CHENAL Louka
MEDJDOUB Assia

Bases de Données Spécialisée

comparaison Cypher 5 et Cypher 25

Université Paris Cité – Master DATA

Table de matiere

1. Présentation du jeu de données

1.1 Source du dataset

Nous utilisons le jeu de données **Supply Chain Dataset** disponible sur Kaggle :
https://www.kaggle.com/datasets/amirmotefaker/supply-chain-dataset

Ce dataset décrit une chaîne logistique complète, allant de la production chez les fournisseurs jusqu'à la livraison aux clients, en passant par le transport et l'inspection qualité.
Il est particulièrement adapté à une modélisation graphique, car les entités sont fortement connectées et les relations portent des propriétés numériques (coûts, délais, volumes).

1.2 Description des colonnes

Catégorie	Colonne	Description
Produits	SKU	Identifiant unique du produit
	Product type	Catégorie/type du produit
	Price	Prix du produit
	Availability	Quantité disponible en stock
	Number of products sold	Nombre d'unités vendues
	Revenue generated	Revenu généré par le produit
Fournisseurs	Supplier name	Nom du fournisseur
	Location	Localisation du fournisseur
	Lead time	Délai de production
	Production volumes	Volume de production
	Manufacturing costs	Coût de fabrication
Transport & Logistique	Shipping carriers	Transporteur
	Shipping times	Temps de transport
	Shipping costs	Coût de transport
	Transportation modes	Mode de transport (air, mer, route…)
	Routes	Itinéraire logistique
	Costs	Coût de la route
Clients	Customer demographics	Type/catégorie de client

	Order quantities	Quantité commandée
Qualité	Inspection results	Résultat de l'inspection
	Defect rates	Taux de défaut

2. Analyse exploratoire et nettoyage des données

Avant l'import dans Neo4j, une **analyse exploratoire en Python (pandas)** a été réalisée afin de :

- Vérifier l'unicité de la clé produit (SKU)
- Détecter les valeurs manquantes
- Identifier les colonnes catégorielles
- Vérifier la cohérence des types de données

2.1 Modélisation de données :

Les données de la chaîne logistique ont été modélisées dans Neo4j sous la forme d'un graphe de propriétés (property graph).
Les principales entités du jeu de données ont été représentées par des nœuds, tandis que les interactions entre ces entités ont été modélisées par des relations orientées.

Nœuds:

- **Product** : représente un produit identifié de manière unique par son SKU, avec des propriétés telles que le type de produit, le prix, la disponibilité, le nombre d'unités vendues et le revenu généré.
- **Supplier** : représente un fournisseur, caractérisé par son nom et sa localisation.
- **Client** : représente un client, décrit par sa catégorie démographique.
- **Carrier** : représente un transporteur chargé de la livraison des produits.
- **Route** : représente un itinéraire logistique emprunté lors du transport.
- **Location** : représente une localisation géographique (ville).
- **Inspection** : représente le résultat d'un contrôle qualité, incluant le taux de défaut.

Ce choix de modélisation permet de représenter explicitement les différents acteurs de la chaîne logistique et leurs interactions, ce qui est particulièrement adapté à une base de données orientée graphe.

Relations et propriétés :

Les relations modélisent les interactions entre les entités de la chaîne logistique et portent des propriétés numériques, conformément aux exigences du sujet. Les principales relations sont les suivantes :

- FOURNIT entre un fournisseur et un produit, avec des propriétés telles que le délai de production, le volume de production et le coût de fabrication.
-
- SITUE_A entre un fournisseur et une localisation.
-
- EXPEDIE entre un produit et un transporteur, avec des propriétés comme le temps de transport, le coût de transport et le mode de transport.
-
- LIVRE entre un transporteur et un client, avec la quantité commandée.
-
- EMPRUNTE entre un transporteur et un itinéraire, avec le coût de la route.
-
- A_RESULTAT_INSPECTION entre un produit et une inspection qualité.

Les propriétés numériques (coûts, délais, volumes, quantités) ont volontairement été placées sur les relations, car elles décrivent des interactions entre entités plutôt que des caractéristiques intrinsèques des entités elles-mêmes.

Contraintes d'intégrité

Afin de garantir la cohérence des données et d'éviter les duplications lors de l'import, plusieurs contraintes d'intégrité ont été définies dans Neo4j :

- Une **clé de nœud (NODE KEY)** a été définie sur l'attribut SKU des produits, qui constitue une véritable clé métier unique.

