



# Les bases de données Graphe

## Plan

- Présentation des bases de données Graphe
- Le langage Cypher
- La modélisation



## Les bases de données orientées graphes

# Qu'est-ce qu'un graphe?

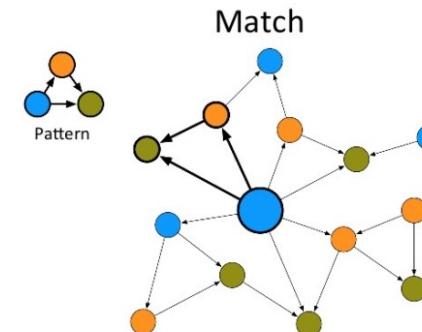


Un graphe est un ensemble de points/nœuds reliés entre eux par des arcs.

Les nœuds sont aussi importants que les relations qui les lient.

Les graphes sont partout :

- les réseaux sociaux : Facebook, Meetic ...
- le PageRank de Google
- les télécommunications
- Géo-spatial
- IA : la détection de fraude

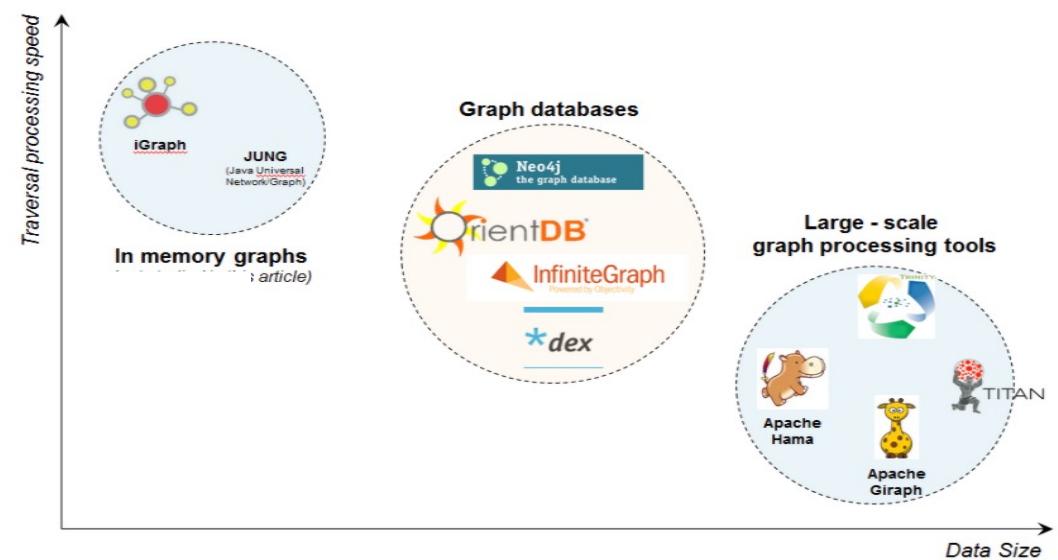
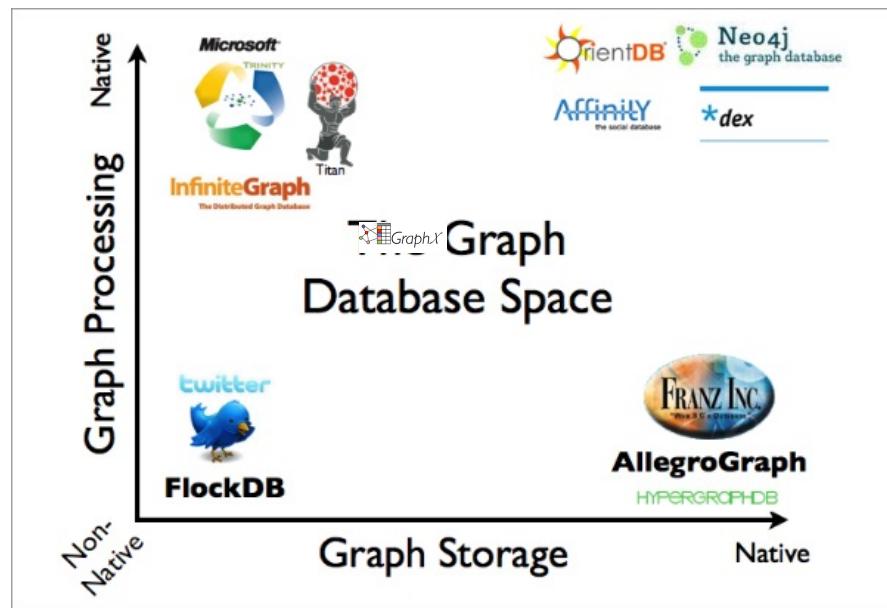


# Les bases graphes



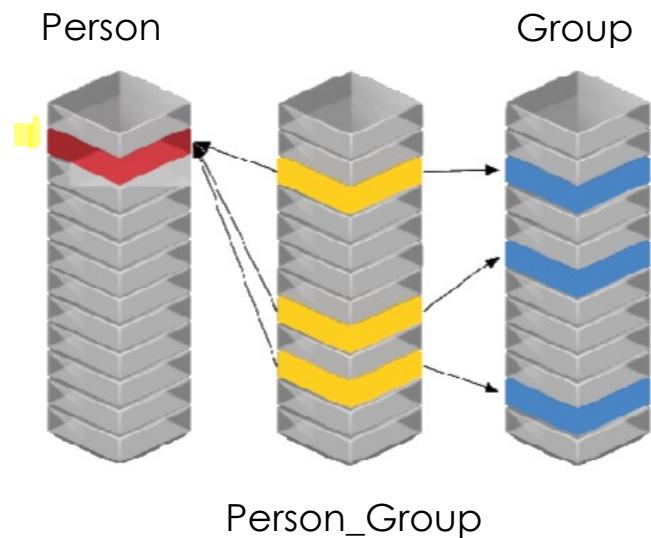
Différentes approches de bases de données graphe :

- Un moteur de traitement optimisé pour le parcours des graphes.
- Un stockage natif pour les données graphes.
- Un moteur de traitement et de stockage natif.

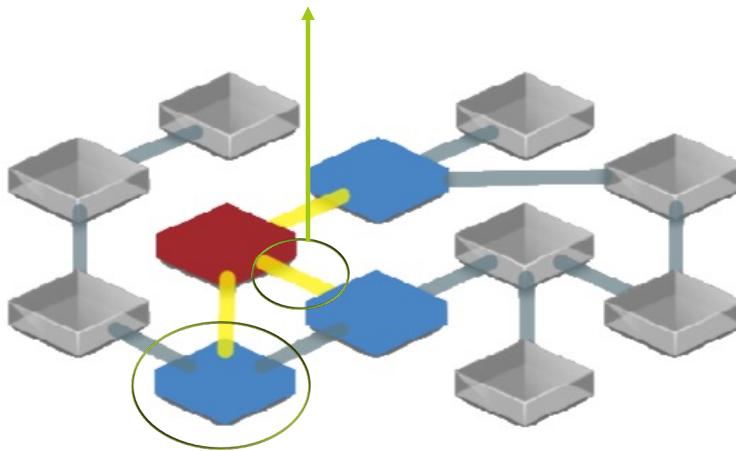


Classification des outils de traitement de graphes (source: Marko Rodriguez)

## Les bases graphes VS relationnelles



Relation : Person\_Group



Nœud : Label

Label :

Source : <http://blog.humancoders.com/interview-florent-biville-formateur-neo4j-pour-human-coders-formations-705/>

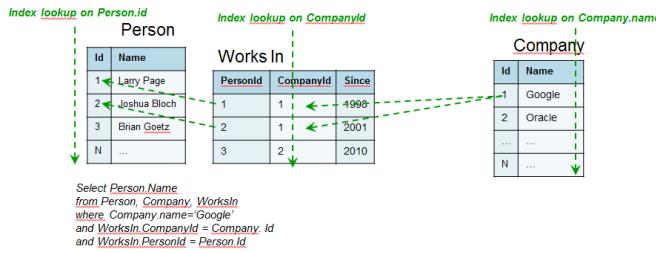


# Les bases graphes VS relationnelles

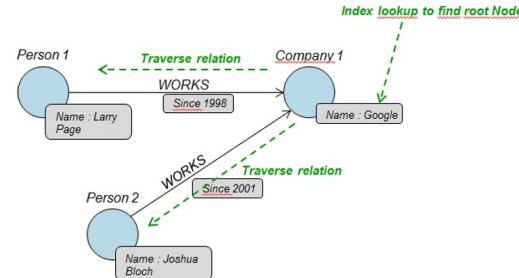


- Recherchons tous les employés de Google ?

