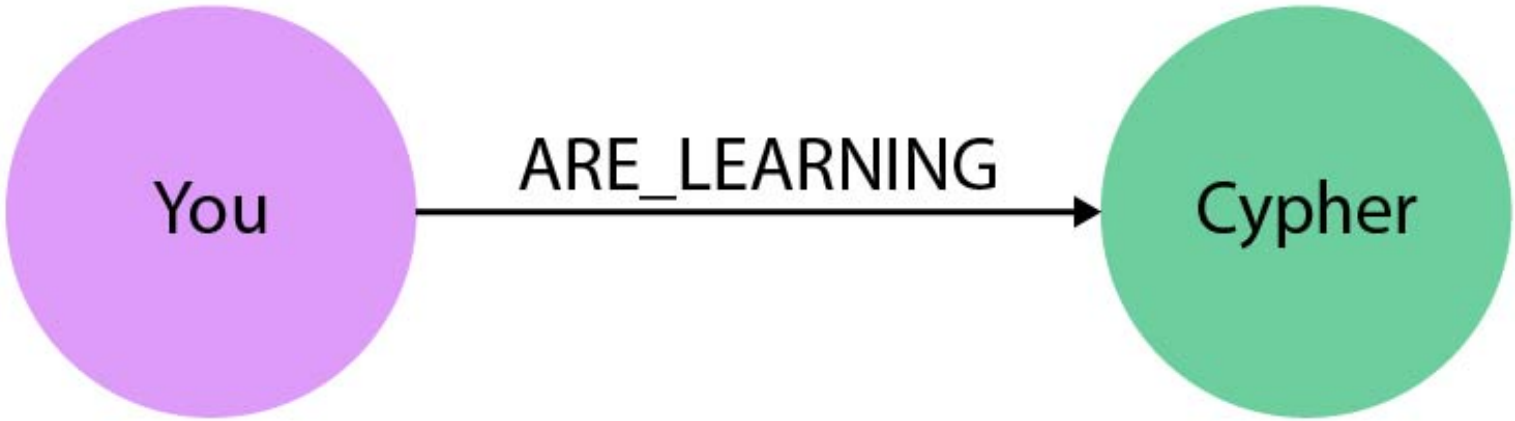
Bases de données spécialisées

Partie 2 : Bases de données orientées graphes
neo4j et Cypher



Cristina Sirangelo
IRIF, Université Paris Cité
amelie@irif.fr

Transparent de cours de Amélie Gheerbrant IRIF, Université Paris Cité amelie@irif.fr

Neo4j et Cypher

- Neo4j est le sgbd graphe le plus utilisé :