- Des **contraintes d'unicité** ont été appliquées sur :

    - le nom des fournisseurs,

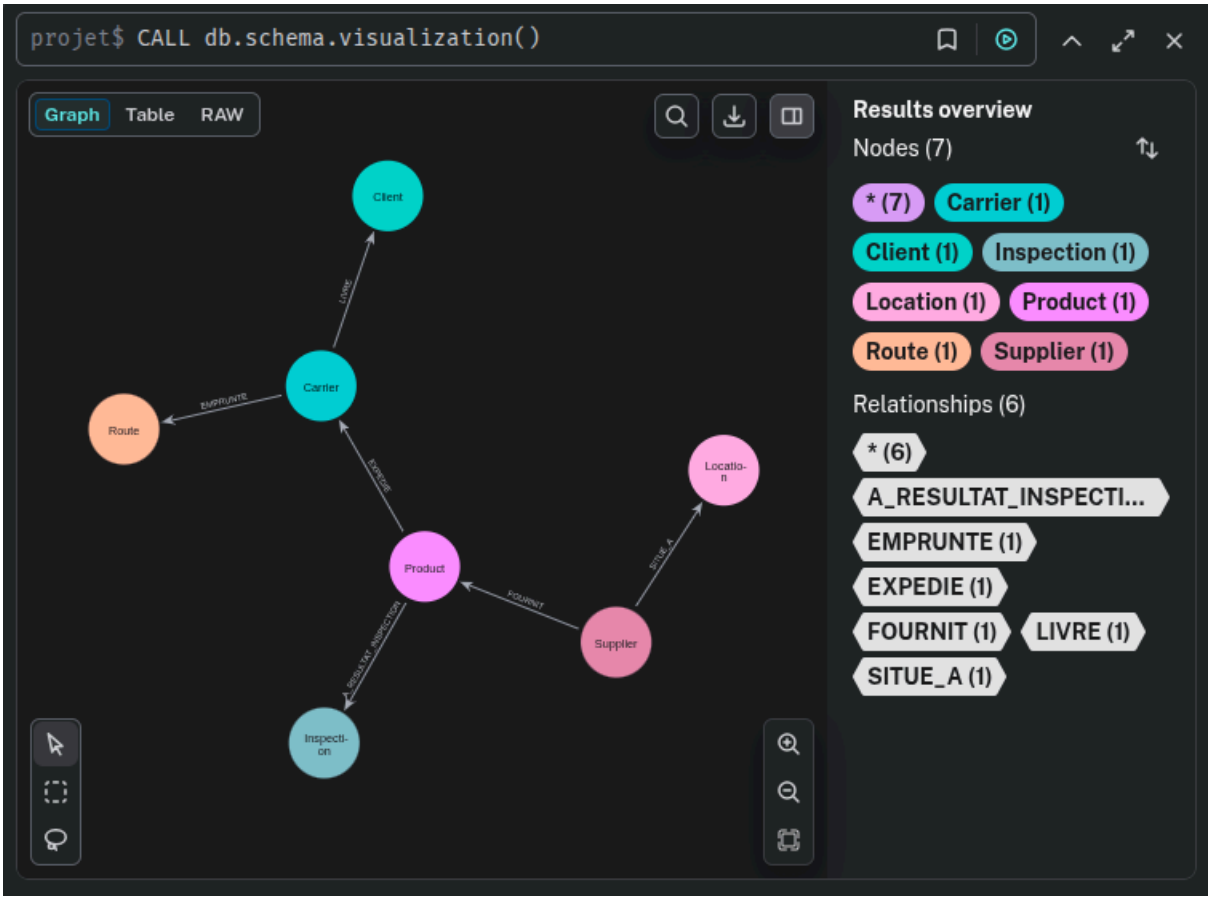
    - le nom des transporteurs,

    - le nom des itinéraires,

    - la ville associée aux localisations.

Ces contraintes permettent d'assurer l'unicité des entités principales du graphe et créent implicitement des index, améliorant ainsi les performances des opérations de recherche et de fusion.

Présentation de schéma



3.cypher 5 vs cypher 25

3.1 Scénario : chemins à coûts croissants

Dans une chaîne logistique, un produit peut passer par plusieurs étapes de transport avant d'atteindre le client final. Chaque relation entre les entités (produit, transporteur, route) possède une **propriété numérique**, ici le **coût de transport**.

L'objectif de ce scénario est d'identifier des chemins logistiques **où le coût augmente à chaque étape**, afin de détecter des anomalies ou des inefficacités dans la chaîne.

Ce type de requête met en évidence **l'intérêt des bases graphe** : en SQL, il serait très complexe de raisonner sur un nombre variable d'étapes et de vérifier des conditions entre relations consécutives. En Neo4j, les chemins et les relations sont des objets de première classe, ce qui rend ce type de raisonnement naturel.

On considère un chemin typique :

(Product)-[:EXPEDIE]->(Carrier)-[:EMPRUNTE]->(Route)

Chaque relation a un coût (**shipping_cost** ou **route_cost**)

On souhaite filtrer uniquement les chemins où le coût croît strictement à chaque étape

Même si, dans notre dataset, tous les chemins existants respectent déjà cette contrainte, ce scénario illustre parfaitement l'utilisation de fonctions avancées de Cypher pour raisonner sur des propriétés d'arêtes le long d'un chemin.