SGDBR : Approche ensembliste



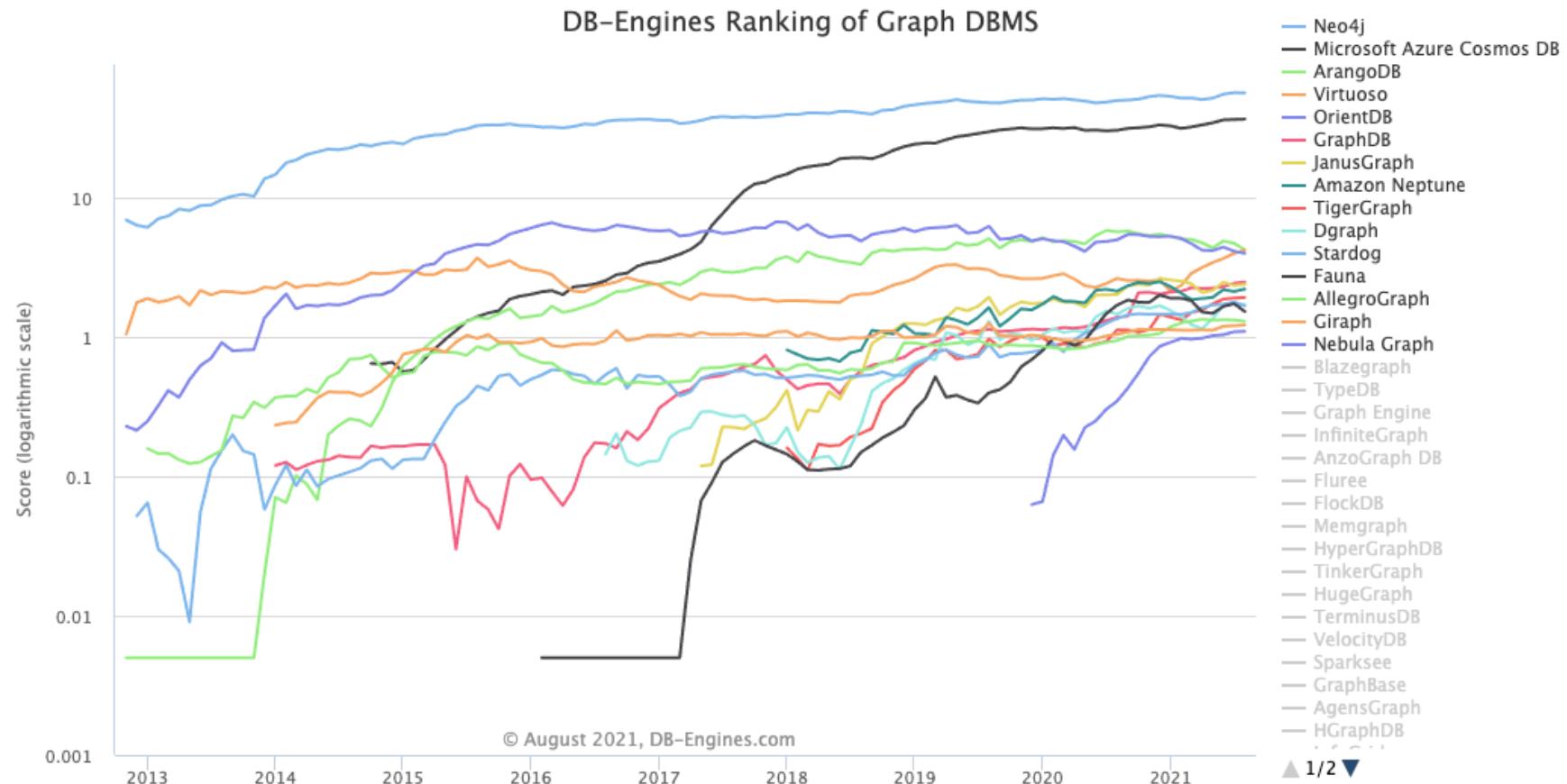
Graphe : Approche par relation



La normalisation accentue le nombre de tables et complexifie les accès aux données à l'inverse de l'approche graphe.

- Que se passe-t'il si on ajoute un nouvel employé à Google ?
- Le modèle graphe, une manière de présenter efficacement et simplement la réalité.

# Les bases graphes



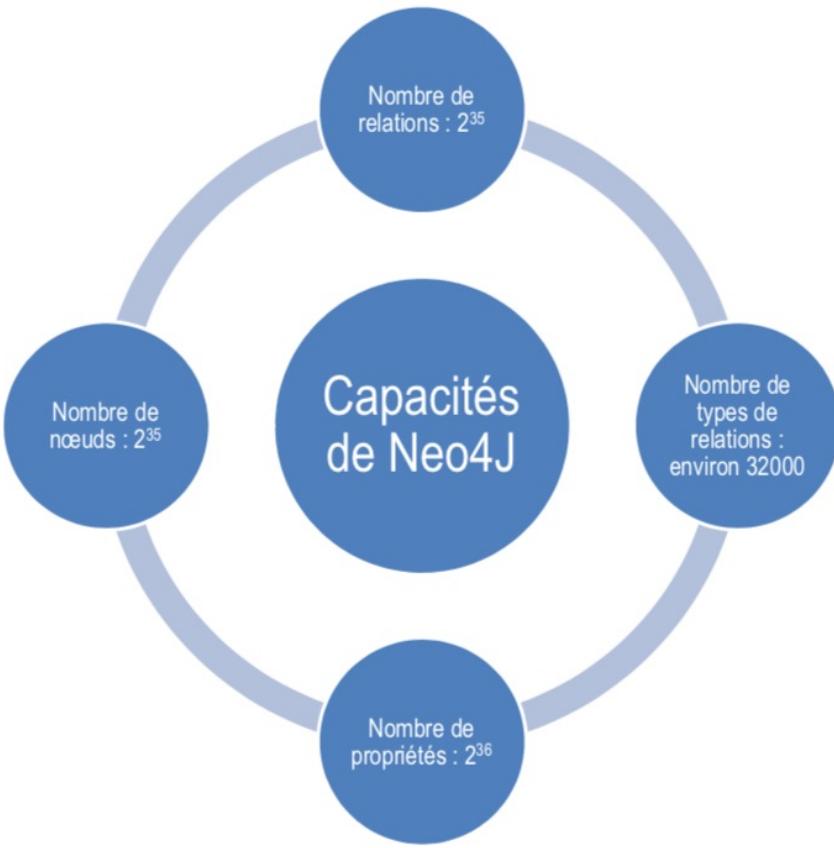


Créé en 2000, aujourd’hui NEO4J est la base de données graphe la plus populaire.

Les API disponibles (Bolt protocol) : Cypher, Python, JAVA, Go, JavaScript, PHP, Perl ...

A l’initiative de openCypher : langage orienté graphe basé sur du ASCII Art.







# Cypher



# SYNTAXE

Inspirée de:

- ASCII-Art:

```
(:Person)-[:LIVES_IN]->(:City)-[:PART_OF]->(:Country)
```

- SQL:

Cypher	SQL
MATCH	FROM ... JOIN ON ...
WHERE	WHERE
ORDER BY	ORDER BY
RETURN	SELECT



Pour effectuer une requête, les mots clés MATCH / WHERE et RETURN sont la base du langage.

# SYNTAXE

Les clauses les plus utilisés sont:

- **MATCH:** sélection de sous-graphes
- **WHERE:** contraintes additionnelles
- **RETURN:** sélection de ce qui est retourné





# SYNTAXE

Structure globale d'une requête:

```
MATCH <patterns>
WHERE <conditions>
RETURN <expressions>
```

Exemple:

```
MATCH (director:Person)-[:DIRECTED]->(movie)
WHERE director.name = "Steven Spielberg"
RETURN movie.title
```

Cypher se veut être un langage simple. La requête ci-dessus recherche les titres des films dont « Steven Spielberg est auteur ».



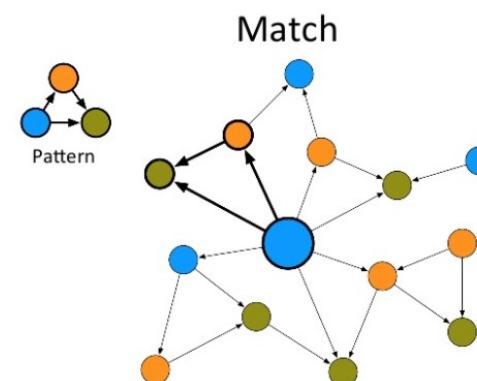


# MATCH

```
MATCH <patterns>
```

Sélectionne le(s) sous-graphe(s) concernés par la requête, en fonction des contraintes exprimées dans les <patterns>:

- contraintes structurelles
  - sur les noeuds
  - sur les relations
- contraintes simples (égalité) sur les propriétés (peuvent être exprimées dans la clause WHERE)



## Syntaxe Cypher : MATCH et les patterns

Le pattern s'exprime sous forme de () pour les nœuds et -[]- > pour les relations . Voyez le comme un moyen d'expliciter des contraintes de recherche sur un ensemble de nœuds ou sur un sous graphe.

Pattern	Description
MATCH (node) RETURN node	Le pattern (node) indique aucune contrainte de parcours. Le résultat affichera l'ensemble des nœuds et les relations qui les lient.
MATCH (node :Person) RETURN node	Ajoute une contrainte sur les nœuds dont le label est Person. L'ensemble des nœuds satisfaisant la contrainte sera conservé dans la variable node . Le résultat affichera uniquement les nœuds Person et les relations qui peuvent exister entre eux.
MATCH (node :Person { name : 'Steve Spielberg'}) RETURN node	Ajoute une contrainte sur la propriété name des nœuds disposant d'un label Person.
MATCH (node :Person { name : 'Steve Spielberg', birthDate : 1978}) RETURN node	Ajoute une contrainte sur les propriétés name et birthDate.