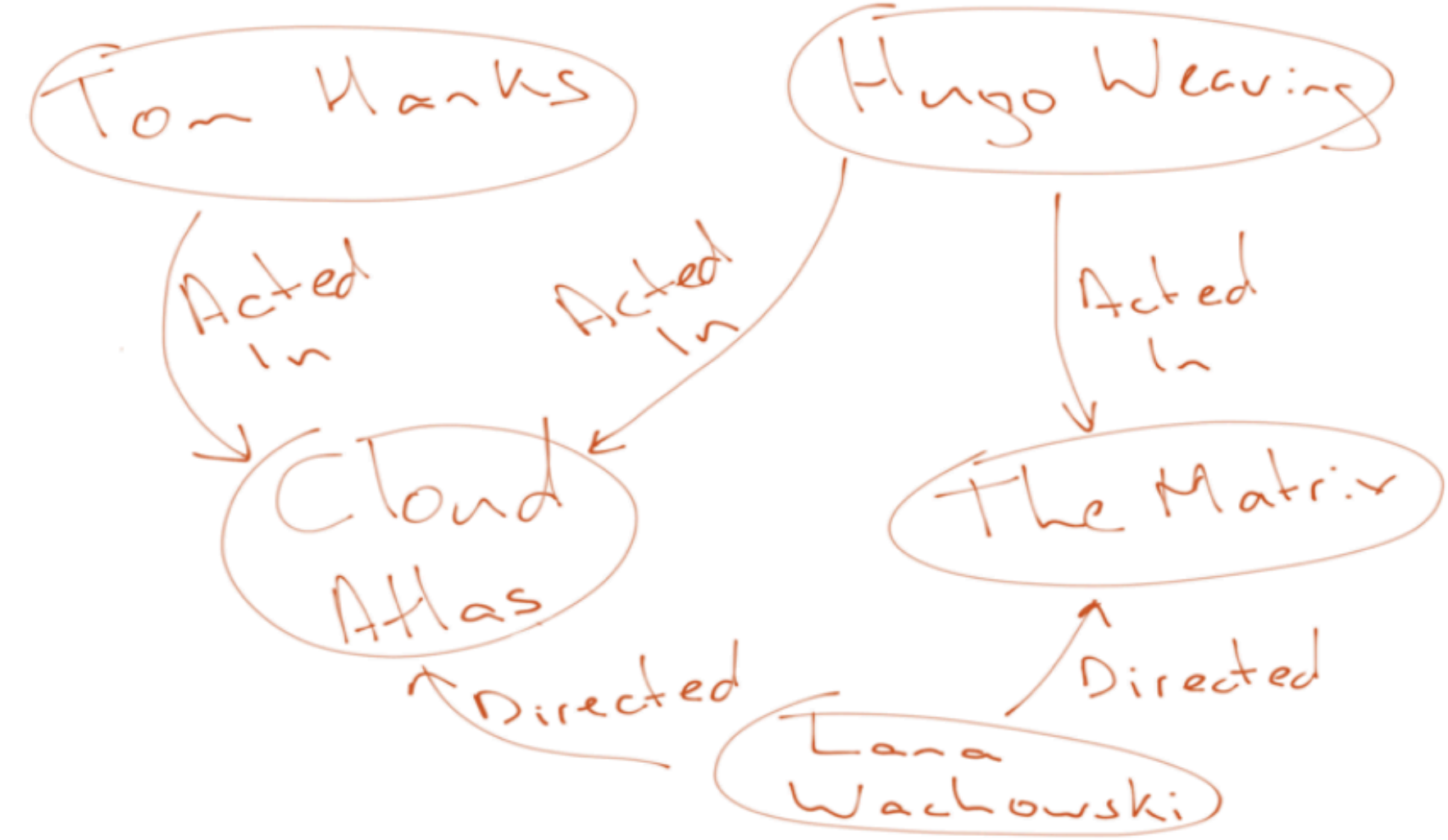
https://db-engines.com/en/ranking/graph+dbms

- Modèle : graphe de propriété (graphe dont les noeuds et les arrêtes comportent des paires nom-valeur de valeurs de données)
- Language de requête : Cypher
  - ASCII-art pattern matching + fonctionnalités BD usuelles

