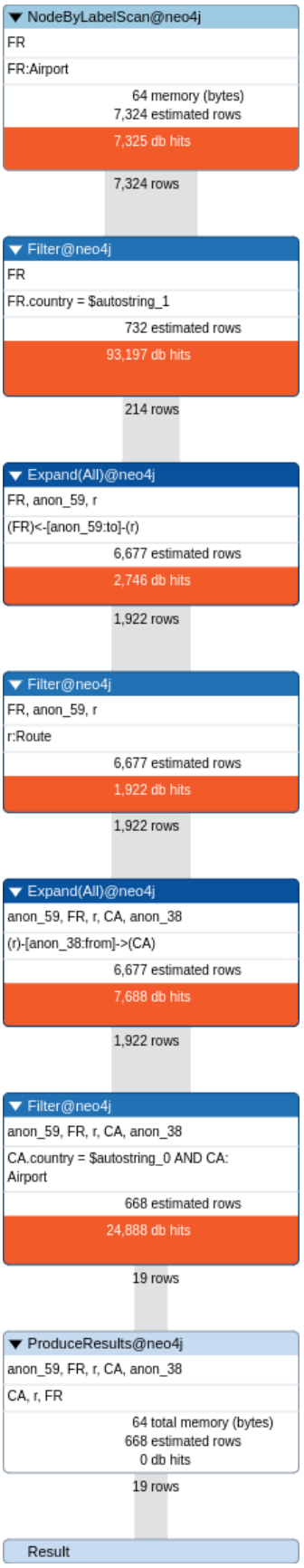
Ici, nous avons choisi d'illustrer quelques-unes de nos requêtes.
La requête suivante retourne le graphe de toutes les routes partant d'un aéroport Canadien, et allant vers un aéroport Français :
MATCH (CA:Airport{country:'Canada'}) <-[:from]- (r:Route) -[:to]->
(FR:Airport{country:'France'})
RETURN CA, r, FR

Cette requête retourne le graphe suivant:



Et en appliquant la méthode profile à cette requête, nous obtenons le plan d'exécution suivant, à gauche avec un index on Airport(country) , à droite sans l'index :