# Syntaxe Cypher : MATCH et les patterns



Pattern	Description
<pre>MATCH (p :Person) - - (m :Movie) RETURN p,m</pre>	Même si Neo4J nous impose lors de l'insertion des données à indiquer le sens de la relation, il est possible de remonter une relation à contre sens. Cette requête recherche tous les noeuds p avec un label Person directement connecté à un noeud de label Movie sans aucune contrainte sur le type de relation.
<pre>MATCH (p :Person) - -&gt; (m :Movie) RETURN p,m</pre>	Recherche tous les noeuds p avec un label Person connectés directement quelque soit le TYPE de relation aux noeuds Movie. La relation entre p et m. On retourne les ensembles p et m.
<pre>MATCH (p :Person) - [ a:ACTED_IN ]-&gt; (m :Movie) RETURN p,a, m</pre>	Ajoute une contrainte sur le TYPE de la relation. L'ensemble des relations sortantes de p sera conservé dans une variable a. On retourne les ensembles p,a et m.
<pre>MATCH (p :Person) - [ a:ACTED_IN {role : 'Neo' }]-&gt; (m :Movie) RETURN p,a, m</pre>	Ajoute une contrainte sur la priorité d'une relation.
<pre>MATCH (j:Person {name: 'Jennifer'})-[*]- (n) RETURN DISTINCT j, n</pre>	Retourne tous les noeuds reliés à "Jennifer" (directement ou indirectement), quelque soit leur distance en nombre de relations.
<pre>MATCH (j:Person {name: 'Jennifer'})-[*1..3]- (n) RETURN DISTINCT j, n</pre>	On peut définir la profondeur d'un chemin en précisant le nombre de relation à parcourir. Dans cet exemple, on retourne tous les noeuds à une distance de 3 relations maximum autour de "Jennifer".
<pre>MATCH (j:Person {name: 'Jennifer'})&lt;[:-PARENT_OF*]- (n) RETURN j, n</pre>	Retourne tous les descendants de "Jennifer".



## Syntaxe Cypher : MATCH et les patterns



```
MATCH (a:Actor)-[role:ACTED_IN {roles: ["Neo"] } ]->
      (matrix:Movie {title: "The Matrix" } )
RETURN a.name
```

Retourne le nom de l'acteur qui a joué le rôle de "Neo" dans le film "The Matrix".

```
MATCH (j:Person {name: 'Jennifer'})-[*1..3]-(n)
RETURN DISTINCT j, n
```

Retourne tous les noeuds à une distance de 3 relations maximum autour de "Jennifer".

```
MATCH (j:Person {name: 'Jennifer'})-[*]- (n)
RETURN DISTINCT j, n
```

Retourne tous les noeuds reliés à "Jennifer" (directement ou indirectement), quelque soit leur distance en nombre de relations.

```
MATCH (j:Person {name: 'Jennifer'})<-[ :PARENT_OF* ]- (n)
RETURN DISTINCT j, n
```

Retourne tous les descendants de "Jennifer".





# WHERE

```
WHERE <conditions>
```

Permet l'expression de contraintes plus complexes que celles de la clause MATCH:

- inégalité
- existence d'une propriété
- test sur valeur partielle
- test sur sous-graphe

(c'est l'équivalent de la clause WHERE du SQL)





# WHERE : ÉGALITÉ

```
MATCH (tom:Person)
WHERE tom.name = "Tom Hanks"
RETURN tom
```

```
MATCH (tom:Person {name:'Tom Hanks'})
RETURN tom
```





# WHERE : INÉGALITÉ

```
MATCH (p:Person)
WHERE NOT p.name = 'Tom Hanks'
RETURN p
```

```
MATCH (nineties:Movie)
WHERE nineties.released > 1990 AND nineties.released < 2000
RETURN nineties.title
```





# WHERE : EXISTANCE D'UNE PROPRIÉTÉ

```
MATCH (p:Person)
WHERE exists(p.birthdate)
RETURN p.name
```

Retourne les personnes ayant une date d'anniversaire.

```
MATCH (p:Person)-[rel:WORKS_FOR]->(:Company)
WHERE exists(rel.startYear)
RETURN p
```

Retourne les personnes ayant une date de début de contrat.





# WHERE : VALEUR PARTIELLE

```
MATCH (tom:Person)
WHERE tom.name STARTS WITH 'Tom'
RETURN tom
```

Retourne les acteurs dont le prénom commence par "Tom".

```
MATCH (p:Person)
WHERE p.name CONTAINS 'k'
RETURN p
```

Retourne les acteurs dont le nom contient la lettre "k".





# WHERE : VALEUR PARTIELLE

```
MATCH (p:Person)
WHERE p.name =~ 'Tom.*'
RETURN p.name
```

```
MATCH (p:Person)
WHERE p.yearsExperience IN [1, 5, 6]
RETURN p.name, p.yearsExperience
```





## WHERE : SOUS-GRAPHE

```
MATCH (p:Person {name:'Tom'})-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE exists((friend)-[:WORKS_FOR]->(:Company {name:'Neo4j'}))
RETURN friend
```

Retourne les amis de Tom qui travaillent pour la société Neo4j.

```
MATCH (p:Person {name:'Tom'})-[r:IS_FRIENDS_WITH]->(friend:Person)
-[:WORKS_FOR]->(:Company {name:'Neo4j'})
RETURN friend
```

Les 2 requêtes donnent exactement le même résultat.





# RETURN : MOTS-CLÉS ASSOCIÉS

```
RETURN [DISTINCT] <expressions>
[ ORDER BY <aliases> [DESC] ]
[ LIMIT <nombre de lignes> ]
```

Le résultat d'une requête peut être contrôlé par quelques mots-clés associés à la clause RETURN:

- DISTINCT
- ORDER BY
- LIMIT





# RETURN : DISTINCT

```
MATCH (p)-[:LIKES]-(t:Technology)
WHERE t.type IN ['Graphs', 'Query Languages']
RETURN p.name
```





# RETURN : LIMIT

Limite la taille de la réponse au nombre de lignes spécifié.

Si le résultat n'est pas trié, le choix des lignes retournées n'a pas de signification particulière.

```
MATCH (p)-[:LIKES]-(t:Technology)
RETURN p.name, count(t) AS numberOfTechnologies
ORDER BY numberOfTechnologies
LIMIT 3
```





# RETURN : ORDER BY

Trie le résultat final (**RETURN**) ou intermédiaire (**WITH**).

(même syntaxe que la clause ORDER BY du SQL)

```
MATCH (p)-[:LIKES]-(t:Technology)
RETURN p.name, count(t) AS numberOfTechnologies
ORDER BY p.name DESC, numberOfTechnologies
```



# Let's graph



**Les CRUD !!!!**



# CRUD



Dans cette section nous aborderons la syntaxe CYPHER pour :

- insérer des nœuds et des relations.
- mettre à jour des propriétés sur des nœuds et des relations.
- supprimer des nœuds, des relations et des propriétés.

Notez que la recherche de données a été abordé dans la section précédente :

## CRUD

Opération	SQL	CYPHER
<u>Create</u>	INSERT	CREATE / MERGE [SET]
<u>Read</u>	SELECT	RETURN
<u>Update</u>	UPDATE	SET
<u>Delete</u>	DELETE	[DETACH] DELETE





# CREATE

```
[ MATCH <patterns> ]
[ WHERE <conditions> ]
CREATE <patterns>
[     SET <expressions> ]
[ RETURN <expressions> ]
```

MATCH/WHERE s'utilisent comme dans une requête en lecture:

- pour relier 2 noeuds existants par une nouvelle relation
- pour obtenir les informations nécessaires à la création des nouveaux objets



# CRUD : création d'un nœud, d'une relation



Création d'un nœud avec une propriété :

```
CREATE (m:Person {name: 'Michael'})  
RETURN m
```

La clause RETURN est optionnelle.

Création d'une relation à partir de 2 nœuds existants :

```
MATCH (j:Person {name: 'Jennifer'})  
MATCH (m:Person {name: 'Michael'})  
CREATE (j)-[rel:IS_FRIENDS_WITH {since:2018}]->(m)
```

Une relation est obligatoirement liée à deux nœuds.

Création de 2 nœuds (Jennifer et Michael) avec une relation de type IS\_FRIENDS\_WITH de Jennifer vers Michael.

```
CREATE (j:Person {name: 'Jennifer'}) -[rel:IS_FRIENDS_WITH]->(m:Person  
{name: 'Michael'})
```

Une autre manière de formuler la création précédente :

```
CREATE (j:Person {name: 'Jennifer'})  
      (m:Person {name: 'Michael'})  
      (j)-[rel:IS_FRIENDS_WITH]->(m)
```



## CRUD : création d'une propriété

2 syntaxes possibles:

- soit dans le <pattern> (plus concis)
- soit avec la clause SET (plus riche, prioritaire)

```
CREATE (j:Person{name: 'Jennifer', })-[rel:IS_FRIENDS_WITH{since : 2018} ]->(m:Person{name:'Michael' })
```

```
CREATE (j:Person)-[rel:IS_FRIENDS_WITH]->(m:Person{name:'Michael' })
SET j.name = 'Jennifer', rel.since = 2018
```

## CRUD : MERGE



La syntaxe de MERGE est identique à celle de CREATE :

```
[ MATCH <patterns> ]
[ WHERE <conditions> ]
MERGE <patterns>
    [ SET <expressions> ]
    [ RETURN <expressions> ]
```

```
MERGE (m:Person {name: 'Michael' })
RETURN m
```

Son comportement est différent de celui de CREATE:

- si il existe, alors le noeud existant est retourné
- sinon le noeud créé est retourné



## CRUD : MERGE



```
MATCH (j:Person {name: 'Jennifer'})  
MATCH (m:Person {name: 'Michael'})  
MERGE (j)-[r:IS_FRIENDS_WITH]->(m)  
RETURN j, r, m
```

Son comportement est différent de celui de CREATE:

- si la relation existe, alors elle est retournée
- sinon la relation créée est retournée

```
MERGE (j:Person {name: 'Jennifer'}) -  
[r:IS_FRIENDS_WITH]->(m:Person {name: 'Michael'})  
RETURN j, r, m
```

ATTENTION: Cette requête peut créer des dupliсats !