1. Requête Cypher 5 (NOT EXISTS)

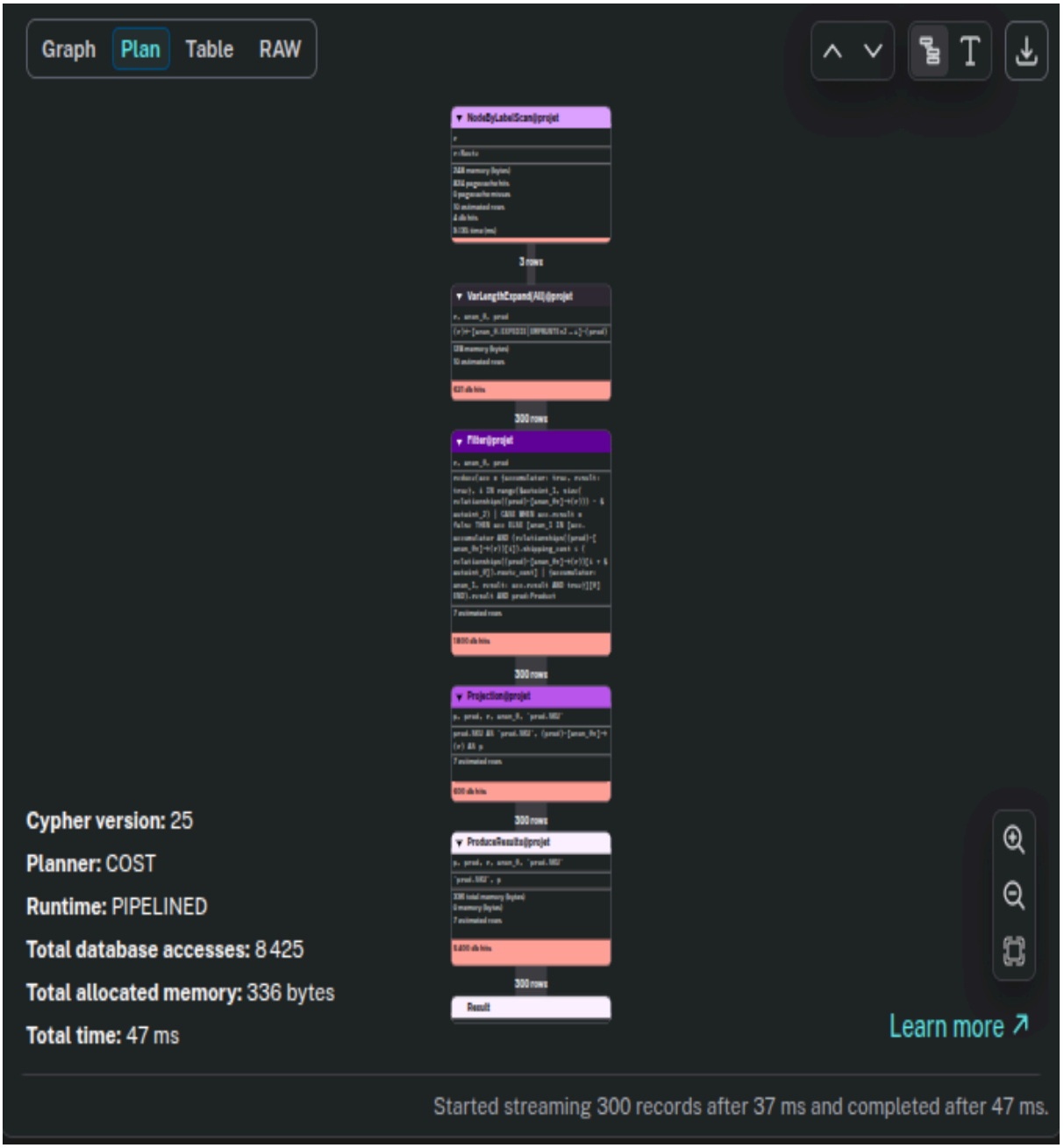
En Cypher 5, la contrainte de croissance est exprimée à l'aide d'un filtre négatif. La requête exclut tous les chemins contenant une paire de relations consécutives qui ne respecte pas la condition croissante :

```
PROFILE
MATCH p = (prod:Product)-[:EXPEDIE|EMPRUNTE*2..4]->
(r:Route)
WHERE NOT EXISTS {
  MATCH (a)-[r1]->(b)-[r2]->(c)
  WHERE r1 IN relationships(p)
    AND r2 IN relationships(p)
    AND r1.shipping_cost >= r2.route_cost
}
RETURN prod.SKU, p;
```

2. Requête Cypher 25 (allReduce)

En Cypher 25, la même logique est exprimée de manière plus déclarative et lisible grâce à allReduce :

```
CYPHER 25
PROFILE
MATCH p = (prod:Product)-[:EXPEDIE|EMPRUNTE*2..4]->
(r:Route)
WHERE allReduce(
    acc = true,
    i IN range(0, size(relationships(p)) - 2) |
        acc AND relationships(p)[i].shipping_cost <
relationships(p)[i + 1].route_cost,
    true
)
RETURN prod.SKU, p;
```



3.comparaison

Bien que les requêtes Cypher 5 et Cypher 25 permettent d'exprimer la même contrainte logique, à savoir la croissance stricte du coût le long d'un chemin logistique, leurs approches et leurs plans d'exécution diffèrent significativement.

En Cypher 5, la condition est formulée de manière indirecte à l'aide de filtres négatifs (NOT EXISTS), ce qui conduit à un plan d'exécution plus profond et plus complexe, composé de plusieurs opérateurs intermédiaires chargés d'exclure les chemins non valides.