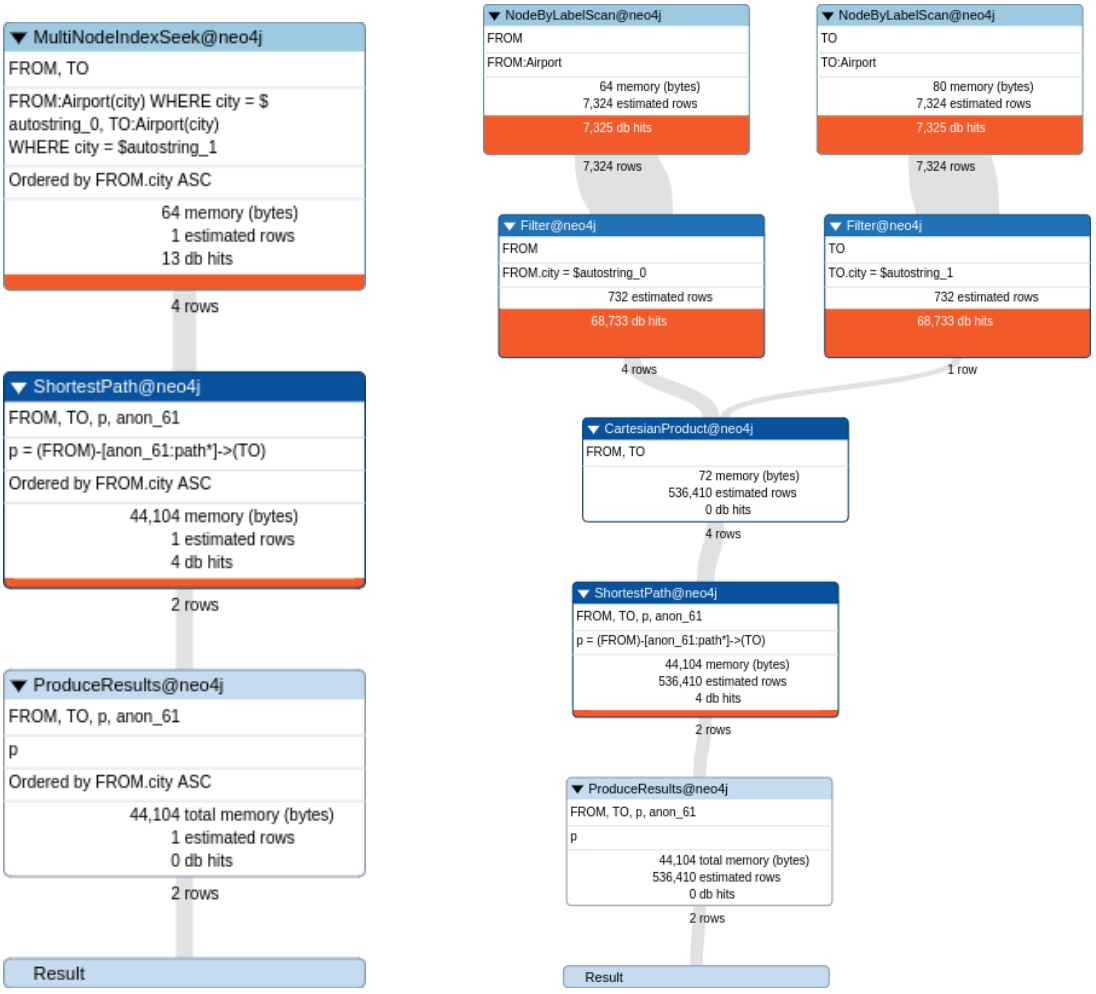


Nous remarquons que sur le plan d'exécution sans index, il a un parcours plus long des données avant d'arriver à l'information recherchée, avec toujours un nombre d'estimated rows plus élevé, alors qu'avec l'index on accède beaucoup plus rapidement à l'information.
En effet, nous avons créé un index sur la propriété "country" de Airport, ce qui permet d'accéder directement aux aéroports de chacun des deux pays de notre requête.
Cela n'est pas étonnant puisque le but des index est de pouvoir accéder plus rapidement aux informations de la base de données. Certes le fait d'avoir recours aux index utilise davantage d'espace de stockage et des écritures plus lentes, mais cela peut s'avérer très utile lors de certaines recherches d'informations, il suffit donc de bien décider de ce qu'il faut indexer.

Un autre exemple de requête où l'index est aussi important est la requête qui consiste à trouver le plus court chemin de Rio De Janeiro à Acapulco:

```
MATCH p=shortestpath( (FROM:Airport{city:'Rio De Janeiro'}) -[:path*]->
(TO:Airport{city:'Acapulco'}) )
RETURN p
```

Les plans d'exécution avec et sans index respectivement sont les suivants :



Ici, nous voyons qu'avec l'index, on accède directement aux aéroports des deux villes en question grâce à l'index sur le paramètre "city" de la table Airport, puis le chemin le plus court est retourné. Sans l'index, la recherche est beaucoup plus longue et le parcours de la table Aéroport se fait doublement.

2. Comparaison : Neo4j et Postgresql

Nous savons que pour ce type de données qui sont "nativement" de forme de graphe, une base de données "graph" est plus efficace qu'une base de données relationnelle. Nous allons donc voir dans cette section à quel point Neo4j est plus efficace que Postgresql dans ce cas précis. Plusieurs points peuvent être mentionnés comme : une syntaxe plus facile pour les requêtes, des algorithmes plus efficaces, etc…

La première différence que nous avons remarqué est : Postgresql nécessite un schéma au départ pour importer les données alors que Neo4j ne le demande pas. Par exemple, le schéma pour la table airport en Postgresql:

```
CREATE TABLE airport
(
    AirportID INT PRIMARY KEY NOT NULL,
    Name VARCHAR,
    City VARCHAR,
    Country VARCHAR,
    IATA VARCHAR,
    ICAO VARCHAR,
    Latitude FLOAT,
    Longitude FLOAT,
    Altitude VARCHAR,
    Timezone VARCHAR,
    DST VARCHAR,
    TZ VARCHAR,
    Type VARCHAR,
    Source VARCHAR
);
```

Comme les données sont représentées sous une structure de graphe en Neo4j, c'est-à-dire par les nœuds, les arêtes et les propriétés, les données peuvent être donc accumulées progressivement sans imposition d'un schéma rigide prédéfini. Cela est particulièrement adapté à la situation actuelle, qui connaît un phénomène du Big Data.

La deuxième différence remarquée est la syntaxe. En tant qu'informaticien, nous espérons toujours avoir une syntaxe simple et compréhensible pour écrire nos requêtes. Nous avons le langage Cypher d'une part, très déclaratif, expressif et intuitif pour écrire les requêtes de "graph" en utilisant des pattern matching et d'autre part le langage SQL qui est très inefficace pour exprimer les patterns de graphe complexe, en particulier avec les requêtes récursives et celles pouvant accepter des longueurs différentes.
Par exemple pour les mêmes requêtes :
all routes coming from a Canadian airport to French airport
En SQL :

```
SELECT r.*
FROM route AS r, airport AS a1, airport AS a2
WHERE r.sourceairportid = a1.airportid AND a1.country = 'Canada'
    AND r.destairportid = a2.airportid AND a2.country = 'France';
```

En Cypher :

```
MATCH (CA:Airport{country:'Canada'}) <-[:from]- (r:Route) -[:to]-> (FR:Airport{country:'France'})
RETURN CA, r, FR
```