C'est l'existence du sous-graphe entier (et non de chacun de ses noeuds/relations) qui est vérifiée.

Par conséquent, l'absence de la relation r a pour conséquence la création de la relation et des 2 noeuds, même si ceux-ci existent déjà.



## CRUD : MERGE



```
MERGE (m:Person {name: 'Michael'}) -[r:IS_FRIENDS_WITH]-
(j:Person {name:'Jennifer'}) SET r.since = date('2018-03-01')
```

La clause SET est exécutée dans tous les cas:

- si la relation existe, alors la propriété r.since de cette relation est modifiée
- sinon la relation est créée, et sa propriété r.since est initialisée

```
MERGE (m:Person {name: 'Michael'}) -[r:IS_FRIENDS_WITH]-
(j:Person {name:'Jennifer'})
    ON CREATE SET r.since = date('2018-03-01')
    ON MATCH SET r.updated = date()
RETURN m, r, j
```

Permet de contrôler le comportement souhaité pour la clause SET:

- ON CREATE SET n'est exécuté qu'en cas de création d'une nouvelle relation
- ON MATCH SET n'est exécuté que si la relation existait déjà



## CRUD : SET



**CREATE | MERGE | MATCH <patterns>**  
**SET <variable>. <propriété> = <valeur>**

Affecte une nouvelle valeur aux propriétés des objets sélectionnés ou créés.

- ajout de propriétés
- mise à jour de propriétés
- suppression de propriétés

```
MATCH (j:Person {name: 'Jennifer'}) -[rel:WORKS_FOR]- (:Company
{name: 'Neo4j'})
SET j.birthdate = date('1980-01-01'), rel.startYear =
date({year: 2018})
RETURN p, rel
```

Si le nœud existe, alors la clause SET est exécutée dans les 2 cas:

- si la propriété existe, elle est modifiée
- sinon elle est créée et initialisée



## CRUD : DELETE



```
MATCH <patterns>
[DETACH] DELETE <variable>
```

Permet de supprimer différents types d'objet:

- noeud
- relation
- sous-graphe

```
MATCH (m:Person {name: 'Michael'})
DELETE m
```

Provoque une erreur si ce nœud a toujours des relations, car Neo4j est ACID-compliant.

```
MATCH (j:Person {name: 'Jennifer'}) -[r:IS_FRIENDS_WITH]-
>(m:Person {name: 'Michael'})
DELETE r
```

Supprime la relation r



## CRUD : DELETE



```
MATCH (m:Person {name: 'Michael'})  
DETACH DELETE m
```

Supprime les relations du noeud m et le noeud lui-même

```
MATCH (n:Person {name:  
'Jennifer'}) REMOVE n.birthdate
```

```
MATCH (n:Person {name:  
'Jennifer'}) SET n.birthdate =  
null
```

Neo4j ne stocke pas de valeur nulle.

Une propriété de valeur nulle équivaut donc à l'absence de cette propriété (schema-less).



Let's graph





## Démarrer le container neo4j

```
$ docker start neo4j
```

```
$ docker start jupyter
```

Allez sur à l'url pour l'énoncé des TP : <http://127.0.0.1:8888/lab>

L'interface jupyterLab s'ouvrira dans votre navigateur.

Si c'est la première fois :

- Une interface vous demandera un mot de passe lequel est cctraining
- Suivez les instructions de la page de bienvenue pour configurer correctement votre environnement.

Ouvrez le fichier Neo4J\_TP02 CRUD.ipynb

L'interface graphique de Neo4J sera utilisée pour passer les commandes interactives:

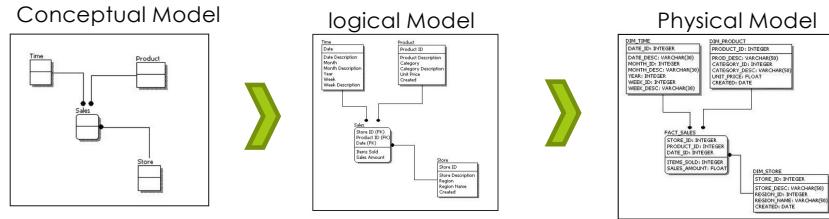
<http://127.0.0.1:7474>





## Modélisation graphe

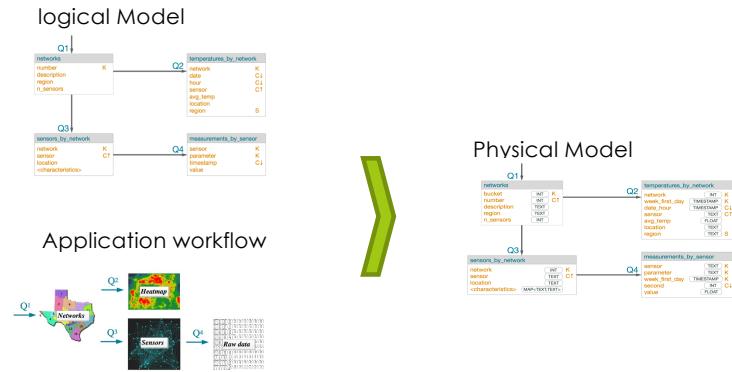
# Introduction à la modélisation NoSQL



## Modèle de données relationnel

- Le modèle métier est représenté conceptuellement par des entités du monde réel lesquelles sont mises en relation entre elles par des associations.
- Le modèle logique n'est autre qu'une représentation des données sous une structure bien particulière :
  - Tables à 2 dimensions.
  - Un ensemble de contrainte d'intégrité : les clés primaires, les clés étrangères, les contraintes d'unicité ...
  - La normalisation évite la duplication des données ( anomalies de lecture et d'écriture) et diminue la volumétrie globale ( 1970 le prix du Mo 300\$ )
  - Plus la normalisation est forte et plus le nombre de tables dans la base de données va augmenter.
  - Plus de tables implique des requêtes plus complexes (requête à multi-jointures) et potentiellement des temps de latences plus important.
- Le modèle physique n'est autre que l'implémentation du modèle logique dans une Système de Gestion de Base de Données.
- **La finalité, le modèle physique est dé-corrélé de la manière dont l'application accède aux données.**

# Introduction au modèle orienté graphe



Dans les bases de données relationnelles, le métier guide le modèle de données et est complètement dé-corrélé de la manière dont les données seront accédées par l'application.

Si les performances ne sont pas au rendez-vous, on ajoute des structures complémentaires pour palier à ce problème ce qui ne suffit pas tout le temps surtout dans un environnement BIG DATA.

Dans le monde NoSQL, le modèle de donnée prend en compte dès la conception du modèle la manière dont les accès seront réalisés. Cette approche confère au modèle NoSQL :

- La performance est privilégiée.
- Le modèle physique est construit à partir du modèle conceptuel et de la nature des accès aux données (INSERT, SELECT, UPDATE ...).
- Le modèle physique est organisée de manière optimale pour l'application. Toutefois, la normalisation des données est souvent incompatible avec cette approche et les données sont souvent dupliquées.
- Gagner en performance au détriment de la cohérence : Approche Query-Driven
- Il n'y a pas de modèle parfait, le modèle est parfait à partir du moment où il n'y a pas de problèmes de performance.

# La modélisation Graphe avec NEO4J



4 structures basiques avec Neo4j:

- Nœuds : il représente généralement des entités du monde réel une personne, une société ...
- Relations : existent entre les différents nœuds. elles sont accessibles de manière autonome ou via les nœuds auxquels ils sont attachés. Les relations peuvent également contenir des propriétés. Chaque relation a un nom et une direction, qui ensemble fournissent un contexte sémantique pour les nœuds connectés par la relation.
- Propriétés : les nœuds et les relations peuvent avoir des propriétés. Les propriétés sont définies par des paires clé-valeur.
- Label : peuvent être utilisées pour regrouper des nœuds similaires afin de faciliter une traversée plus rapide des graphiques.



# Processus de modélisation



1 - Analyser les compétences des entreprises.