Cette décomposition de la logique entraîne un nombre élevé d'accès à la base et rend le plan moins lisible. À l'inverse, Cypher 25 adopte une approche plus déclarative grâce à la fonction allReduce, qui permet de raisonner directement sur l'ensemble des relations du chemin et de

vérifier la contrainte de croissance en une seule étape logique. Cette expressivité se traduit par un plan d'exécution plus compact, plus lisible et plus efficace, avec moins d'opérateurs et un nombre réduit d'accès à la base.

Cette comparaison met en évidence l'évolution de Cypher vers un langage plus adapté au raisonnement global sur les chemins, exploitant pleinement les spécificités des bases de données graphe.

3.2 Scénario : trouver le plus court chemin fournisseur → client

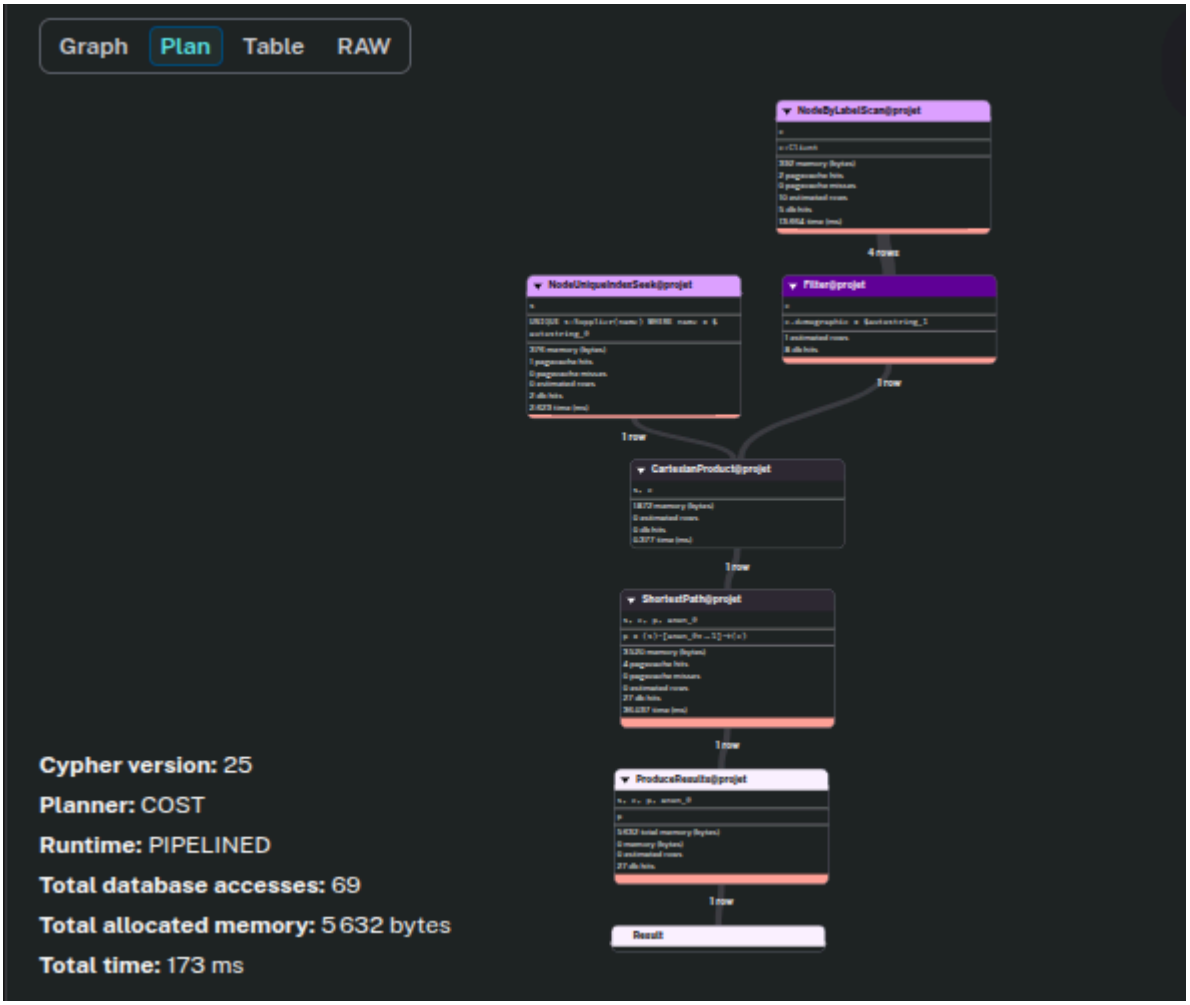
Dans une chaîne logistique, il est souvent nécessaire de déterminer le **chemin le plus court** pour livrer un produit d'un fournisseur à un client. Ce chemin peut être mesuré selon :

1. **Le nombre d'étapes** (non pondéré) : le chemin avec le moins de relations entre le fournisseur et le client.

2. **Le coût total** (pondéré) : le chemin où la somme des coûts de production, transport et livraison est minimale.