Ou encore pire pour la requête : #paths delivered by ”Air Canada” between all Canadian airports

En SQL :

```
WITH RECURSIVE access(sourceairportid, destairportid) AS
(
    SELECT sourceairportid, destairportid
    FROM route AS r, airport AS a1, airport AS a2, airline AS al
    WHERE r.sourceairportid = a1.airportid AND a1.country = 'Canada'
    AND r.destairportid = a2.airportid AND a2.country = 'Canada'
    AND r.airlineid = al.airlineid AND al.name = 'Air Canada'
    UNION
    SELECT r.sourceairportid, a.destairportid
    FROM route AS r, access AS a, airport AS a1, airport AS a2,airline AS al
    WHERE r.destairportid = a.sourceairportid
    AND r.sourceairportid = a1.airportid AND a1.country = 'Canada'
    AND r.destairportid = a2.airportid AND a2.country = 'Canada'
    AND r.airlineid = al.airlineid AND al.name = 'Air Canada'
)
SELECT count(*)
FROM access;
```

En Cypher :

```
MATCH (from:Airport{country:'Canada'}) -[:path{airline:'Air Canada'}]-> (to:Airport{country:'Canada'})
RETURN from, to
```

Nous observons que les requêtes en Cypher sont beaucoup plus simples et intuitives à faire qu’en Postgresql.

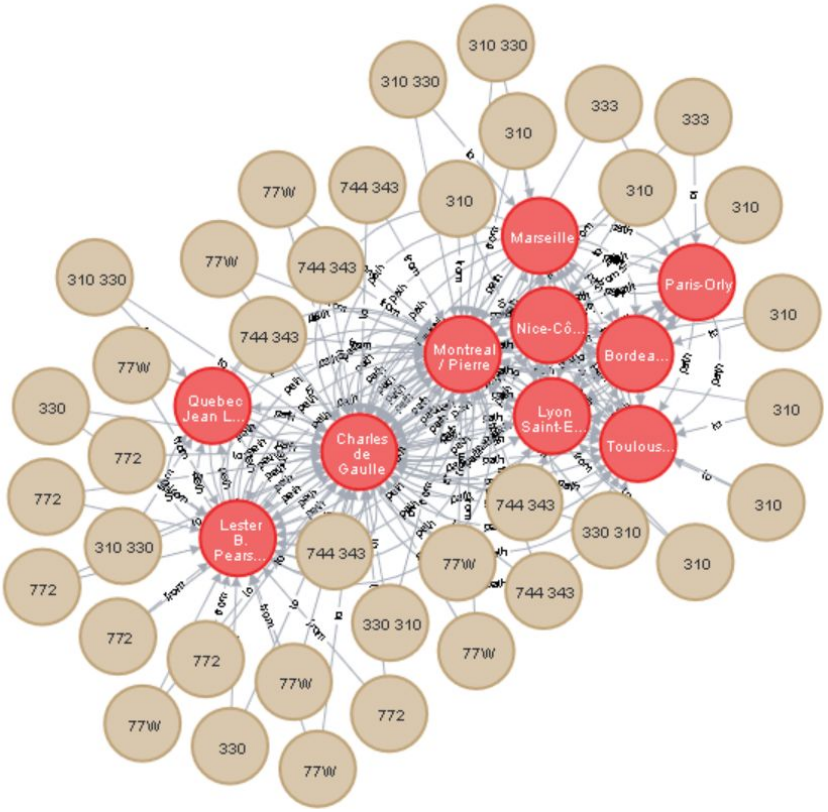
La troisième remarque que nous avons constaté est que Neo4j donne des résultats beaucoup plus interprétables que Postgresql parce que Neo4j donne la possibilité de donner les résultats sous forme de graphe.
Par exemple pour la requête : # all routes coming from a Canadian airport to French airport

En SQL :

airline	airlineid	sourceairport	sourceairportid	destairport	destairportid	codeshare	stops	equipment
AC	330	YUL	146	CDG	1382		0	77W
AC	330	YYZ	193	CDG	1382		0	77W
AF	137	YUL	146	CDG	1382		0	744 343
AF	137	YYZ	193	CDG	1382		0	772
AZ	596	YUL	146	CDG	1382	Y	0	744 343
AZ	596	YYZ	193	CDG	1382	Y	0	772
DL	2009	YUL	146	CDG	1382	Y	0	744 343
DL	2009	YYZ	193	CDG	1382	Y	0	772
LH	3320	YUL	146	CDG	1382	Y	0	77W
LH	3320	YYZ	193	CDG	1382	Y	0	77W
SS	1908	YUL	146	ORY	1386		0	333
TS	1317	YQB	111	CDG	1382		0	310 330
TS	1317	YUL	146	BOD	1264		0	310
TS	1317	YUL	146	CDG	1382		0	330 310
TS	1317	YUL	146	LYS	1335		0	310
TS	1317	YUL	146	MRS	1353		0	310 330
TS	1317	YUL	146	NCE	1354		0	310
TS	1317	YUL	146	TLS	1273		0	310
TS	1317	YYZ	193	CDG	1382		0	330