The #1 Database for Connected Data

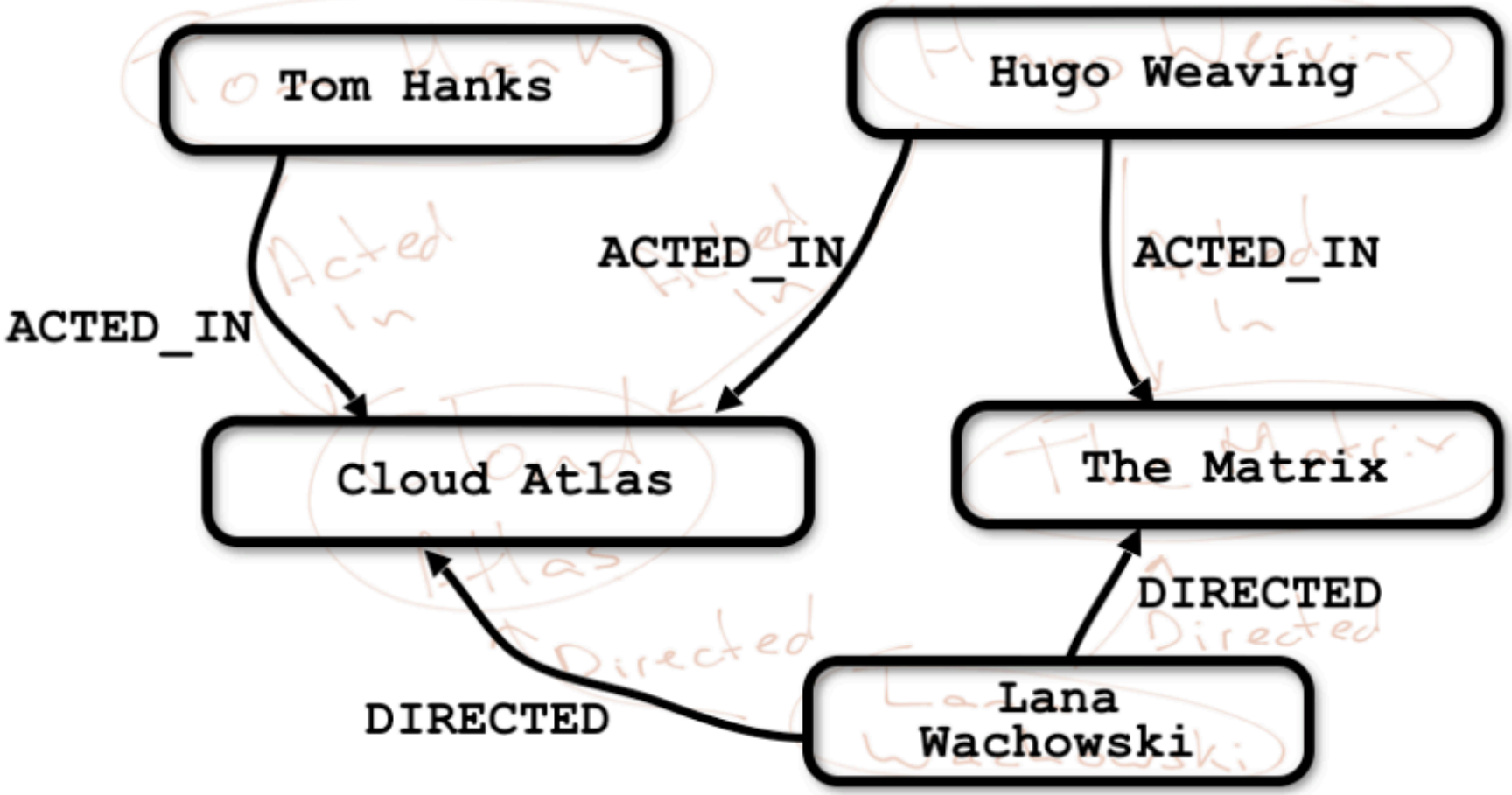
Whiteboard friendliness

Matrix - whiteboard model



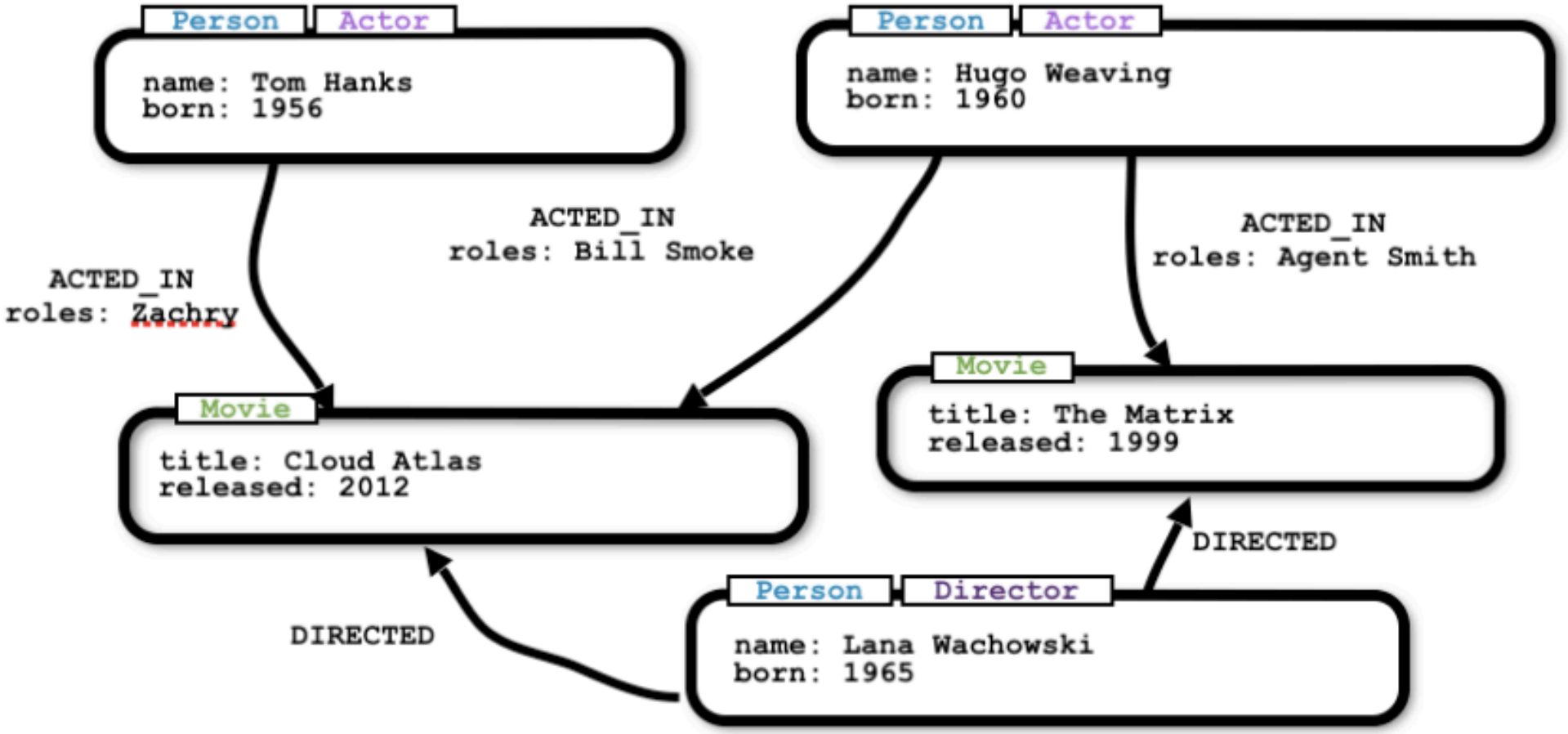
