Projet BDS
Implémentation d’une graphe dans Neo4j

Année universitaire : 2023/2024

Zenkri Jihed: 21964937
Helal Imad: 21969357

Sommaire:

I) Jeu de données (Choix et import) :

1-Choix: base de données Northwind

La base de données Northwind, créée par Microsoft, contient les données de vente de “Northwind Traders”, une société fictive d’import-export de produits alimentaires spécialisés. Elle est utilisée pour les tutoriels de divers produits de base de données et a été portée sur plusieurs bases de données non Microsoft, dont PostgreSQL.

L’ensemble de données Northwind comprend des informations sur les fournisseurs, clients, employés, produits, expéditeurs, ainsi que les commandes et leurs détails. La base de données comprend 14 tables.

lien vers le jeu de données :
https://github.com/Microsoft/sql-server-samples/tree/master/samples/databases/northwind-pubs

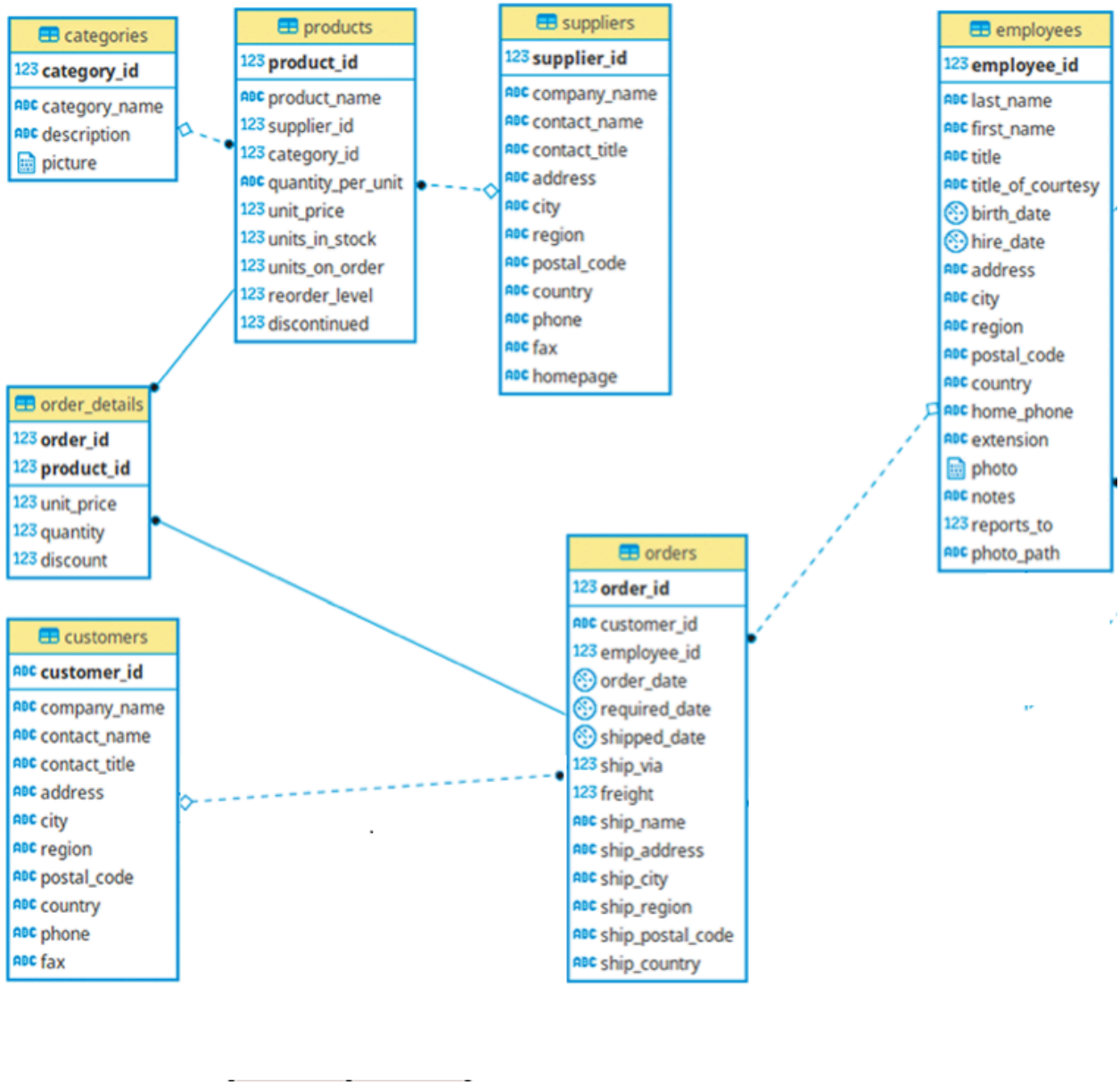
Pour notre projet, nous avons sélectionné 6 tables, modélisé par le schéma en dessous :

- **Suppliers** : Cette table contient les informations sur les fournisseurs et vendeurs de Northwind.
- **Customers** : les clients qui achètent des produits de Northwind.
- **Employees** : les détails sur les employés de Northwind Traders.
- **Products** : informations sur les produits.
- **Categories** : informations sur les catégories des produits.
- **Orders** : les commandes passées.

REMARQUE 1:
La table **order_details** est le résultat d’une relation n-n entre les tables Orders et Products. car, chaque commande peut contenir plusieurs produits et chaque produit peut figurer dans plusieurs commandes.