# Processus de modélisation



## 2 - Identifier les requêtes par priorité :

**Q1 : Rechercher les employés et les entreprises disposant d'une compétence spécifique ?**

**Q2 : Rechercher les compétences par entreprise ?**

**Q3 : Rechercher les employés avec les mêmes compétences ?**

# Processus de modélisation



## 3- Identifier les entités et leurs attributs

2 approches possibles :

- Modèle Entité Association ( EA ) vers graphe
- Modèle Whiteboard



# Processus de modélisation : EA vers graphe



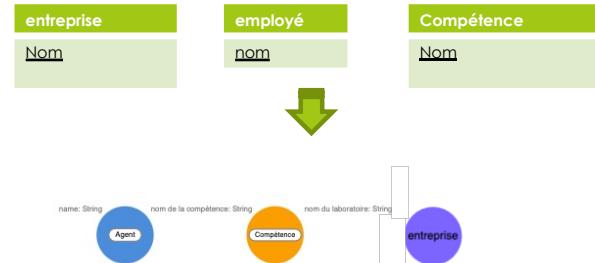
## 3- Identifier les entités et leurs attributs à partir des requêtes :

- Identifier les classes d'entité et leurs attributs.
- Les classes d'entités seront alors représentées par des Labels.
- Les attributs des classes d'entités (les colonnes d'une table) deviennent des propriétés pour les nœuds.
- Chaque instance d'entité est un nœud unique : l'unicité est garantie par la PK.

**Q1 : Rechercher les employés et les entreprises disposant d'une compétence spécifique ?**

**Q2 : Rechercher les compétences par entreprise ?**

**Q3 : Rechercher les employés avec les mêmes compétences ?**



# Processus de modélisation : EA vers graphe



4 - Définir le nom de la relation et leurs propriétés qui lie les entités entre elles par un verbe :

## Approche par modèle Entité Association vers graphe



# Processus de modélisation : EA vers graphe

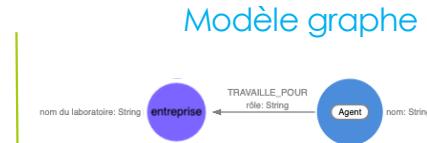
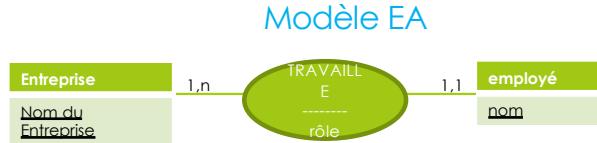


Les associations 1-N sont représentées par un arc, la direction de l'arc dépend du contexte métier:

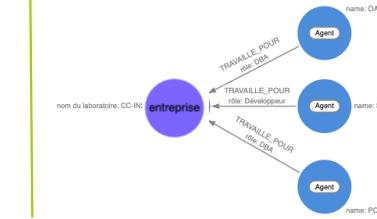


Les associations ( N-M ) sont transformées en arc et les attributs deviennent des propriétés de l'arc.

La direction de l'arc est définie suivant les besoins métier.



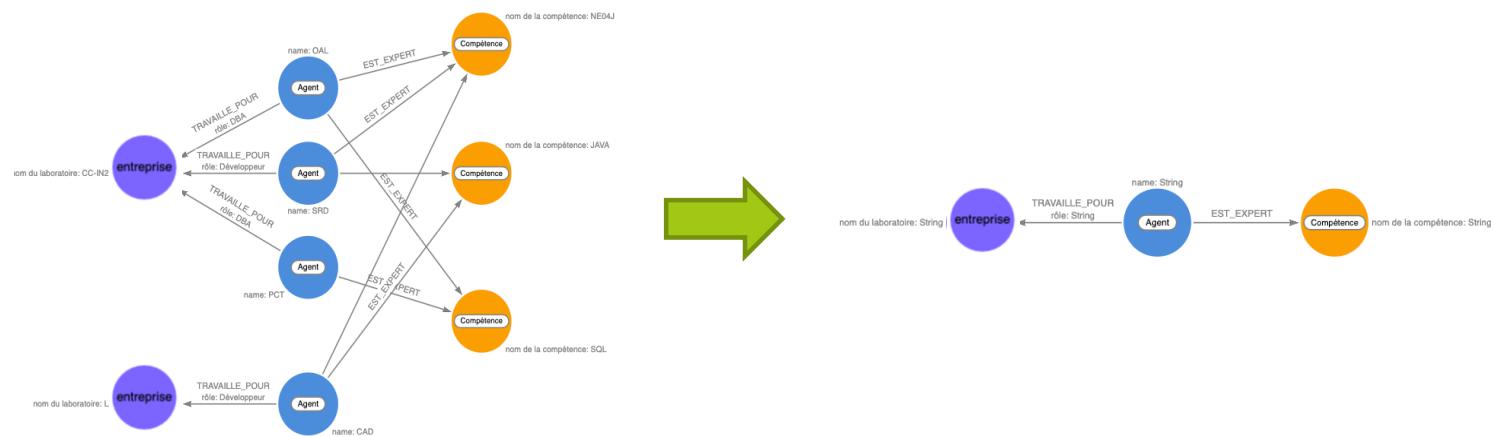
Exemple de graphe



# La modélisation graphe : Whiteboard



Modéliser les données par le dessin (Whiteboard) :



# Processus de modélisation



## 5 - Exprimer les requêtes en Cypher :

- NEO4J parcours les données avec une approche par profondeur.

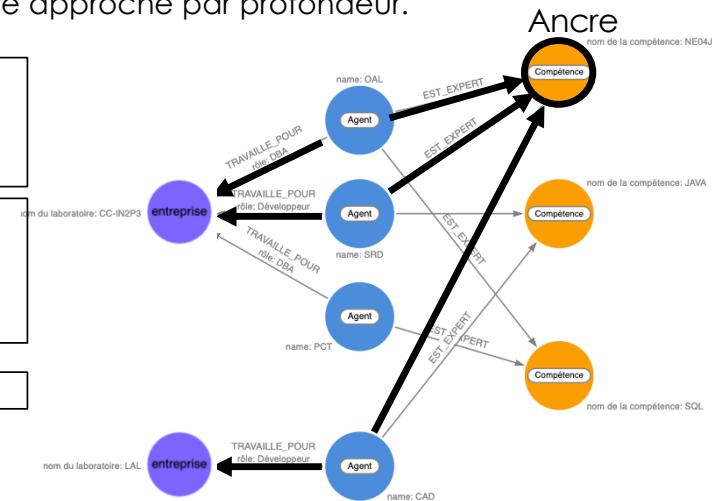
Q1 : MATCH

```
(labo:Entreprise)-[:TRAVAILLE_POUR]-(person:employé)-[:EST_EXPERT]->(skill:Compétence)  
WHERE skill.nom_de_la_compétence = 'NEO4J'  
RETURN employé, labo
```

Q2 :

```
MATCH (labo:Entreprise)-[:TRAVAILLE_POUR]-(person:employé)  
WITH labo, person  
(person)-[:EST_EXPERT]->(skill:Compétence)  
RETURN labo, skill
```

Q3 : ...



# Processus de modélisation



## 6 - Valider le modèle

A partir d'un échantillon de données valider le modèle de données pour chacune des requêtes.

Q1

Q2

Q3

Si une requête est défaillante reprendre l'étape 3 et 4. Certains optimisations peuvent être réalisées par certaines astuces.



## Démonstration



Pour modéliser sous forme de graphe vous pouvez utiliser l'outil arrows :

<https://arrows.app/>

Une vidéo d'introduction à arrows est disponible [ici](#).



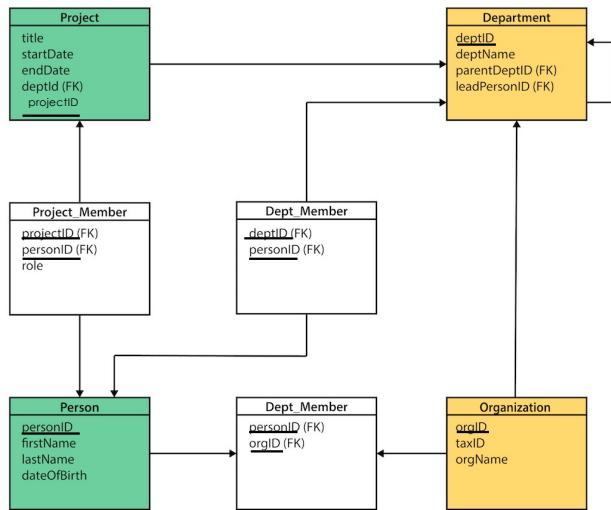
## TD modélisation



# Travaux pratique : exercice 1



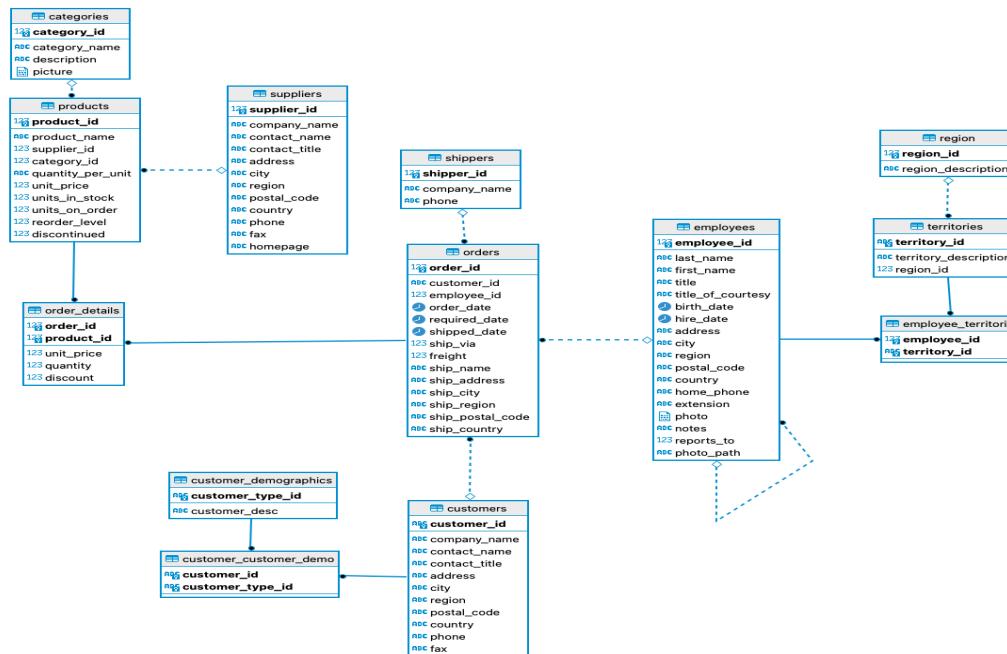
Identifiez les labels et les attributs ?