1. Chemin non pondéré: Cypher 5 et Cypher 25 :

```
PROFILE
MATCH p = shortestPath(
  (s:Supplier {name:'Supplier 3'})-[*..5]->(c:Client
{demographic:'Non-binary'})
)
RETURN p;
```



La fonction **shortestPath** retourne le chemin comportant le moins de relations entre un fournisseur et un client, sans tenir compte des coûts des relations. Elle permet ainsi d'identifier rapidement une route possible dans le graphe. Dans ce cas simple, la syntaxe est identique en Cypher 5 et Cypher 25, les différences apparaissent surtout sur des requêtes plus complexes ou utilisant des fonctions avancées.

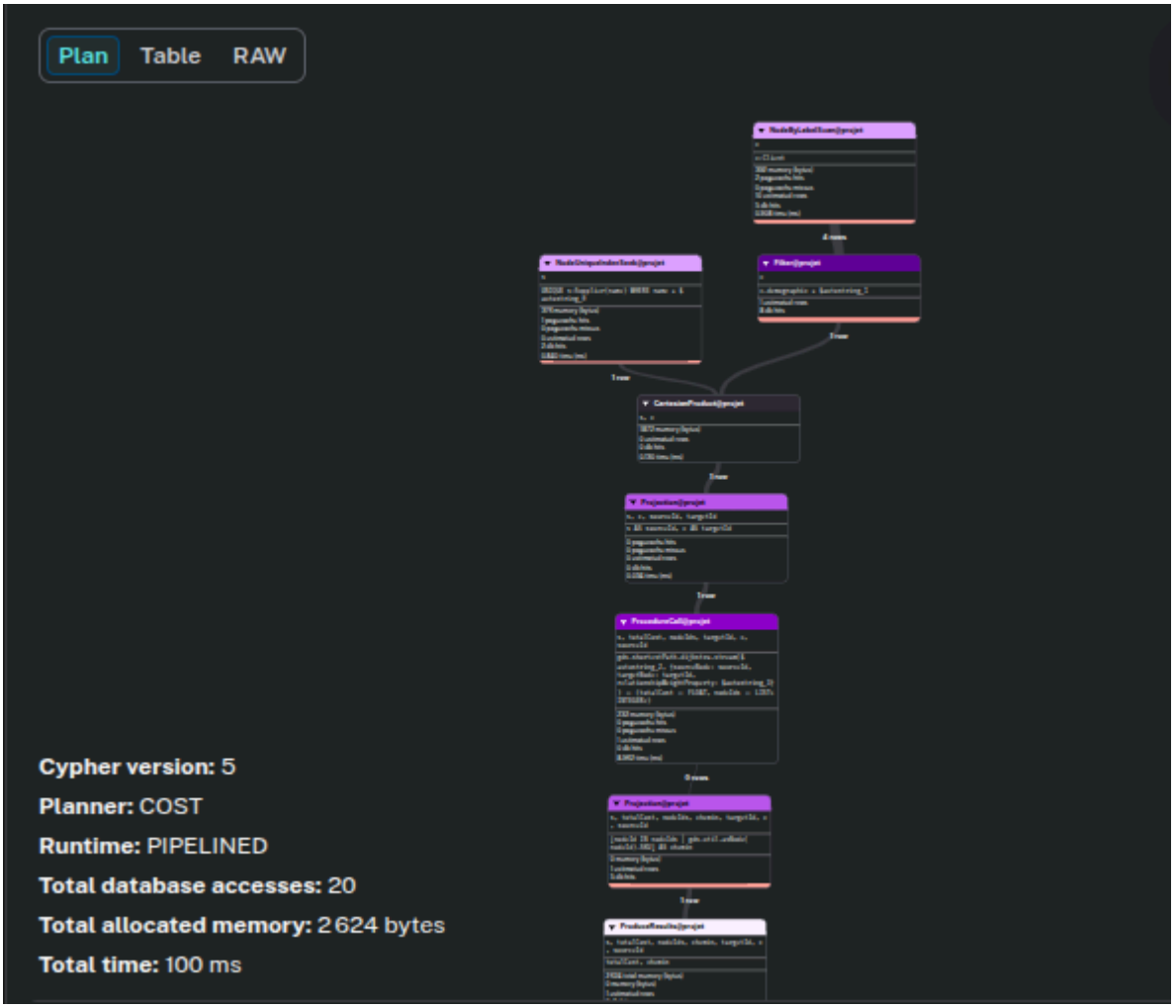
2.Chemin pondéré avec GDS (Dijkstra)

Pour calculer le chemin ayant le coût total minimal, nous avons utilisé l'algorithme de Dijkstra de la bibliothèque Graph Data Science (GDS).

Avant la projection du graphe, une propriété cost a été ajoutée aux relations afin de représenter les coûts logistiques : le lead_time pour les relations FOURNIT, le shipping_time pour EXPEDIE, et un coût fixe de 1 pour LIVRE.

Le graphe pondéré est ensuite projeté en mémoire avec gds.graph.project, ce qui permet d'exécuter l'algorithme efficacement.

L'appel à gds.shortestPath.dijkstra.stream calcule alors le chemin entre le fournisseur et le client en minimisant la somme des coûts, et retourne à la fois le coût total et le chemin suivi.



3.comparaison

L'analyse des plans d'exécution met en évidence des différences nettes entre le calcul du chemin non pondéré avec shortestPath et le chemin pondéré avec GDS (Dijkstra).