REMARQUE 2:
dans la table **employees**, l’attribut **reports_to** est une clé étrangère qui fait référence à la même table employees

Schéma relationnel



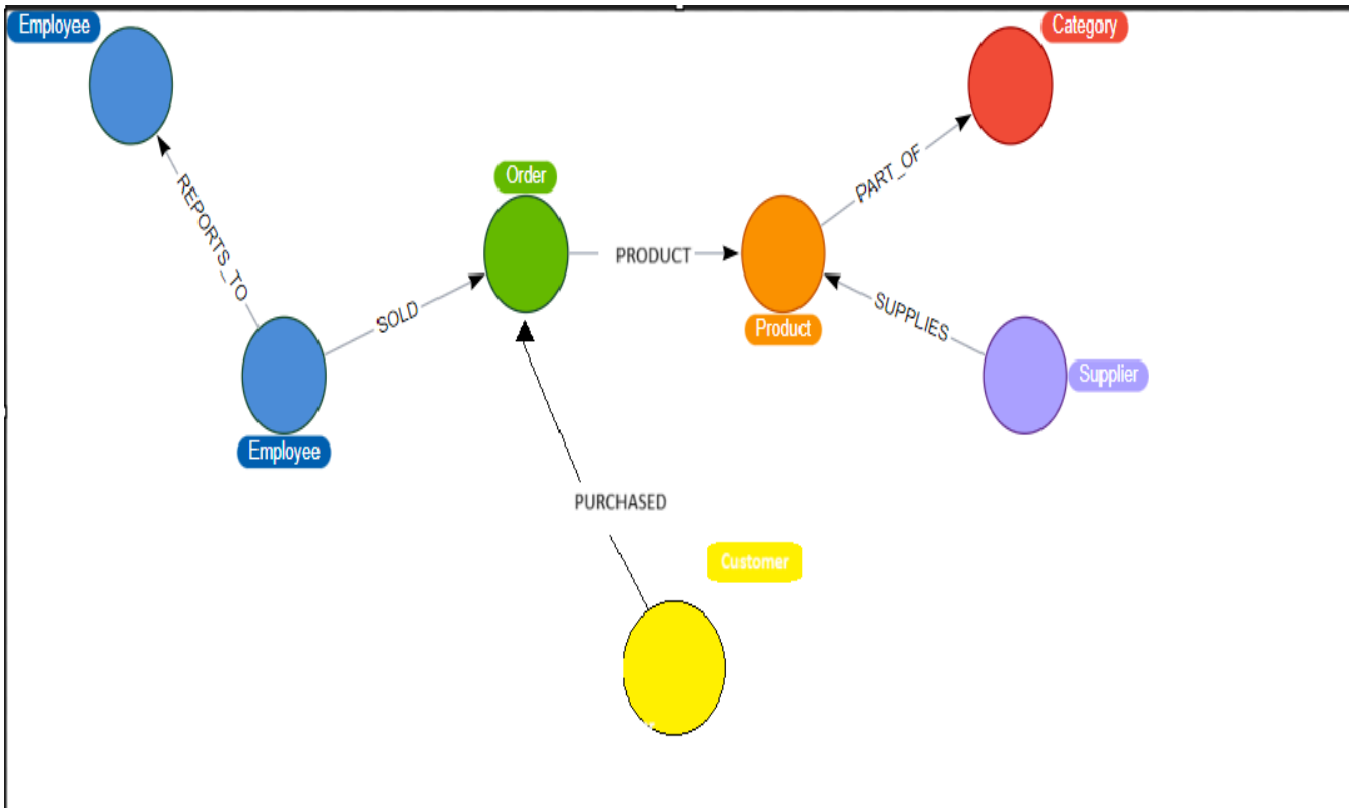
Voici la stratégie qu'on a utilisé pour convertir le modèle relationnel en un modèle graph:

1- Une ligne est un nœud.

2- Un nom de table est un nom d'étiquette.

3- Une jointure ou une clé étrangère est une relation

on obtient ainsi le data model pour le graph :



2-Import :

1-Postgresql : Dans le dossier postgres,exécuter les fichiers suivants avec la commande ‘ \i ‘ :

- **init_tables.sql :** initialise les tables avec les contraintes d’intégrité.
- **fill_tables.sql:** remplit les tables à partir des fichiers CSV. N’oubliez pas de remplacer le chemin absolu par votre chemin correspondant vers les fichiers CSV.

2-**Cypher:** Dans le dossier Neo4j, exécuter le fichier suivant :

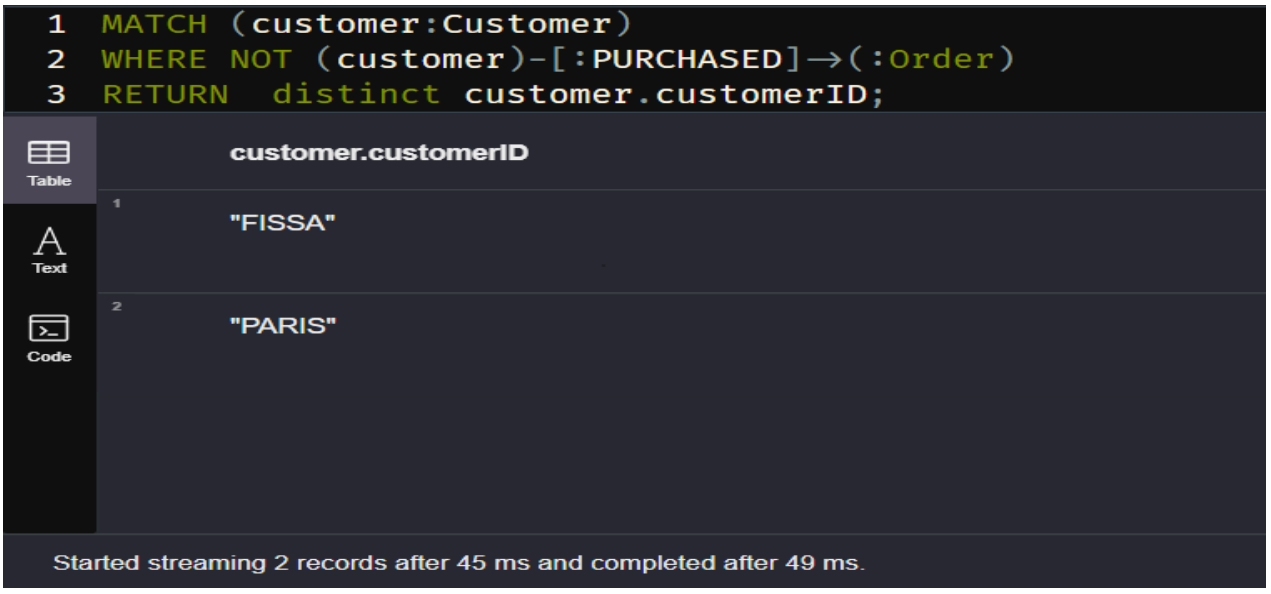
- **import_csv.cypher :** initialise et remplit le graphe à partir des fichiers CSV.