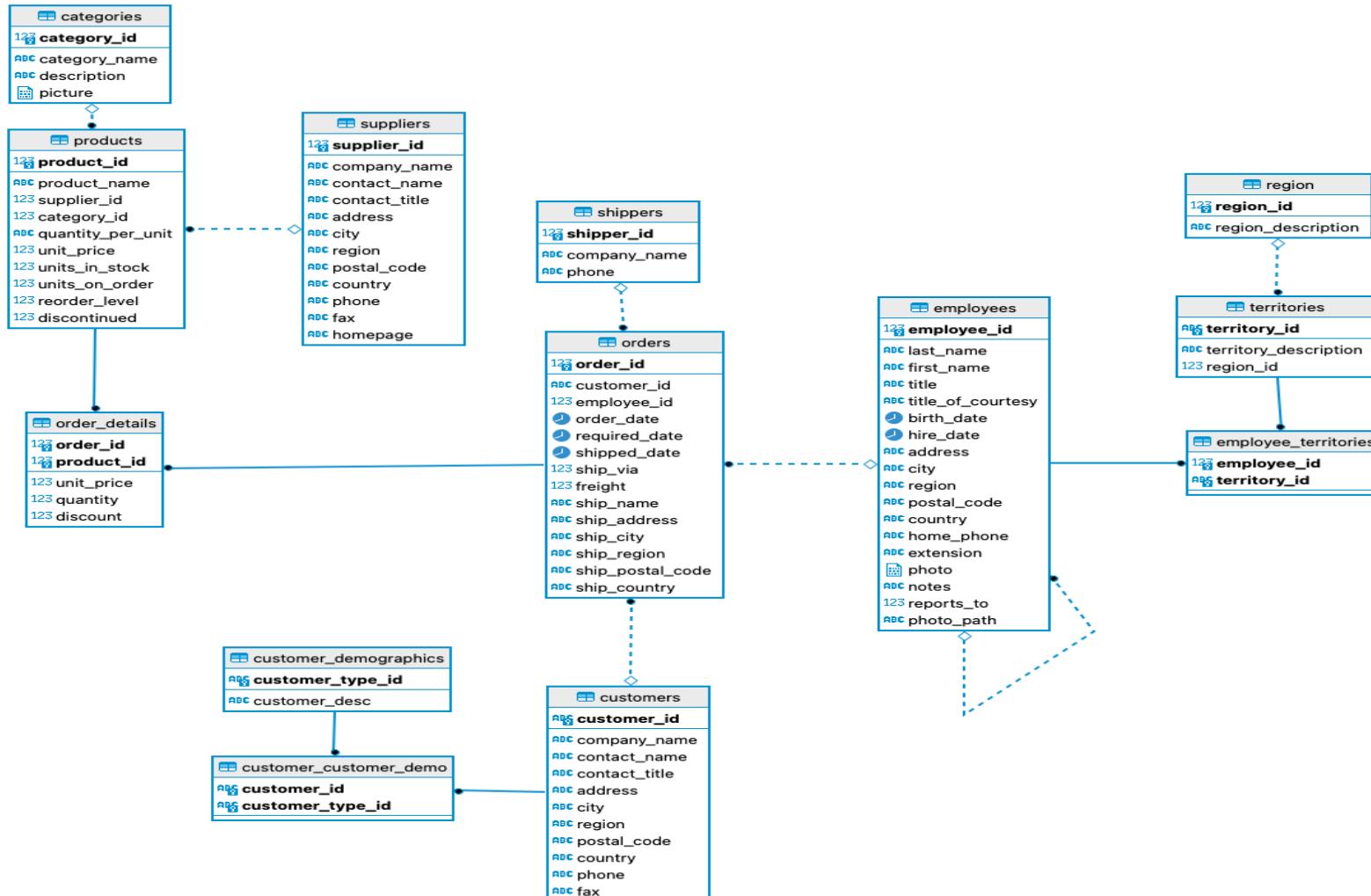
Identifiez les arcs et les attributs des relations ?

## Travaux pratique : exercice 2

Vous travaillez pour un site de vente en ligne qui souhaite évaluer le passage vers une base de données graphe. Le site repose sur une base de données relationnelle dont le schéma est décrit ci-dessous :



# Travaux pratique : exercice 2



## Travaux pratique : Exercice 2



En vous appuyant sur le schéma relationnel précédent proposez un graphe en sachant que le site soumet quotidiennement les requêtes SQL suivantes :

Q1

```
SELECT e.last_name, e.first_name, count(*) AS  
CountOrder  
FROM employees AS e  
JOIN Order AS o  
    ON (o.EmployeeID = e.EmployeeID)  
GROUP BY e.last_name, e.first_name  
ORDER BY CountOrder DESC;
```

Q2

```
SELECT e.last_name,  
et.territory_description  
FROM employees AS e  
JOIN employee_territories AS et  
    ON (et.employee_id =  
e.employee_id)  
JOIN territories AS t  
    ON (et.territory_id =  
t.territory_id );
```

Q3

```
SELECT DISTINCT c.CompanyName  
FROM customers AS c  
JOIN orders AS o  
    ON (c.CustomerID = o.CustomerID)  
JOIN order_details AS od  
    ON (o.OrderID = od.OrderID)  
JOIN products AS p  
    ON (od.ProductID = p.ProductID)  
WHERE p.ProductName = 'Chocolade';
```



## Travaux pratique : exercice 3



Les brasseurs produisent des bières de différents types (Blonde, brune, ...) sous différentes marques. On pourra prendre l'exemple du brasseur Kronenbourg dont la description est donnée ci-dessous :

[https://fr.wikipedia.org/wiki/1664\\_\(marque\)#Kronenbourg\\_1664](https://fr.wikipedia.org/wiki/1664_(marque)#Kronenbourg_1664)

Kronenbourg brasse différentes bière ( 1664, 1664 Blanc, 1664 Rosé ...) et chaque bière est d'un certain type : Blonde, Blanche, Rosé ...

Proposer à partir de l'exemple du brasseur Kronenbourg, un modèle graphe qui permette de répondre aux questions suivantes :

- Quels sont les différents bières produites par un brasseur ?
- Quels sont les différents bières produites par un brasseur pour un type de bière particulier ?



## Evolution du modèle

La modélisation est basée sur les requêtes formulées. C'est une approche empirique. Si le résultat nous convient, that's it. On peut faire évoluer le modèle sur la base des prédictats des requêtes.

3 manières différentes de modéliser un attribut d'une classe d'entité :

- Par un label indiquant l'existence ou non de l'attribut ( exemple (:Person:Feminin))
- Par une propriété (:Person { genre: 'Feminin' })
- Par un nœud (:Person)-[:GENRE]->(:Genre { nom: 'Feminin' })

2 critères à prendre en compte :

- La cardinalité: le nombre de valeurs distinct ( genre : Masculin/ Feminin)
- La vélocité : A quelle fréquence la valeur peut elle changer ? Le genre d'une personne change très rarement.



## Evolution du modèle



Les labels permettent des recherches rapides dans Neo4j, mais elles n'expriment réellement que des booléens, c'est-à-dire la présence ou l'absence d'une catégorie telle que masculin ou féminin.

### Quand considérer un attribut d'un nœud comme un Label ?

- Les labels fonctionnent pour les attributs à faible cardinalité et lorsque les catégories peuvent se chevaucher.
- Les labels sont généralement utilisées pour partitionner les graphes.
- La vitesse est faible.
  - Exemple : (:Person:French:US)

### Quand l'éviter :

- Il n'est pas conseillé d'avoir plus de 4 labels sur un nœud. En pratique, un nœud dispose dans la plupart des cas de 1 Label voire 2. Au-delà de 4 Labels, les performances peuvent se dégrader.
- NEO4J recommande que les labels soient orthogonaux entre eux. Un label Personne et Compétence n'ont pas de lien contrairement à un label Personne et employé.
- La cardinalité est importante.
- La vitesse est forte: modifier un label est une opération coûteuse.