Dans le cas non pondéré, la requête est exécutée directement sur le graphe stocké en base, ce qui se traduit par un nombre réduit d'accès à la base de données, une faible consommation mémoire et un temps d'exécution plus court. Les plans montrent une exécution simple, sans phase de préparation supplémentaire.

À l'inverse, le calcul du chemin pondéré avec GDS nécessite plusieurs étapes visibles dans les plans : création des propriétés de coût, projection du graphe en mémoire, puis exécution de l'algorithme de Dijkstra.

Cette approche entraîne une consommation plus élevée de mémoire et un temps d'exécution supérieur, mais elle permet d'obtenir un résultat plus pertinent d'un point de vue métier, car le chemin retourné minimise le coût total et non le nombre d'étapes. Les plans d'exécution reflètent ainsi un compromis clair entre performance brute et qualité du résultat.

3.3 Scénario : chemins dont le coût total est inférieur à 200

Dans une chaîne logistique, il peut être nécessaire d'identifier tous les chemins reliant un fournisseur à un client dont le coût total cumulé reste inférieur à un seuil donné, ici 200. Chaque relation du chemin possède plusieurs propriétés de coût (manufacturing_cost, shipping_cost, route_cost) qui contribuent au coût global du transport.

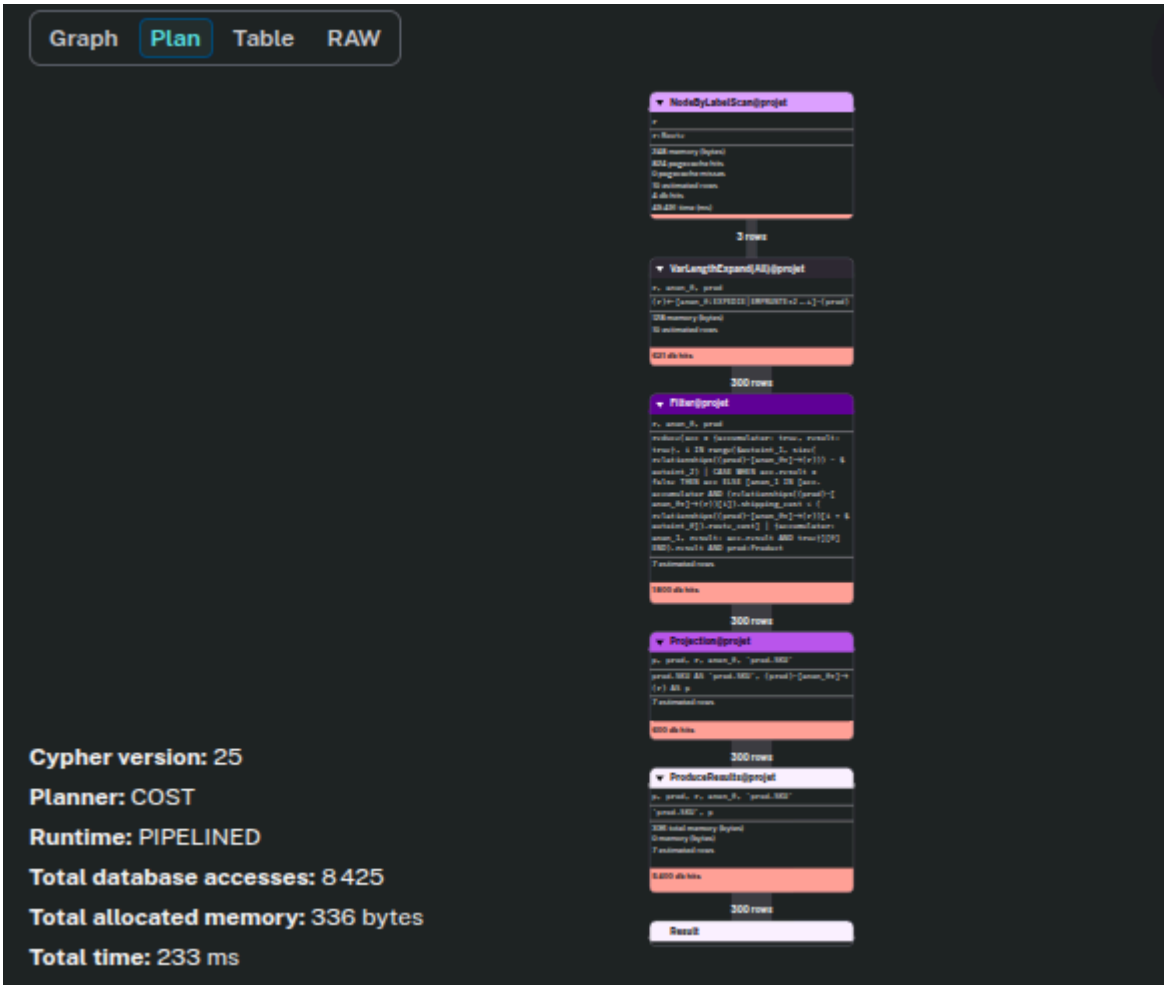
L'objectif de ce scénario est de filtrer uniquement les chemins valides en fonction de cette contrainte globale. Ce type de raisonnement, impliquant une agrégation sur l'ensemble des relations d'un chemin de longueur variable, illustre l'intérêt des bases de données graphe et des fonctions avancées de Cypher.