II)Requêtes :

-les requêtes cypher sont dans le fichier: cypher_queries.cypher
-les requêtes SQL équivalentes sont dans le fichier : SQL_equivalents.sql

1-Une requête avec un filtre négatif

Cette requête extrait les clients qui n’ont pas effectué d’achat.



Requête équivalente en SQL :

```
SELECT DISTINCT c.CustomerID
FROM customers c
LEFT JOIN orders o ON c.CustomerID = o.CustomerID
WHERE o.CustomerID IS NULL;
```

2- OPTIONAL MATCH :

Cette requête extrait les employés, leur manager et le nombre total de ventes effectuées par chaque employé si y'en a .

```
1 MATCH (e:Employee)-[:REPORTS_TO]→(manager:Employee)
2 OPTIONAL MATCH (e)-[:SOLD]→(o:Order)
3 RETURN e.employeeID, manager.employeeID, COUNT(o) AS totalSales
4 order by totalSales;
```

Table | Text | Code

	e.employeeID	manager.employeeID	totalSales
1	"5"	"2"	42
2	"9"	"5"	43
3	"6"	"5"	67
4	"7"	"5"	72
5	"8"	"2"	104
6	"1"	"2"	123
7	"3"	"2"	127
8	"4"	"2"	156

Requête équivalente en SQL :

```
SELECT e.employeeID, e.ReportsTo AS managerID,
COUNT(o.OrderID) AS totalSales
FROM employees e
LEFT JOIN orders o ON e.employeeID = o.EmployeeID
WHERE e.ReportsTo IS NOT NULL
GROUP BY e.employeeID, e.ReportsTo;
```

3 et 4-Deux utilisations de WITH différentes :

*with pour filtrer les résultats d'un agrégat:

Cette requête extrait les identifiants de commande et le nombre de produits vendus pour chaque commande.

```
MATCH (order:Order)-[pu:PRODUCT]→(product:Product)
WITH order, COUNT(pu) AS numberOfProducts
RETURN order.orderID, numberOfProducts
ORDER BY numberOfProducts DESC
LIMIT 20;
```

Table | Text | Code

	order.orderID	numberOfProducts
1	"11077"	25
2	"10979"	6
3	"10657"	6
4	"10847"	6
5	"10893"	5
6	"10612"	5

Requête équivalente en SQL :

```
SELECT orders.orderID, COUNT(order_details.productID) AS
numberOfProducts
FROM orders
JOIN order_details ON orders.orderID = order_details.orderID
GROUP BY orders.orderID
ORDER BY numberOfProducts DESC
LIMIT 20;
```

*with pour séparer lecture et mise à jour du graphe:

Cette requête met à jour les commandes qui ont un nombre total de produits vendus supérieur à 100 en ajoutant une propriété “highQuantity” à true.

```
1 MATCH (order:Order)-[pu:PRODUCT]→(product:Product)
2 WITH order, SUM(pu.quantity) AS totalQuantity
3 WHERE totalQuantity > 100
4 SET order.highQuantity = true;
```

5-Une requête utilisant COLLECT et UNWIND:

Cette requête extrait les noms des entreprises des clients, les identifiants des commandes et le nombre distinct de produits achetés pour chaque commande.

```
MATCH (customer:Customer)-[:PURCHASED]→(order:Order)-[pu:PRODUCT]→(product:Product)
WITH customer, order, COLLECT(product) AS purchasedProducts
UNWIND purchasedProducts AS purchasedProduct
RETURN customer.companyName, COUNT(DISTINCT purchasedProduct) AS numberOfProductsPurchased
ORDER BY numberOfProductsPurchased DESC;
```

Table | Text | Code

	customer.companyName	numberOfProductsPurchased
1	"Ernst Handel"	56
2	"Save-a-lot Markets"	53
3	"QUICK-Stop"	49
4	"Rattlesnake Canyon Grocery"	45
5	"Berglunds snabbköp"	37
6	"Hungry Owl All-Night Grocers"	36
7		

Started streaming 89 records in less than 1 ms and completed after 13 ms.