(19 rows)

En Cypher :



Le dernier point que nous avons observé est l'efficacité des algorithmes et des requêtes, c'est-à-dire le temps de réponse. Les faiblesses des bases de données relationnelles sont :

- Incapable de gérer de très grands volumes de données.
- Impossible de gérer des débits extrêmes.
- Surcoût en latence, accès au disque, temps CPU.
- Performances limitées par les accès disques.

Alors que Neo4j est très puissant pour gérer les grands volumes de données avec des débits élevés. Cette différence vient essentiellement du fait d'éviter les jointures classiques. Par exemple, en SQL, pour retourner les vols en n escales demande n jointures, cela entraîne au phénomène "join pain". Pour essayer de mieux comprendre cette différence, nous pouvons regarder le plan d'exécution de certaines requêtes.

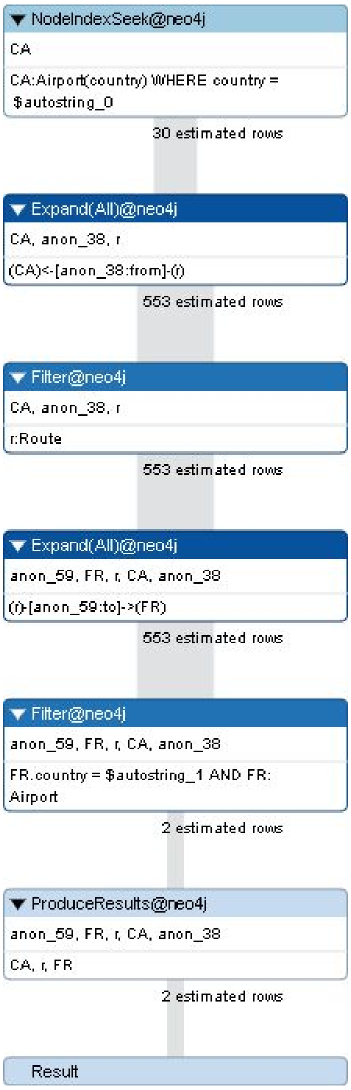
Par exemple :

all routes coming from a Canadian airport to French airport

En SQL :

```
neo4j=> EXPLAIN
neo4j-> SELECT r.*
neo4j-> FROM route AS r, airport AS a1, airport AS a2
neo4j-> WHERE r.sourceairportid = a1.airportid AND a1.country = 'Canada'
neo4j->  AND r.destairportid = a2.airportid AND a2.country = 'France';
                                        QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Hash Join  (cost=304.54..1726.77 rows=115 width=34)
   Hash Cond: (r.sourceairportid = a1.airportid)
   ->  Hash Join  (cost=149.60..1566.61 rows=1989 width=34)
         Hash Cond: (r.destairportid = a2.airportid)
         ->  Seq Scan on route r  (cost=0.00..1241.65 rows=66765 width=34)
         ->  Hash  (cost=146.93..146.93 rows=214 width=4)
               ->  Bitmap Heap Scan on airport a2  (cost=5.94..146.93 rows=214 width=4)
                     Recheck Cond: ((country)::text = 'France'::text)
                     ->  Bitmap Index Scan on idx_airport_country  (cost=0.00..5.89 rows=214 width=0)
                           Index Cond: ((country)::text = 'France'::text)
   ->  Hash  (cost=149.73..149.73 rows=417 width=4)
         ->  Bitmap Heap Scan on airport a1  (cost=11.51..149.73 rows=417 width=4)
               Recheck Cond: ((country)::text = 'Canada'::text)
               ->  Bitmap Index Scan on idx_airport_country  (cost=0.00..11.41 rows=417 width=0)
                     Index Cond: ((country)::text = 'Canada'::text)
```

En Cypher :

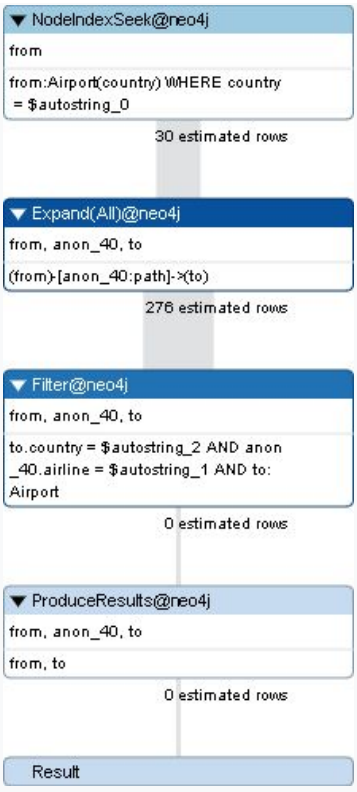


Ou encore pire la requête: #paths delivered by ”Air Canada” between all Canadian airports