## Attribut d'une relation comme Type

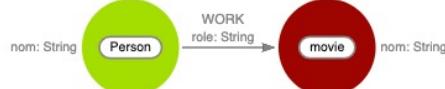


Les propriétés d'une relation peuvent être représentées comme type de relation si :

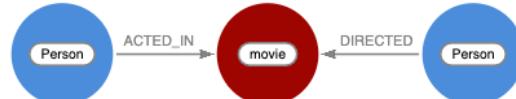
- ❑ La propriété est statique.
- ❑ La cardinalité de la propriété est faible.

Dans NEO4J, le coût d'analyse d'une relation est constant et la recherche d'un pattern à partir d'une relation sera plus efficace qu'une recherche sur la propriété d'un nœud.

Exemple : Rechercher les films dirigés par Clint Eastwood ?



VS



```
Match(clint:Person)-[relation:WORK]->(film:Movie)
Where clint.nom='Clint Eastwood' AND relation.role='director'
Return film
```

```
Match(clint:Person)-[r:DIRECTED]->(film:Movie)
Where clint.nom='Clint Eastwood'
Return film
```



## Attribut d'un nœud comme propriété



### Quand considérer un attribut d'un nœud comme une propriété ?

- la propriété est l'option la plus simple. elles sont flexibles, indexables et faciles à utiliser.
- Bien adapté pour les fortes cardinalités.
- Elles peuvent être soumises à des contraintes de domaines.
- Lorsque les données changent fréquemment.

### Quand éviter :

- lorsque l'attribut est de type complexe : tableau ou map. Le temps de traitement peut être coûteux .
- Lorsqu'on recherche d'autres nœuds qui partagent une même propriété.

```
MATCH (p:Person { id: 5 })
WITH p
MATCH (othersInSamePostalCode :Person { postalCode: p.postalCode })
RETURN othersInSamePostalCode;
```



## Attribut d'un nœud comme noeud

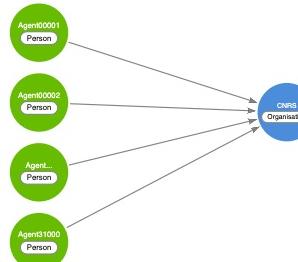


Quand considérer un attribut d'un nœud comme un nœud ?

- ❑ Idéal lorsque vous devez rechercher d'autres nœuds qui partage une valeur de propriété.
- ❑ Lorsque la cardinalité est très élevée. Par exemple, si un prédictat sur l'une de vos requêtes consiste à filtrer les personnes par CodePostal.

Quand éviter :

- ❑ Cette approche peut provoquer l'apparition de nœuds denses ou super nœud.
- ❑ Il existe souvent une relation entre la cardinalité et la sélectivité; moins une variable peut prendre d'options, plus elle aura d'arc avec un grand nombre de nœuds.
- ❑ D'un autre côté, plus une variable a d'options, moins vous avez de chances de vous retrouver avec un super-nœud. "Job" a probablement des milliers de codes possibles.



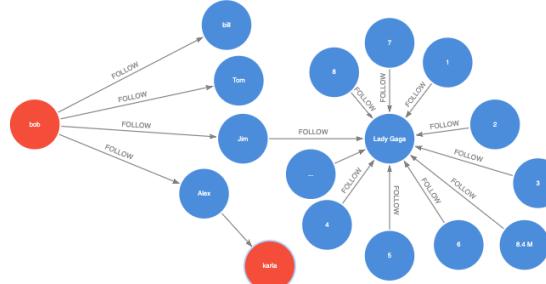
## Les super nœuds

Un super nœud ou nœud dense est un nœud connecté avec un grand nombre de nœuds.

Il devient un réel problème pour les parcours de graphes. NEO4J devra évaluer toutes les relations connectées à ce nœud afin de déterminer quelle sera la prochaine étape du parcours de graphe.

Twitter : Cherchons l'ensemble des chemins qui lie Bob à Karla :

```
MATCH (bob:TwitterUser { name: "bob" })
WITH bob
MATCH p=(bob)-[:FOLLOW*-]-(:TwitterUser { name:
"karla" })
RETURN p
```



**Lady Gaga** @ladygaga

"Chromatica" 🎵 OUT NOW [smarturl.it/Chromatic](http://smarturl.it/Chromatic) | Love For Sale 💃 October 1  
TonyGaga.Ink.to/LoveForSale | House of Gucci 🇮🇹 @houseofguccimov

[ladygaga.com](http://ladygaga.com) 📱 A rejoint Twitter en mars 2008

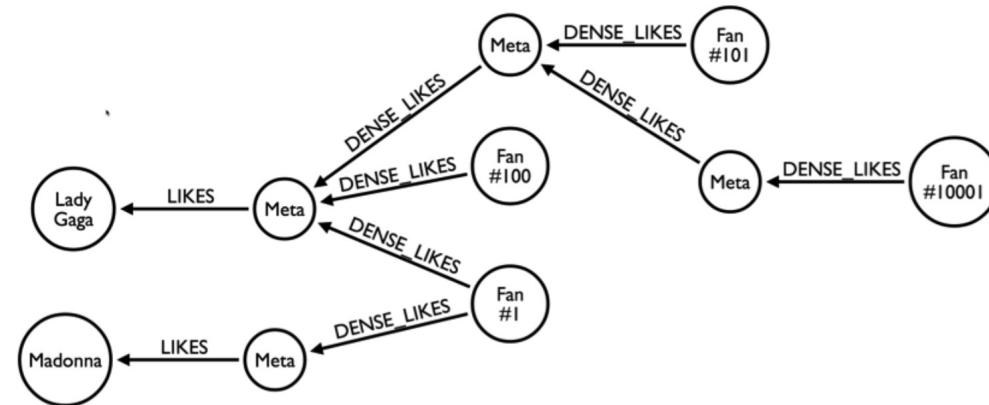
119,2 k abonnements 83,6 M abonnés

## Evolution du modèle : Les super nœuds



Comment résoudre le problème des super nœuds :

- Eviter les requêtes bi-directionnelles. En précisant la direction, on limite le parcours à 120 000 utilisateurs que suit Lady Gaga : au lieu des 80M d'utilisateurs.
- Les supers nœuds ne sont pas problématiques si on ne les parcourt pas. Ils doivent donc être le dernier nœud du parcours.
- On divise pour mieux régner.



## Travaux pratique : exercice 3.2



On souhaite à présent faire évoluer le modèle afin d'identifier les bières avec le même le % d'alcool.

Proposer un modèle graphe ?

Vous pouvez utiliser l'outil arrows.



## Travaux pratique : exercice 3.3



On souhaite à présent faire évoluer le modèle afin d'identifier les bières avec un % d'alcool supérieur à une bière donnée.

Proposer un modèle graphe ?

Vous pouvez utiliser l'outil arrows.



## Travaux pratique : exercice 3.4



On souhaite à présent alimenter le modèle pour conserver les ventes par marque de bières tous les mois avec une rétention de 5 ans.

On notera que le nombre des ventes par marque est fournie au premier de chaque mois.

Proposer un modèle graphe ?

Vous pouvez utiliser l'outil arrows.





**Haute disponibilité**



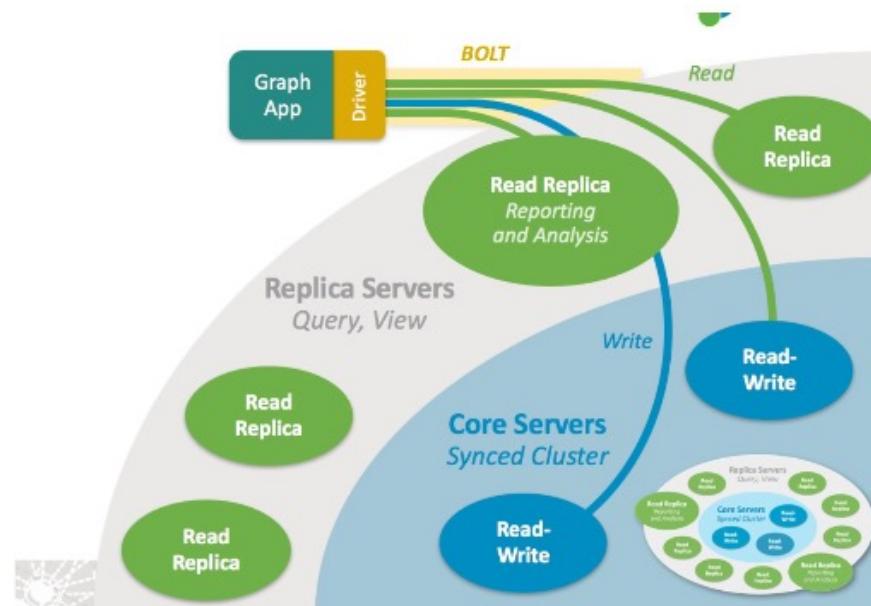
**Haute disponibilité**

# Haute disponibilité

NEO4J intègre des mécanismes de haute disponibilité :