Whiteboard friendliness

Matrix - match node and relationship format of property graph model

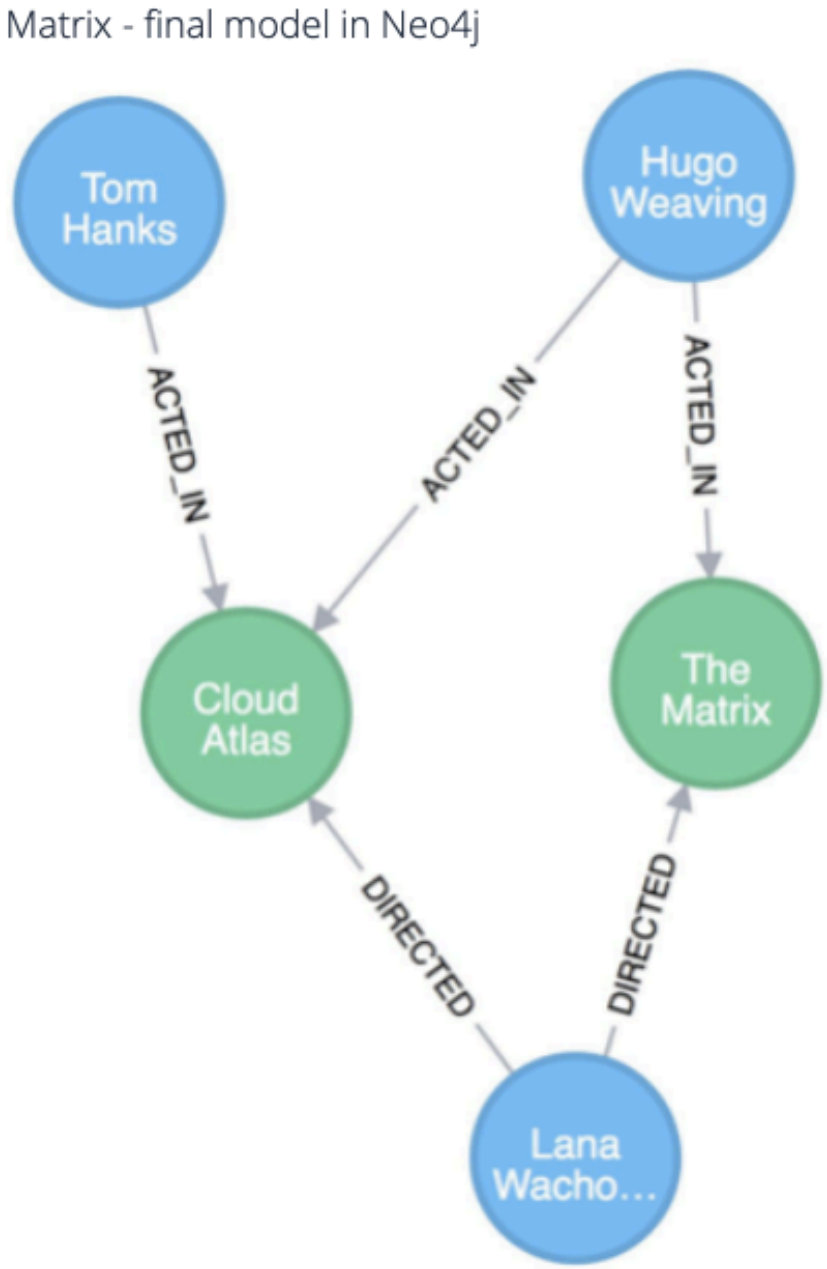


Whiteboard friendliness

Matrix - add labels and properties