En SQL :

```
                                                         QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=2485.59..2485.60 rows=1 width=8)
   CTE access
     ->  Recursive Union  (cost=139.60..2485.34 rows=11 width=8)
           ->  Nested Loop  (cost=139.60..1559.81 rows=1 width=8)
                 ->  Nested Loop  (cost=139.32..1559.50 rows=1 width=8)
                       ->  Hash Join  (cost=139.04..1556.07 rows=11 width=8)
                             Hash Cond: (r.airlineid = al.airlineid)
                             ->  Seq Scan on route r  (cost=0.00..1241.65 rows=66765 width=12)
                             ->  Hash  (cost=139.03..139.03 rows=1 width=4)
                                   ->  Seq Scan on airline al  (cost=0.00..139.03 rows=1 width=4)
                                         Filter: ((name)::text = 'Air Canada'::text)
                       ->  Index Scan using idx_airport_id on airport a1  (cost=0.28..0.31 rows=1 width=4)
                             Index Cond: (airportid = r.sourceairportid)
                             Filter: ((country)::text = 'Canada'::text)
                 ->  Index Scan using idx_airport_id on airport a2  (cost=0.28..0.31 rows=1 width=4)
                       Index Cond: (airportid = r.destairportid)
                       Filter: ((country)::text = 'Canada'::text)
           ->  Nested Loop  (cost=1.14..92.53 rows=1 width=8)
                 ->  Nested Loop  (cost=0.86..92.22 rows=1 width=8)
                       ->  Nested Loop  (cost=0.57..87.58 rows=15 width=12)
                             Join Filter: (a.sourceairportid = r_1.destairportid)
                             ->  Nested Loop  (cost=0.28..79.26 rows=1 width=12)
                                   ->  WorkTable Scan on access a  (cost=0.00..0.20 rows=10 width=8)
                                   ->  Index Scan using idx_airport_id on airport a2_1  (cost=0.28..7.90 rows=1 width=4)
                                         Index Cond: (airportid = a.sourceairportid)
                                         Filter: ((country)::text = 'Canada'::text)
                             ->  Index Scan using idx_dest_airport_id on route r_1  (cost=0.29..8.01 rows=25 width=12)
                                   Index Cond: (destairportid = a2_1.airportid)
                       ->  Index Scan using idx_airline_id on airline al_1  (cost=0.28..0.31 rows=1 width=4)
```

En Cypher :



Nous constatons que pour la même requête, Cypher n'a utilisé aucune jointure alors que SQL a utilisé des jointures . En fait, pour effectuer les requêtes, Cypher utilise des parcours de graphes, il n'y a donc pas d'opération de jointure explicite car les nœuds maintiennent la référence directe de leurs arêtes adjacentes. Les arêtes servent de structure de jointure explicite, hard-wired (non calculé à la volée) et ce qui rend ce parcours efficace est le fait d'aller d'un nœud vers un autre est une opération O(1) en temps constant. Au contraire, SQL effectue les requêtes via les jointures qui sont extrêmement chers. L'opération de jointure forme un graphe dynamiquement construit au fur et à mesure qu'une table est liée à une autre. Malgré une construction dynamique, la structure de graphe implicite doit être inférée via des opérations exigeantes en termes d'indexes. De plus, alors que seulement un sous ensemble des données est pertinent (les vols pour départ Canada et arrivé France), toutes les données de la table doivent être examinées pour extraire le sous ensemble voulu.

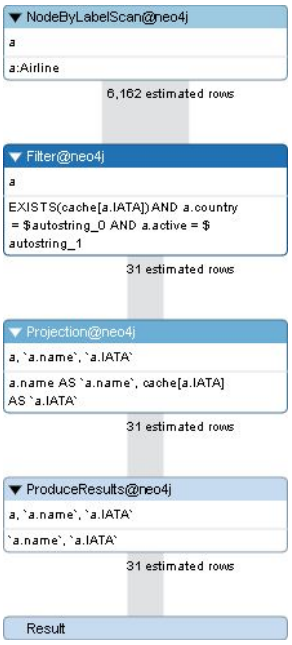
(Une requête aussi bon en Cypher qu'en SQL) :
En Cypher :

```
MATCH (a:Airline{country:'Canada'})
WHERE EXISTS(a.IATA) AND a.active = 'Y'
RETURN a.name, a.IATA
```

En SQL :

```
SELECT name,IATA
FROM airline
WHERE EXISTS(SELECT IATA FROM airline) AND active = 't' AND country = 'Canada';
```