- RéPLICATION synchrone
- RéPLICATION asynchrone
- Sharding

Enterprise Edition

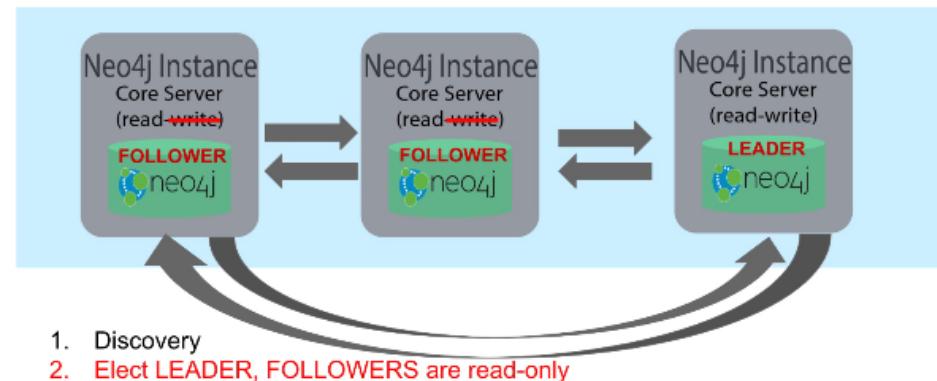


# Haute disponibilité

Enterprise Edition

## RéPLICATION SYNCHRONE :

- Elle est utilisée pour les accès en lecture et en écriture sur plusieurs CORE serveurs distants.
- Les transactions sont validées si le quorum est atteint.
- La coordination entre les CORE serveurs s'appuie sur le protocole Raft.
- Le nombre de CORE serveur dépend du niveau de protection que l'on souhaite :  $2 * \text{pannes tolérées} + 1$ .
- Un seul CORE serveur est défini comme LEADER et les autres sont déclarés en tant que FOLLOWER.
- Si le LEADER tombe en panne, le FOLLOWER le plus à jour sera élu comme nouveau LEADER.



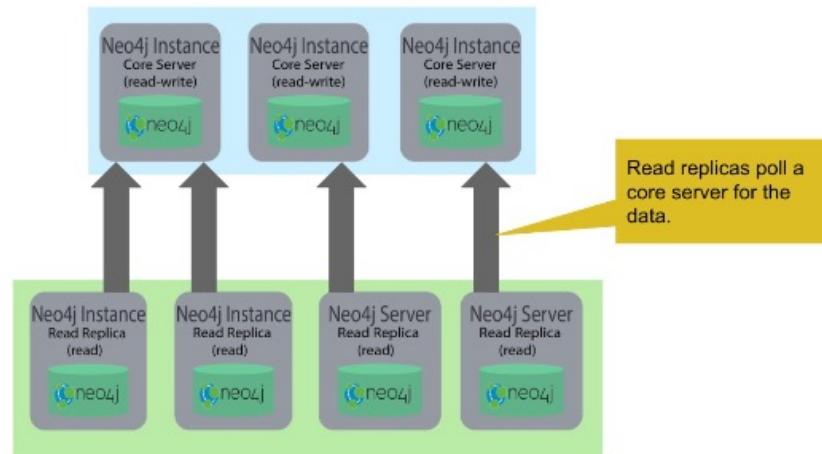
# Haute disponibilité



Enterprise Edition

## RéPLICATION ASYNCHRONE :

- Elle est utilisée pour distribuer la charge en lecture.
- Les écritures sont assurées uniquement par le ou les CORE serveurs.
- Les répliques appelés READ serveurs interrogent régulièrement les CORE serveurs pour obtenir les changements.
- Les READ serveurs agissent comme un cache de données.
- En cas de panne d'un READ serveur, l'application peut être redirigée vers un autre READ serveur survivant.

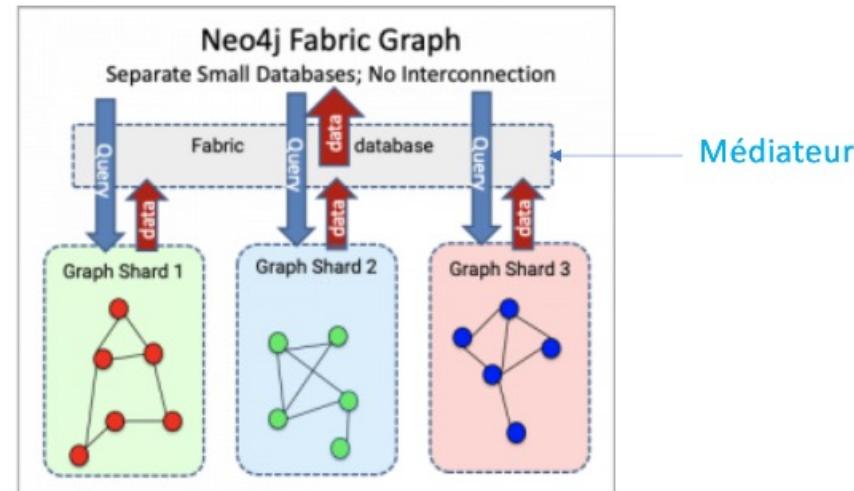


# Haute disponibilité

Enterprise Edition

## Le partitionnement / Sharding :

- Le principe est de découper un graphe en plusieurs bases de données et de les regrouper en une base de données virtuelle.
- Chaque base de données est indépendante.
- Pour lier les bases de données entre elles, NEO4J implémente un médiateur appelé Fabric

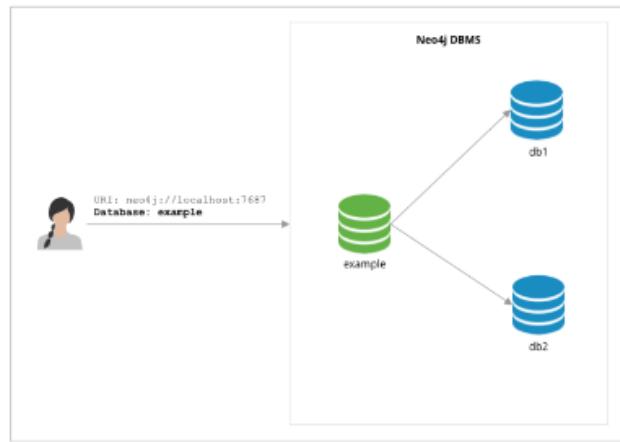


Base de données distribuées : Sharding / partitionnement

# Haute disponibilité

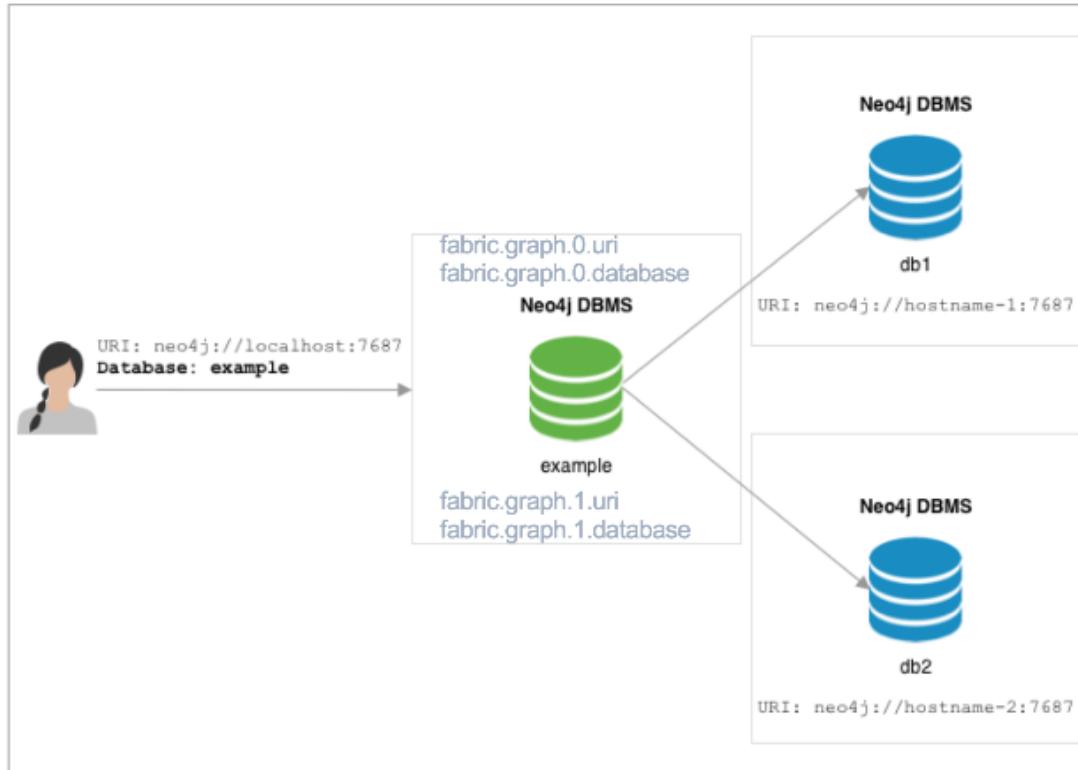


La « Fabric » ne contient pas de données, c'est une sorte de proxy.



```
USE db1  
MATCH (n:Movie) RETURN n  
UNION  
USE db2  
MATCH (n:Movie) RETURN n
```

# Haute disponibilité



```
USE example.graph(0) MATCH (n)  
RETURN n  
UNION  
USE example.graph(1) MATCH (n)  
RETURN n
```

```
UNWIND example.graphIds() AS graphId  
CALL {  
    USE example.graph(graphId)  
    MATCH (n)  
    RETURN n  
}  
RETURN n
```