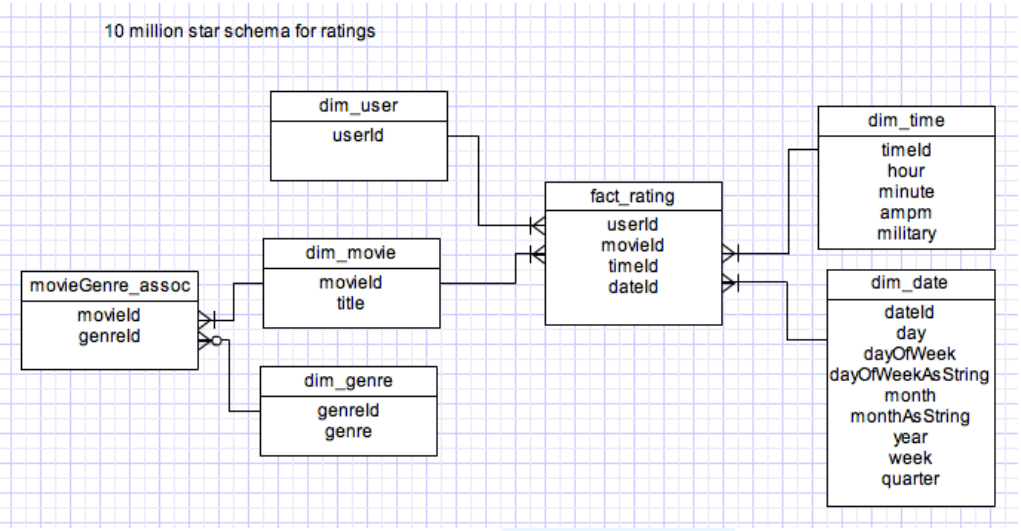


Whiteboard friendliness



L' étape pénible de génération de diagramme entité relation (cas relationnel) : on oublie !!!