Les plans d'exécution:
En Cypher :



En SQL :

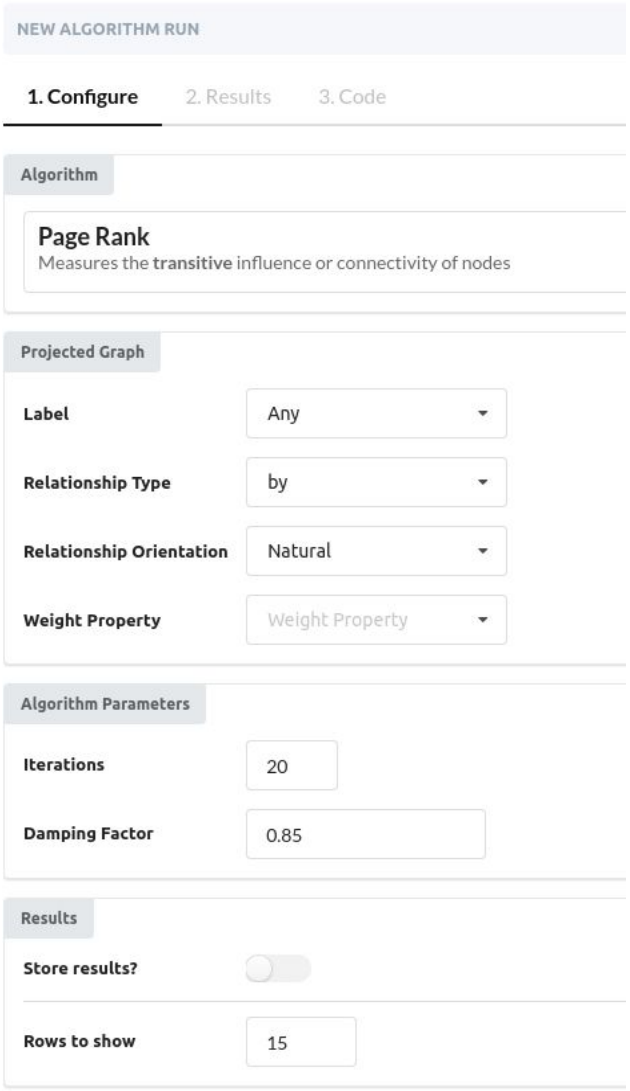
```
                                QUERY PLAN
---------------------------------------------------------------------------
 Result  (cost=0.02..139.05 rows=66 width=19)
   One-Time Filter: $0
   InitPlan 1 (returns $0)
     ->  Seq Scan on airline airline_1  (cost=0.00..123.62 rows=6162 width=0)
   ->  Seq Scan on airline  (cost=0.02..139.05 rows=66 width=19)
         Filter: (active AND ((country)::text = 'Canada'::text))
(6 rows)
```

Nous constatons que Neo4j et Postgresql n'ont utilisé aucune jointure pour les requêtes. Nous observons que Neo4j a utilisé un scan by label, un filter et une projection alors que SQL a utilisé deux scan et un filter. Donc nous nous permettons de dire que ces deux requêtes sont aussi efficaces l'un que l'autre.