Requête équivalente en SQL :

```
SELECT c.companyName, COUNT(DISTINCT p.productID) AS
numberOfProductsPurchased
FROM customers c
JOIN orders o ON c.customerID = o.CustomerID
JOIN order_details od ON o.orderID = od.orderID
JOIN products p ON od.productID = p.productID
GROUP BY c.companyName
ORDER BY numberOfProductsPurchased DESC
LIMIT 10;
```

6-Une requête utilisant REDUCE:

Cette requête Cypher extrait et affiche les noms des fournisseurs ainsi que le total des unités en commande pour chaque fournisseur.

```
MATCH (supplier:Supplier)-[:SUPPLIES]→(product:Product)
WITH supplier, COLLECT(product.UnitsOnOrder) AS unitOnOrders
WITH
  supplier.companyName AS supplierName,
  REDUCE(totalunitOnOrders = 0, unitOnOrders IN unitOnOrders | totalunitOnOrders + unitOnOrders) AS totalSupplierunitOnOrders
WHERE totalSupplierunitOnOrders <> 0
RETURN supplierName, totalSupplierunitOnOrders
ORDER BY totalSupplierunitOnOrders DESC;
```

Table | Text | Code

	supplierName	totalSupplierunitOnOrders
1	"Exotic Liquids"	110
2	"Formaggi Fortini s.r.l."	110
3	"New Orleans Cajun Delights"	100
4	"Plutzer Lebensmittelgroßmärkte AG"	80
5	"Lyngbysild"	70
6	"Zaanse Snoepfabriek"	70

Requête équivalente en SQL :

```
SELECT s.companyName AS supplierName,
       SUM(p.UnitsOnOrder) AS total_products_on_order
FROM suppliers s
JOIN products p ON s.supplierID = p.supplierID
GROUP BY s.companyName
HAVING SUM(p.UnitsOnOrder) > 0
ORDER BY total_products_on_order DESC;
```

7-Un filtre post UNION avec CALL :

Cette requête identifie l'employé qui a vendu le plus et celui qui a vendu le moins de produits.

```
CALL {
    MATCH (employee:Employee)-[:SOLD]→(order:Order)-[:PRODUCT]→(product:Product)
    RETURN employee, COUNT(DISTINCT product) AS soldProducts
    ORDER BY soldProducts ASC LIMIT 1
        UNION
    MATCH (employee:Employee)-[:SOLD]→(order:Order)-[:PRODUCT]→(product:Product)
    RETURN employee, COUNT(DISTINCT product) AS soldProducts
    ORDER BY soldProducts DESC LIMIT 1
}
RETURN employee.firstName, employee.lastName, soldProducts
ORDER BY soldProducts DESC;
```

Table | Text | Code

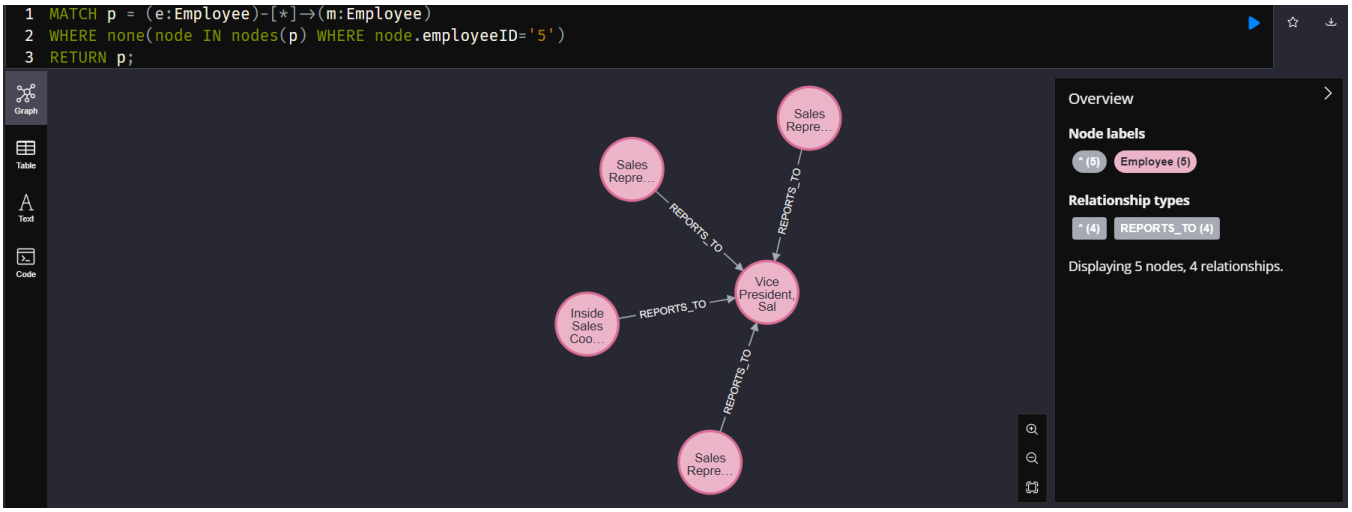
	employee.firstName	employee.lastName	soldProducts
1	"Margaret"	"Peacock"	75
2	"Steven"	"Buchanan"	52

Requête équivalente en SQL :

```
(SELECT e.employeeID, e.firstName, e.lastName,
COUNT(DISTINCT p.productID) AS soldProducts
 FROM employees e
 JOIN orders o ON e.employeeID = o.EmployeeID
 JOIN order_details od ON o.orderID = od.orderID
 JOIN products p ON od.productID = p.productID
 GROUP BY e.employeeID, e.firstName, e.lastName
 ORDER BY soldProducts ASC LIMIT 1)
UNION ALL
(SELECT e.employeeID, e.firstName, e.lastName,
COUNT(DISTINCT p.productID) AS soldProducts
 FROM employees e
 JOIN orders o ON e.employeeID = o.EmployeeID
 JOIN order_details od ON o.orderID = od.orderID
 JOIN products p ON od.productID = p.productID
 GROUP BY e.employeeID, e.firstName, e.lastName
 ORDER BY soldProducts DESC LIMIT 1);
```

8-Requête avec none():

-Cette requête utilise le prédicat none() pour trouver un chemin (p) entre deux employés (e et m) sans aucun nœud ayant l'identifiant d'employé '5'.