1. Requête Cypher 5 (NOT EXISTS)

En Cypher 5, la contrainte de coût est exprimée de manière indirecte à l'aide de NOT EXISTS. La requête consiste à exclure tous les chemins pour lesquels la somme des coûts dépasse 200.
Cette approche nécessite de parcourir à nouveau les chemins afin d'identifier les cas invalides, ce qui conduit à une logique plus complexe et à un plan d'exécution plus profond, avec davantage d'opérateurs intermédiaires et d'accès à la base.

2. Requête Cypher 25 (allReduce)

En Cypher 25, la même contrainte est exprimée de manière plus directe grâce à la fonction allReduce. Celle-ci permet de calculer le coût total directement lors du parcours des relations du chemin et de vérifier la condition cost < 200 en une seule expression.
Cette écriture est plus lisible, plus concise et correspond mieux à un raisonnement global sur le chemin.



3. Comparaison basée sur les plans d'exécution

Bien que les deux requêtes permettent d'obtenir le même résultat fonctionnel, leurs plans d'exécution diffèrent nettement.
La version Cypher 5, basée sur NOT EXISTS, génère un plan plus complexe, avec un nombre élevé d'opérateurs et d'accès à la base, traduisant un double parcours des chemins.
À l'inverse, Cypher 25 regroupe la logique dans une seule opération grâce à allReduce, ce qui se traduit par un plan plus compact, plus lisible et plus efficace en termes de ressources.

Ce scénario met en évidence l'évolution de Cypher vers un langage plus expressif et mieux adapté au calcul de contraintes globales sur les chemins dans un graphe.

4.GDS vs cypher 25

Cypher 25 permet d’approcher certains algorithmes de graphe, mais ne peut se substituer aux implémentations dédiées de la bibliothèque Graph Data Science (GDS), tant en termes de performance que de garanties algorithmiques.

Plus précisément, Cypher 25 met à disposition des outils robustes pour formuler des contraintes globales sur les trajets (par exemple à travers allReduce) et pour restreindre l'étendue de la recherche pendant les explorations. Néanmoins, cela reste uniquement déclaratif et offre aucun contrôle direct sur les structures algorithmiques essentielles, comme les files de priorité par exemple.

À l’inverse, GDS fournit des implémentations algorithmiques dédiées, exécutées sur des graphes projetés en mémoire, avec des garanties formelles sur la correction et la complexité des algorithmes. Ces implémentations sont nettement plus performantes et mieux adaptées à des analyses globales ou intensives sur de grands graphes.