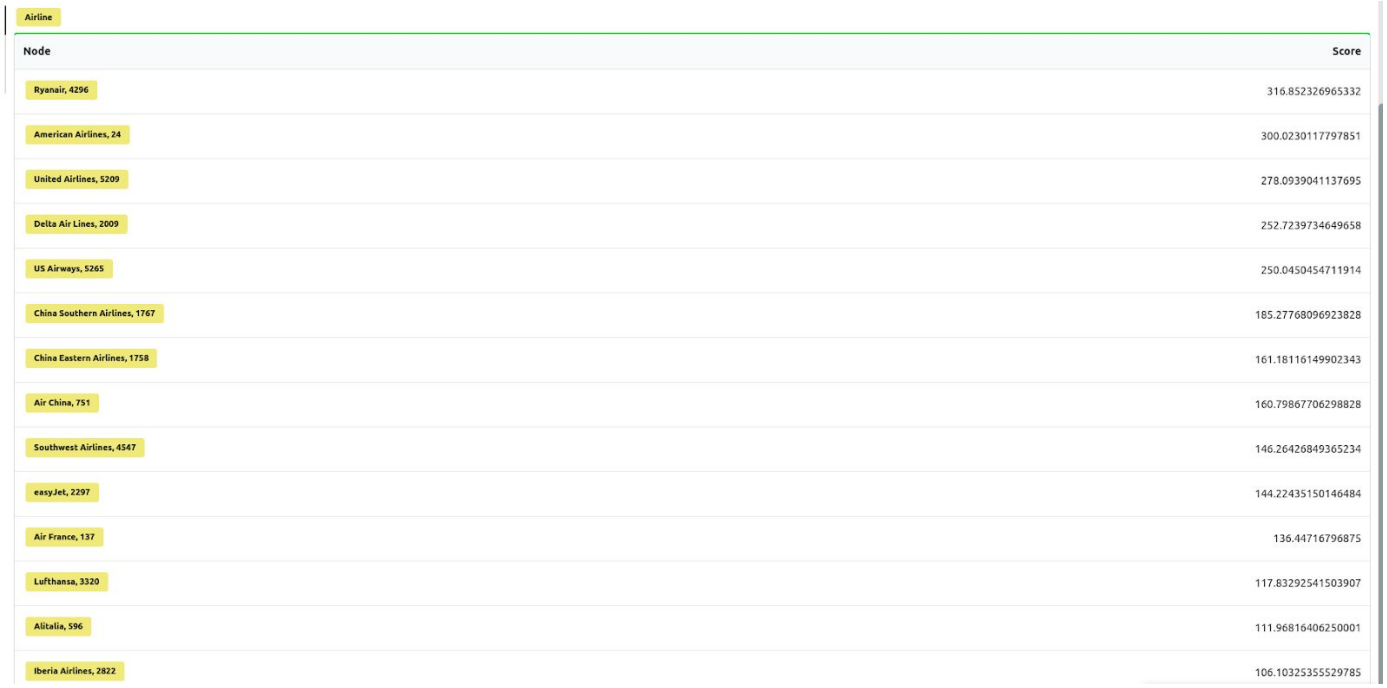
III. Analytique de graphe

1. L'algorithme PageRank sur la relationship "by":

L'algorithme de PageRank étant un algorithme mesurant l'importance de chaque noeud dans le graphe, en fonction du nombre de relations entrantes et de l'importance des nœuds source correspondants, nous avons décidé d'utiliser cet algorithme de centralité, avec les paramètres suivants:



En le faisant tourner avec le mode stream, cela nous donne les compagnies aériennes les plus utilisées dans notre base de données, puisque nous avons précisé le type de relationship “by”. Et nous pouvons bien voir que le résultat obtenu est bien conforme à la réalité. En effet, il s’avère que Ryanair est la première compagnie aérienne en Europe car elle propose des vols à bas prix. Puis à la deuxième position arrive American Airlines qui est la plus grande compagnie aérienne au monde puisqu’elle exploite des vols intérieurs et internationaux depuis ses nombreux hubs basés aux États-Unis, suivie de près par United Airlines qui est également une des compagnies aériennes les plus importantes au monde.
Ensuite viennent les compagnies chinoises qui sont bien entendu des compagnies très fréquemment utilisées pour des vols internes ou internationaux, qui comptent des nombres très importants de vols.



Airline	Node	Score
	Ryanair, 4296	316.852326965332
	American Airlines, 24	300.0230117797851
	United Airlines, 5209	278.0939041137695
	Delta Air Lines, 2009	252.7239734649658
	US Airways, 5265	250.0450454711914
	China Southern Airlines, 1767	185.27768096923828
	China Eastern Airlines, 1758	161.18116149902343
	Air China, 751	160.79867706298828
	Southwest Airlines, 4547	146.26426849365234
	easyJet, 2297	144.22435150146484
	Air France, 137	136.44716796875
	Lufthansa, 3320	117.83292541503907
	Alitalia, 596	111.96816406250001
	Iberia Airlines, 2822	106.10325355529785