9-Requête avec any():

-Cette requête utilise le prédicat any() pour trouver des produits (p) qui font partie d'une catégorie (cat) ayant pour nom 'Beverages' ou 'Dairy Products'.

```
1 MATCH (p:Product)-[:PART_OF]→(cat:Category)
2 WHERE ANY(category IN ['Beverages', 'Dairy Products'] WHERE cat.categoryName = category)
3 RETURN p.productName, cat.categoryName;
```

Table | Text | Code

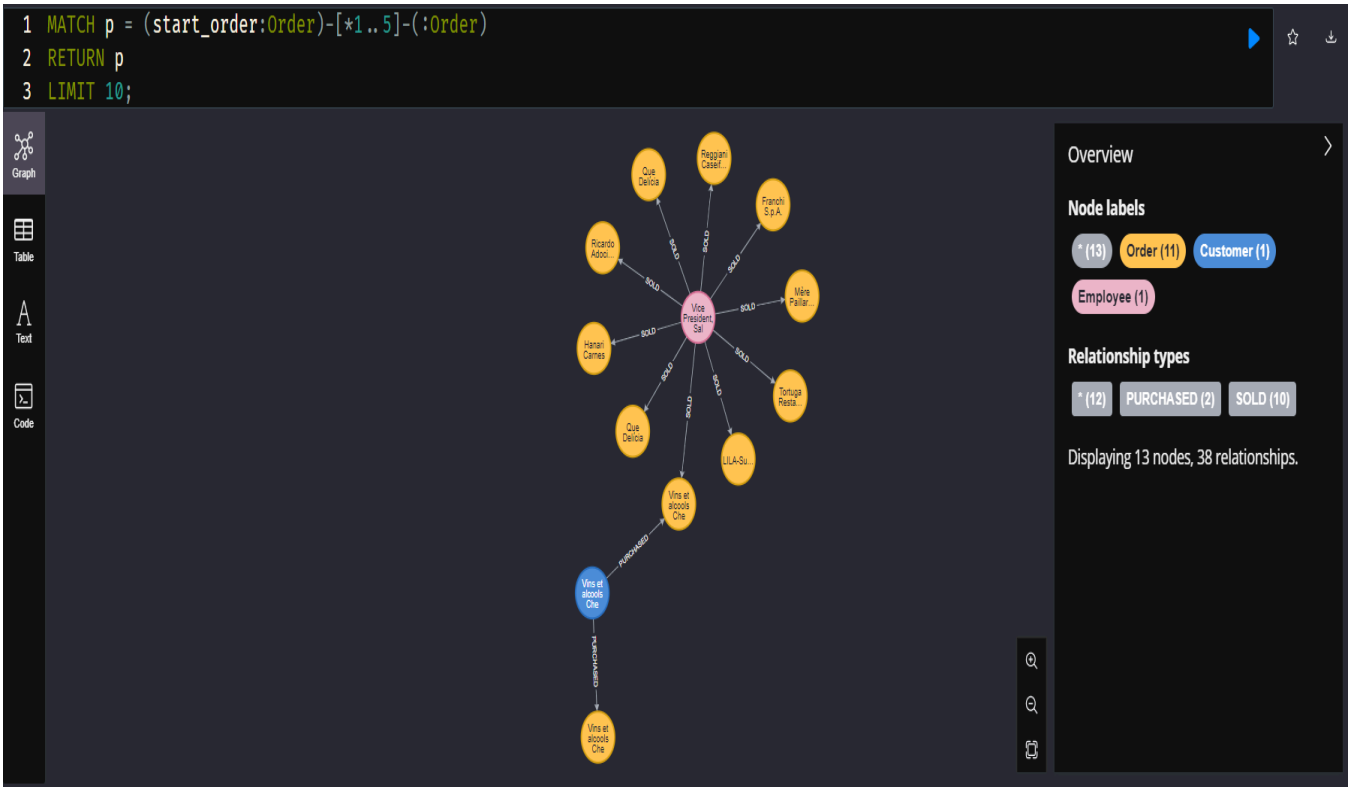
	p.productName	cat.categoryName
1	"Outback Lager"	"Beverages"
2	"Laughing Lumberjack Lager"	"Beverages"
3	"Ipoh Coffee"	"Beverages"
4	"Chai"	"Beverages"
5	"Lakkalikööri"	"Beverages"
6	"Rhönbräu Klosterbier"	"Beverages"
7		

Requête équivalente en SQL :

```
SELECT p.productID, p.productName, c.categoryName
FROM products p NATURAL JOIN categories c
WHERE c.categoryName IN ('Beverages', 'Dairy Products')
RETURN p
```

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

Cette requête recherche les commandes (nœuds de type "Order") qui peuvent être connectées les unes aux autres par des relations, avec une longueur de chemin allant de 1 à 5 relations.



III) - Entre SQL et CYPHER:

Remarque : l’ensemble de données n’est pas très volumineux, donc les différences de temps d’exécution entre SQL et Cypher peuvent ne pas être très marquées. Cependant, certaines requêtes présentent une nette amélioration du temps d’exécution d’un système par rapport à l’autre.