oui, ça



Neo4j : les fondamentaux

- Les noeuds
- Les relations
- Les propriétés
- Les labels

The #1 Database for Connected Data

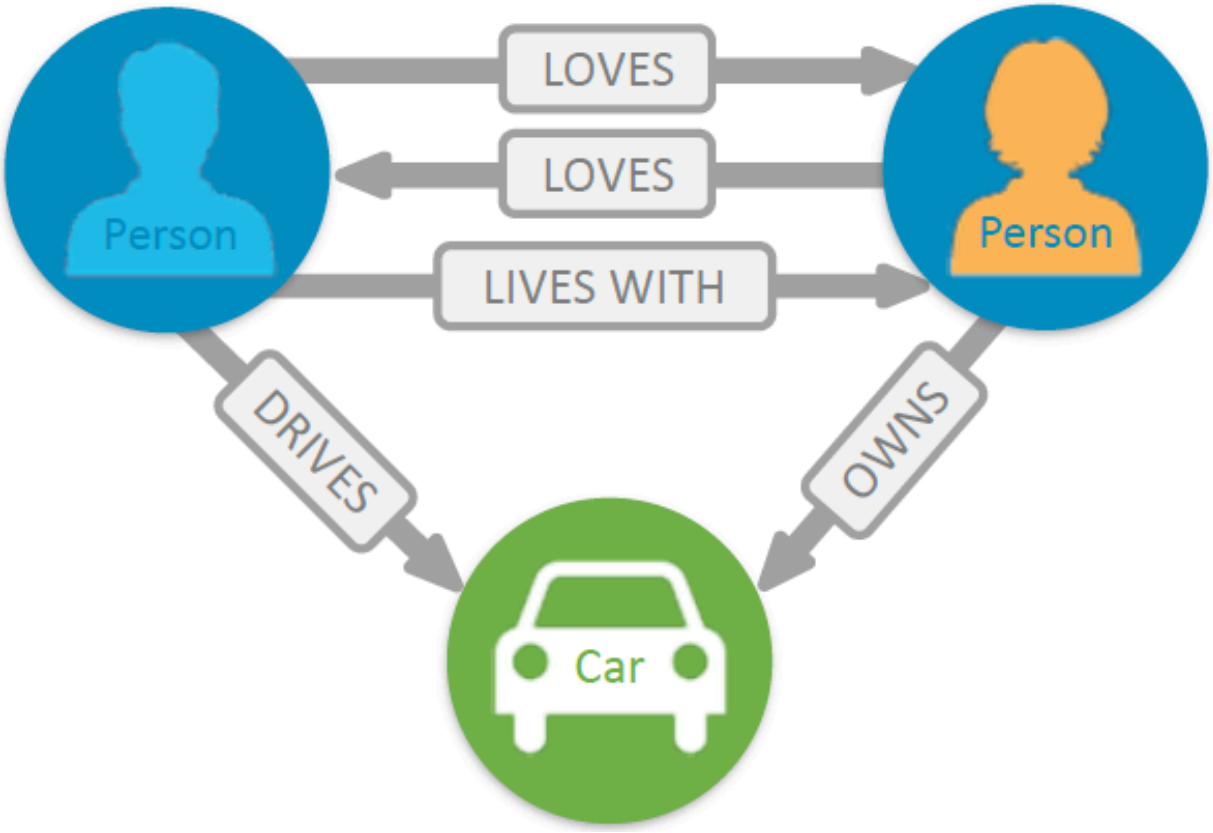
Les composants du modèle de graphe de propriété

- Les noeuds
  - représentent les objets dans le graphe (~tuples)
  - peuvent comporter un ou des labels / types / étiquettes