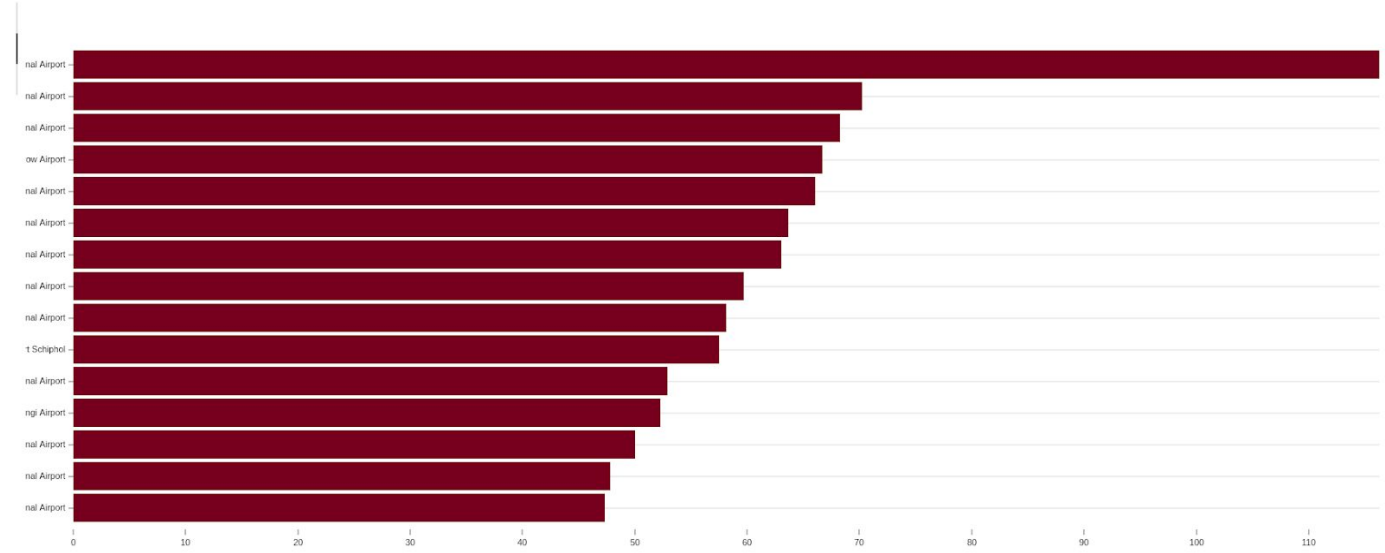
Airline	Node
	Ryanair, 4296
	American Airlines, 24
	United Airlines, 5209
	Delta Air Lines, 2009
	US Airways, 5265
	China Southern Airlines, 1767
	China Eastern Airlines, 1758
	Air China, 751
	Southwest Airlines, 4547
	easyJet, 2297
	Air France, 137
	Lufthansa, 3320
	Alitalia, 596
	Iberia Airlines, 2822

En utilisant le mode write, nous obtenons les mêmes résultats.

En décidant donc d'utiliser cet algorithme de centralité, nous avons pu mettre en évidence les compagnies aériennes les plus fréquemment utilisées.

2. L’algorithme PageRank sur la relationship “to”:

En appliquant le même algorithme de centralité sur la relationship “to”, nous obtenons une liste triée des aéroports les plus visités (avec le plus de vols d’arrivée) de notre base de données.



Nous obtenons la liste suivante d’aéroports:

Airport	
Node	
Hartsfield Jackson Atlanta International Airport, 3682	
Chicago O'Hare International Airport, 3830	Dallas Fort Worth International Airport, 3670
Beijing Capital International Airport, 3364	John F Kennedy International Airport, 3797
London Heathrow Airport, 507	Amsterdam Airport Schiphol, 580
Charles de Gaulle International Airport, 1382	Shanghai Pudong International Airport, 3406
Los Angeles International Airport, 3484	Singapore Changi Airport, 3316
Frankfurt am Main International Airport, 340	Barcelona International Airport, 1218

Encore ici, ces résultats sont très concordant avec la réalité puisque l’aéroport international Hartsfield-Jackson est le plus grand aéroport des États-Unis et du monde par nombre de passagers ainsi que par le nombre de mouvements d'aéronefs.

3. Algorithme Shortest Path :

Nous avons appliqué l'algorithme de recherche du plus court chemin Shortest Path en prenant comme exemple de start node "CDG" et de end node "GYE". Et nous obtenons le chemin suivant pour arriver à José Joaquín de Olmedo International Airport (Guayaquil, Équateur) en partant de Charles de Gaulle International Airport (Paris, France) :

Labels	Properties	Cost
Airport	Charles de Gaulle International Airport, 1382	0
Airport	Tocumen International Airport, 1871	1
Airport	José Joaquín de Olmedo International Airport, 2673	2

Il s'agit de l'algorithme Dijkstra, qui est efficace pour retrouver le plus court chemin entre deux destinations prédéfinies. Nous avons essayé en mettant le nom d'un pays en end node et l'algorithme a réussi à retrouver le plus court chemin.

Bonus ?

Pourquoi pas les autres algorithmes de centrality de la GDS ?

- Betweenness : suppose que toutes les communications entre les nœuds se produisent le long du chemin le plus court et avec la même fréquence, ce qui n'est pas le cas dans la vie réelle, ni dans notre base de données.
- Closeness : fonctionne mieux sur les graphes connexes. Si nous utilisons la formule originale sur un graphe non connexe, nous pouvons nous retrouver avec une distance infinie entre deux nœuds dans des composantes connexes séparées. Cela signifie que nous nous retrouverons avec un score de centralité de proximité infini lorsque nous additionnons toutes les distances de ce nœud.

Sources:

- https://fr.wikipedia.org/wiki/A%C3%A9roports_les_plus_fr%C3%A9quent%C3%A9s_du_monde_par_des_mouvements_d%27a%C3%A9ronefs
- https://fr.wikipedia.org/wiki/A%C3%A9roport_international_Hartsfield-Jackson_d%27Atlanta
- https://neo4j.com/docs
- https://neo4j.com/docs/graph-algorithms
- cours Moodle