1- Requête Cypher plus efficace qu’une requête SQL (Celui de UNWIND et COLLECT)

```
northwind=# EXPLAIN ANALYZE
northwind-# SELECT c.companyName, COUNT(DISTINCT p.productID) AS numberOfProductsPurchased
northwind-# FROM customers c
northwind-# JOIN orders o ON c.customerID = o.CustomerID
northwind-# JOIN order_details od ON o.orderID = od.orderID
northwind-# JOIN products p ON od.productID = p.productID
northwind-# GROUP BY c.companyName
northwind-# ORDER BY numberOfProductsPurchased DESC;
                                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------------------------
 Sort  (cost=232.94..233.17 rows=91 width=28) (actual time=23.687..23.707 rows=89 loops=1)
   Sort Key: (count(DISTINCT p.productid)) DESC
   Sort Method: quicksort  Memory: 29kB
   ->  GroupAggregate  (cost=212.90..229.98 rows=91 width=28) (actual time=22.281..23.620 rows=89 loops=1)
         Group Key: c.companyname
         ->  Sort  (cost=212.90..218.29 rows=2155 width=24) (actual time=22.256..22.533 rows=2155 loops=1)
               Sort Key: c.companyname, p.productid
               Sort Method: quicksort  Memory: 204kB
               ->  Hash Join  (cost=40.45..93.59 rows=2155 width=24) (actual time=1.150..6.877 rows=2155 loops=1)
                     Hash Cond: (od.productid = p.productid)
                     ->  Hash Join  (cost=37.72..84.89 rows=2155 width=24) (actual time=1.020..5.351 rows=2155 loops=1)
                           Hash Cond: (o.customerid = c.customerid)
                           ->  Hash Join  (cost=33.67..74.92 rows=2155 width=10) (actual time=0.848..3.208 rows=2155 loops=1)
                                 Hash Cond: (od.orderid = o.orderid)
                                 ->  Seq Scan on order_details od  (cost=0.00..35.55 rows=2155 width=8) (actual time=0.025..0.610 rows=2155 loops=1)
                                 ->  Hash  (cost=23.30..23.30 rows=830 width=10) (actual time=0.794..0.795 rows=830 loops=1)
                                       Buckets: 1024  Batches: 1  Memory Usage: 44kB
                                       ->  Seq Scan on orders o  (cost=0.00..23.30 rows=830 width=10) (actual time=0.027..0.412 rows=830 loops=1)
                           ->  Hash  (cost=2.91..2.91 rows=91 width=26) (actual time=0.145..0.146 rows=91 loops=1)
                                 Buckets: 1024  Batches: 1  Memory Usage: 14kB
                                 ->  Seq Scan on customers c  (cost=0.00..2.91 rows=91 width=26) (actual time=0.030..0.070 rows=91 loops=1)
                     ->  Hash  (cost=1.77..1.77 rows=77 width=4) (actual time=0.103..0.104 rows=77 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 11kB
                           ->  Seq Scan on products p  (cost=0.00..1.77 rows=77 width=4) (actual time=0.045..0.067 rows=77 loops=1)
 Planning Time: 1.471 ms
 Execution Time: 24.032 ms
(26 rows)
```

temps d'exécution SQL : 24ms

```
1  MATCH (customer:Customer)-[:PURCHASED]→(order:Order)-[pu:PRODUCT]→(product:Product)
2  WITH customer, order, COLLECT(product) AS purchasedProducts
3  UNWIND purchasedProducts AS purchasedProduct
4  RETURN customer.companyName, COUNT(DISTINCT purchasedProduct) AS numberOfProductsPurchased
5  ORDER BY numberOfProductsPurchased DESC;
```

Table | Text | Code

	customer.companyName	numberOfProductsPurchased
1	"Ernst Handel"	56
2	"Save-a-lot Markets"	53
3	"QUICK-Stop"	49
4	"Rattlesnake Canyon Grocery"	45
5	"Berglunds snabbköp"	37
6	"Hungry Owl All-Night Grocers"	36
7		

Started streaming 89 records in less than 1 ms and completed after 13 ms.

temps d'exécution cypher : 13 ms

—-> la requête cypher est 46% plus efficace

2-Requête SQL plus efficace qu'une requête Cypher:(Celui de OPTIONAL MATCH)

temps d'exécution SQL : 1.592 ms

```
northwind=# EXPLAIN ANALYZE
northwind-# SELECT e.employeeID, e.ReportsTo AS managerID, COUNT(o.OrderID) AS totalSales
northwind-# FROM employees e
northwind-# LEFT JOIN orders o ON e.employeeID = o.EmployeeID
northwind-# WHERE e.ReportsTo IS NOT NULL
northwind-# GROUP BY e.employeeID, e.ReportsTo;
                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=41.13..41.73 rows=60 width=16) (actual time=1.449..1.457 rows=8 loops=1)
   Group Key: e.employeeid
   Batches: 1  Memory Usage: 24kB
   ->  Hash Right Join  (cost=11.35..36.98 rows=830 width=12) (actual time=0.139..1.059 rows=734 loops=1)
         Hash Cond: (o.employeeid = e.employeeid)
         ->  Seq Scan on orders o  (cost=0.00..23.30 rows=830 width=8) (actual time=0.028..0.291 rows=830 loops=1)
         ->  Hash  (cost=10.60..10.60 rows=60 width=8) (actual time=0.083..0.084 rows=8 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 9kB
               ->  Seq Scan on employees e  (cost=0.00..10.60 rows=60 width=8) (actual time=0.053..0.064 rows=8 loops=1)
                     Filter: (reportsto IS NOT NULL)
                     Rows Removed by Filter: 1
 Planning Time: 0.876 ms
 Execution Time: 1.592 ms
```

temps d'exécution Cypher : 4ms

```
1  MATCH (e:Employee)-[:REPORTS_TO]→(manager:Employee)
2  OPTIONAL MATCH (e)-[:SOLD]→(o:Order)
3  RETURN e.employeeID, manager.employeeID, COUNT(o) AS totalSales
4  order by totalSales;
5
```

Table | Text | Code

	e.employeeID	manager.employeeID	totalSales
1	"5"	"2"	42
2	"9"	"5"	43
3	"6"	"5"	67
4	"7"	"5"	72
5	"8"	"2"	104
6	"1"	"2"	123
7			