5.sql vs cypher 25

Le plan d’exécution de la requête SQL récursive met en évidence une explosion combinatoire sévère. Bien que la profondeur de récursion et le coût total soient explicitement bornés, PostgreSQL génère plus de deux millions de tuples intermédiaires avant de ne produire que cinq résultats valides. La majorité de ces tuples est éliminée par des conditions de filtrage appliquées tardivement, comme l’indique le nombre élevé de lignes supprimées par les filtres

```
                                                  QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=7562.50..7599.80 rows=5 width=472) (actual time=1975.279..1975.297 rows=5 loops=1)
   CTE supplychainpath
     ->  Recursive Union  (cost=3.41..6528.66 rows=133000 width=166) (actual time=0.082..1437.596 rows=2262100 loops=1)
           ->  Hash Join  (cost=3.41..11.86 rows=100 width=53) (actual time=0.079..0.194 rows=100 loops=1)
                 Hash Cond: ((sp.product_sku)::text = (p.sku)::text)
                 ->  Nested Loop  (cost=0.16..8.33 rows=100 width=15) (actual time=0.026..0.107 rows=100 loops=1)
                       ->  Seq Scan on supplier_product sp  (cost=0.00..2.00 rows=100 width=15) (actual time=0.008..0.019 rows=100 loops=1)
                       ->  Memoize  (cost=0.16..0.66 rows=1 width=4) (actual time=0.000..0.000 rows=1 loops=100)
                             Cache Key: sp.supplier_id
                             Cache Mode: logical
                             Hits: 95  Misses: 5  Evictions: 0  Overflows: 0  Memory Usage: 1kB
                             ->  Index Only Scan using supplier_pkey on supplier s_1  (cost=0.15..0.65 rows=1 width=4) (actual time=0.002..0.003 rows=1 loops=5)
                                   Index Cond: (id_supplier = sp.supplier_id)
                                   Heap Fetches: 5
                 ->  Hash  (cost=2.00..2.00 rows=100 width=5) (actual time=0.041..0.042 rows=100 loops=1)
                       Buckets: 1024  Batches: 1  Memory Usage: 12kB
                       ->  Seq Scan on product p  (cost=0.00..2.00 rows=100 width=5) (actual time=0.005..0.019 rows=100 loops=1)
           ->  Merge Right Join  (cost=35.09..518.68 rows=13290 width=166) (actual time=55.216..229.209 rows=452400 loops=5)
                 Merge Cond: (cr.carrier_id = cc.carrier_id)
                 ->  Index Scan using carrier_route_pkey on carrier_route cr  (cost=0.15..71.70 rows=1570 width=20) (actual time=0.009..0.021 rows=7 loops=5)
                 ->  Materialize  (cost=34.93..148.26 rows=1693 width=173) (actual time=54.212..127.168 rows=452398 loops=5)
                       ->  Merge Join  (cost=34.93..144.03 rows=1693 width=173) (actual time=54.207..78.942 rows=150800 loops=5)
                             Merge Cond: (cc.carrier_id = pc.carrier_id)
                             ->  Index Only Scan using carrier_client_pkey on carrier_client cc  (cost=0.15..78.75 rows=2040 width=8) (actual time=0.004..0.024 rows=10 loops=5)
                                   Heap Fetches: 49
                             ->  Sort  (cost=34.78..35.20 rows=166 width=169) (actual time=54.024..65.001 rows=150798 loops=5)
                                   Sort Key: pc.carrier_id
                                   Sort Method: quicksort  Memory: 25kB
                                   ->  Hash Join  (cost=3.25..28.66 rows=166 width=169) (actual time=21.845..37.626 rows=37700 loops=5)
                                         Hash Cond: ((scp_1.current_node)::text = (pc.product_sku)::text)
                                         ->  WorkTable Scan on supplychainpath scp_1  (cost=0.00..22.50 rows=333 width=158) (actual time=21.831..28.176 rows=37700 loops=5)
                                               Filter: (depth < 5)
                                               Rows Removed by Filter: 414720
                                         ->  Hash  (cost=2.00..2.00 rows=100 width=16) (actual time=0.044..0.045 rows=100 loops=1)
                                               Buckets: 1024  Batches: 1  Memory Usage: 14kB
                                               ->  Seq Scan on product_carrier pc  (cost=0.00..2.00 rows=100 width=16) (actual time=0.006..0.022 rows=100 loops=1)
   ->  Unique  (cost=1033.85..331740.96 rows=44333 width=472) (actual time=1975.277..1975.283 rows=5 loops=1)
         ->  Incremental Sort  (cost=1033.85..331297.63 rows=44333 width=472) (actual time=1975.275..1975.277 rows=5 loops=1)
               Sort Key: s.name, c.demographic, scp.total_cost, scp.depth
               Presorted Key: s.name
               Full-sort Groups: 1  Sort Method: quicksort  Average Memory: 27kB  Peak Memory: 27kB
               ->  Nested Loop  (cost=0.29..329160.10 rows=44333 width=472) (actual time=1.978..1975.216 rows=49 loops=1)
                     ->  Nested Loop  (cost=0.15..321799.72 rows=44333 width=258) (actual time=0.102..1975.114 rows=98 loops=1)
                           Join Filter: (s.id_supplier = scp.id_supplier)
                           Rows Removed by Join Filter: 207
                           ->  Index Scan using supplier_name_key on supplier s  (cost=0.15..52.95 rows=320 width=222) (actual time=0.011..0.015 rows=2 loops=1)
                           ->  CTE Scan on supplychainpath scp  (cost=0.00..2992.50 rows=44333 width=44) (actual time=0.056..987.528 rows=152 loops=2)
                                 Filter: (total_cost < '200'::numeric)
                                 Rows Removed by Filter: 1131388
                     ->  Index Scan using client_pkey on client c  (cost=0.15..0.17 rows=1 width=222) (actual time=0.001..0.001 rows=0 loops=98)
                           Index Cond: (id_client = scp.client_id)
 Planning Time: 1.013 ms
 Execution Time: 1996.516 ms
(53 rows)
```

À expressivité équivalente, Neo4j évalue plus efficacement ce type de requêtes. Cependant, lorsque la requête est purement tabulaire et ne nécessite pas d'exploration de chemins, PostgreSQL s'avère plus efficace que Neo4j, qui introduit un surcoût lié au modèle graphe, par exemple des agrégations simples ou des filtres sur des attributs.

Ces observations montrent que les deux approches sont complémentaires : Cypher 25 est particulièrement adapté aux requêtes impliquant des parcours complexes et des contraintes sur les chemins, tandis que le modèle relationnel reste plus efficace pour des traitements purement tabulaires.

conclusion

Ce projet a permis de mettre en évidence l'intérêt des bases de données orientées graphe pour l'analyse de chaînes logistiques complexes. La modélisation sous Neo4j facilite la représentation des entités et des interactions, en particulier lorsque les relations portent des propriétés numériques comme les coûts ou les délais.

La comparaison entre Cypher 5 et Cypher 25 montre une nette évolution du langage. Cypher 25 offre une écriture plus déclarative et plus lisible, notamment pour les requêtes impliquant des contraintes globales sur les chemins, ce qui se traduit par des plans d'exécution plus simples et plus efficaces.

Les expérimentations avec GDS et SQL confirment que chaque approche a ses avantages : GDS reste indispensable pour les algorithmes de graphe complexes et performants, tandis que Cypher 25 est particulièrement adapté aux explorations de chemins. Le modèle relationnel, quant à lui, demeure plus efficace pour les traitements purement tabulaires.