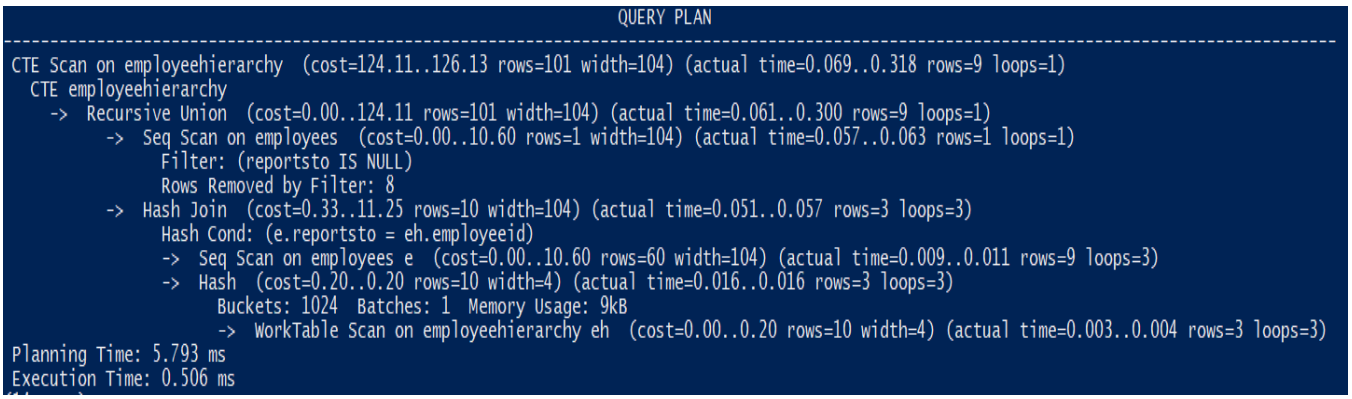
Started streaming 8 records after 1 ms and completed after 4 ms.

la requête SQL est 60% plus efficace

3- Requête récursive SQL avec une requête cypher équivalente :

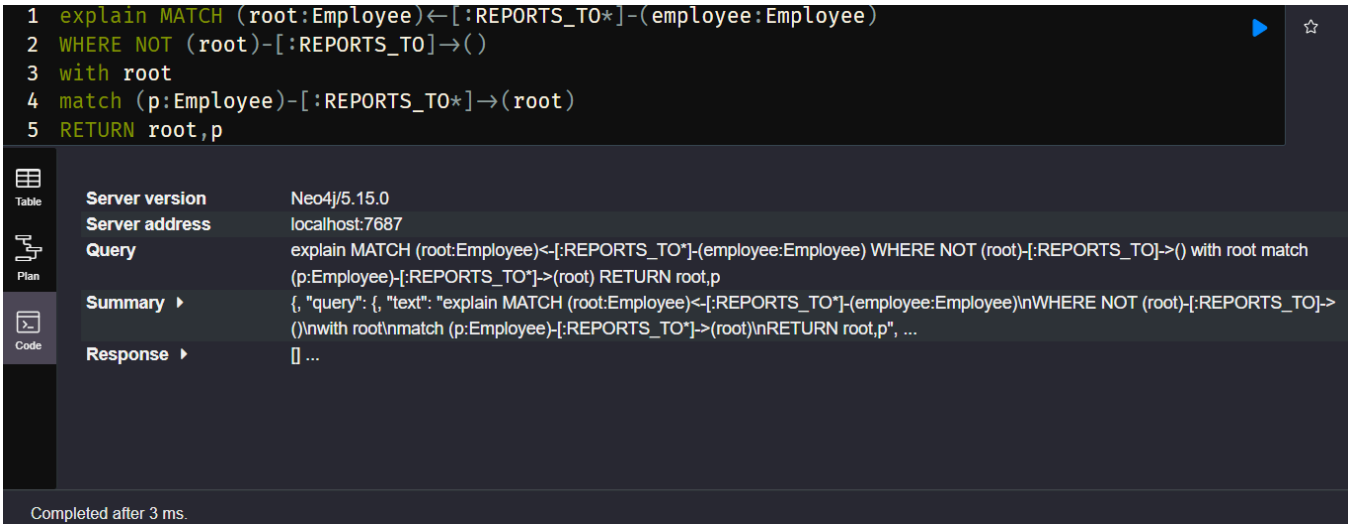
en SQL :

```
WITH RECURSIVE EmployeeHierarchy AS (
    SELECT EmployeeID, LastName, FirstName, ReportsTo
    FROM employees
    WHERE ReportsTo IS NULL

    UNION ALL

    SELECT e.EmployeeID, e.LastName, e.FirstName, e.ReportsTo
    FROM employees e
    JOIN EmployeeHierarchy eh ON e.ReportsTo = eh.EmployeeID
)
SELECT * FROM EmployeeHierarchy;
```



temps d’execution SQL : 0.5ms

en Cypher:



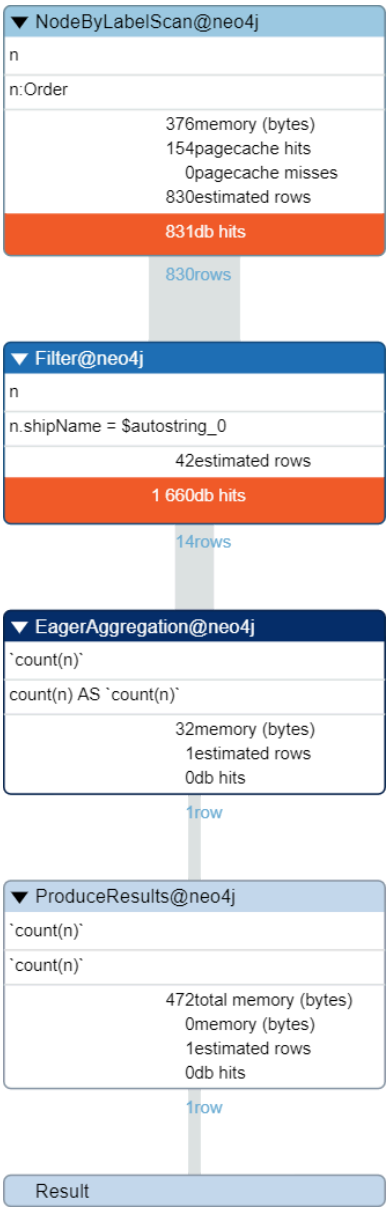
temps d'execution Cypher: 3ms

Remarque : Il est important de noter que la comparaison entre SQL et Cypher peut ne pas être pertinente dans ce contexte, étant donné la petite taille de la table ' employees '. Bien que les résultats montrent que la requête SQL récursive est plus performante que son équivalent en Cypher pour cette taille de données, mais en général : **les requêtes Cypher surpassent les requêtes SQL récursives sur des ensembles de données plus importants.**

IV) Plans d’exécution :

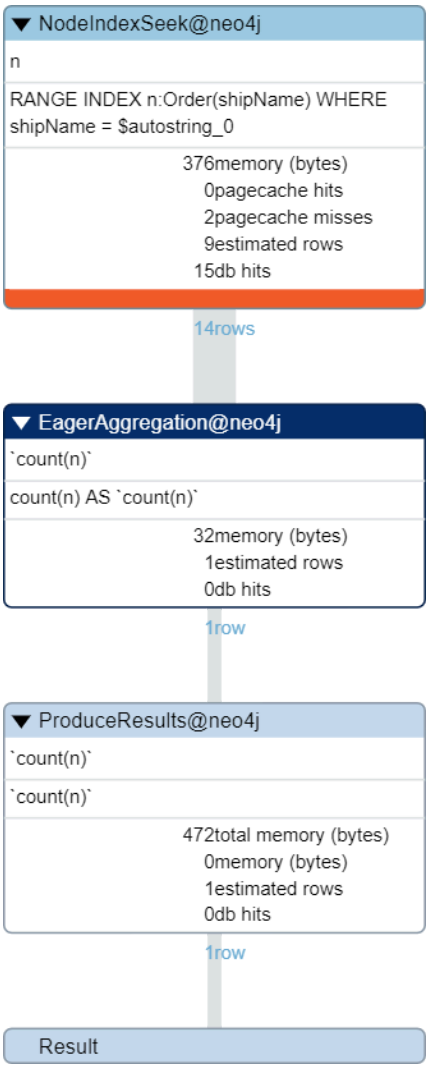
PROFILE : Afficher le plan d'exécution sans exécuter la requête.

```
1  PROFILE
2  MATCH (n:Order)
3  WHERE n.shipName = "Hanari Carnes"
4  RETURN count(n)
```



Création d’un index (dans le fichier index.cypher)

```
CREATE INDEX order_index FOR (n:Order) ON (n.shipName)
```



V) Analytique de graphe

PageRank

L'algorithme PageRank évalue la pertinence de chaque nœud dans le graphe en tenant compte du nombre de liens entrants et de l'importance des nœuds source associés. Son principe repose sur l'idée qu'un nœud est d'autant plus important que des nœuds importants pointent vers lui. Dans notre contexte, l'algorithme PageRank a été utilisé pour classer les employés en fonction du nombre de liens hiérarchiques (représentés par la relation 'REPORTS_TO') dans lesquels ils sont impliqués, ainsi que de l'importance associée à chaque relation hiérarchique.

Création de graph:(dans le fichier EmployeeGraph.cypher)

```
CALL gds.graph.project(
  'EmployeeGraph',
  ['Employee'],
  'REPORTS_TO'

);
```

Table | Text | Code

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	projectMillis
{ "Employee": { "label": "Employee", "properties": { } } }	{ "REPORTS_TO": { "aggregation": "DEFAULT", "orientation": "NATURAL", "indexInverse": false, "properties": { }, "type": "REPORTS_TO" } }	"EmployeeGraph"	9	8	622

En mode d'exécution stream, l'algorithme renvoie le score pour chaque nœud:

```
1 //Run PageRank algorithm on 'Employee' nodes connected by 'REPORTS_TO'
2 CALL gds.pageRank.stream('EmployeeGraph')
3 YIELD nodeId, score
4
5 //Retrieve employeeID and score for each 'Employee' node
6 WITH gds.util.asNode(nodeId) AS employee, score
7 RETURN employee.employeeID AS employeeID, score
8 ORDER BY score DESC, employeeID ASC;
```

Table | Text | Code

	employeeID	score
1	"2"	1.1126250000000002
2	"5"	0.5325000000000001
3	"1"	0.15000000000000002
4	"3"	0.15000000000000002
5	"4"	0.15000000000000002
6	"6"	0.15000000000000002

En mode d'exécution stats (statistiques), l'algorithme renvoie une seule ligne contenant un résumé du résultat de l'algorithme:

```
CALL gds.pageRank.stats('EmployeeGraph', {
maxIterations: 20,
dampingFactor: 0.85
})
YIELD centralityDistribution
RETURN centralityDistribution AS stats
```

Table | Text | Code

stats


```
{
  "min": 0.14999961853027344,
  "max": 1.112624168395996,
  "p90": 1.112624168395996,
  "p999": 1.112624168395996,
  "p99": 1.112624168395996,
  "p50": 0.14999961853027344,
  "p75": 0.14999961853027344,
  "p95": 1.112624168395996,
  "mean": 0.2994575500488281
}
```