Les composants du modèle de graphe de propriété

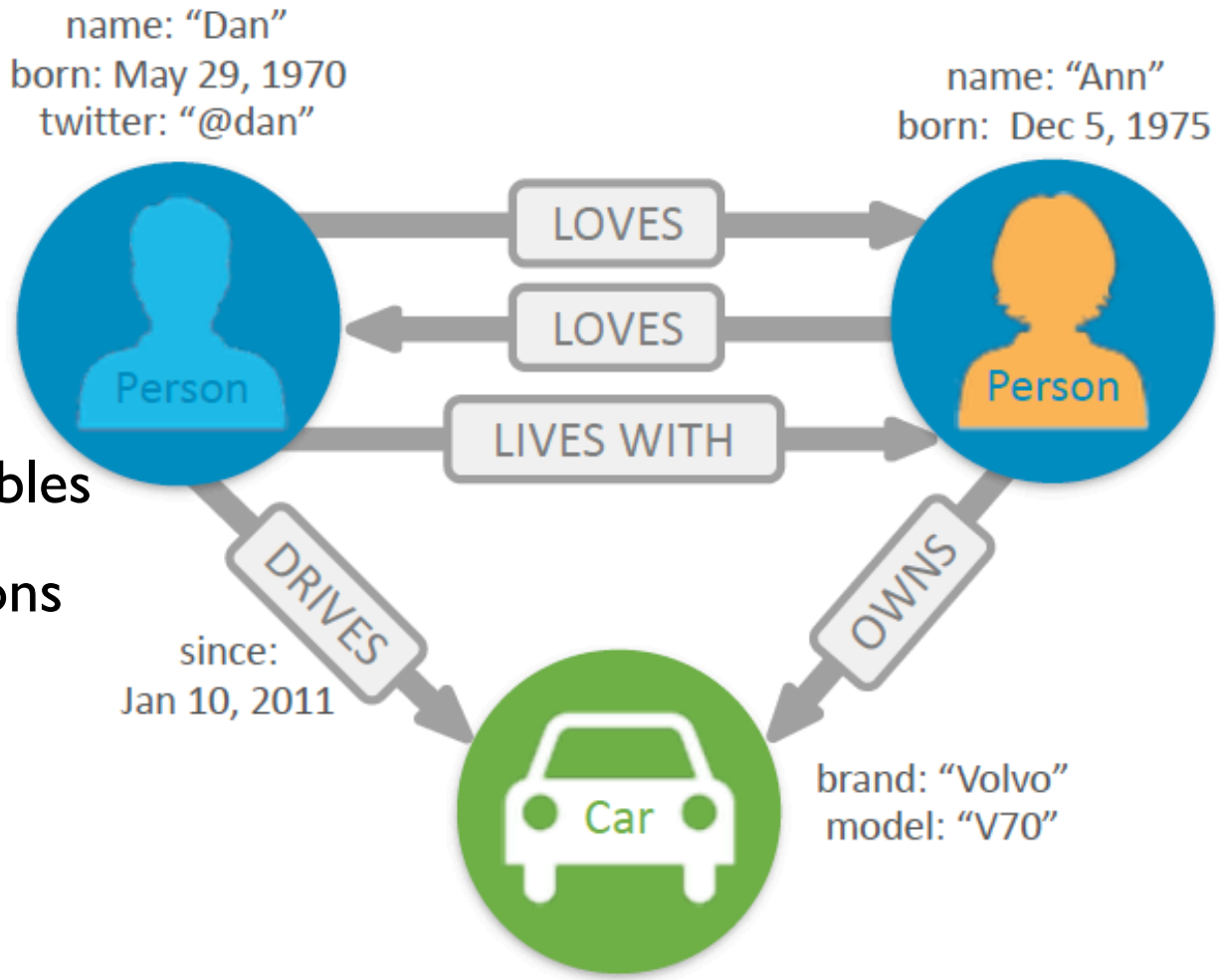
- Les noeuds
  - représentent les objets dans le graphe (~tuples)
  - peuvent comporter un ou des labels / types / étiquettes
- Les relations
  - relient les noeuds par

    type et direction



Les composants du modèle de graphe de propriété

- Les noeuds
  - représentent les objets dans le graphe (~tuples)
  - peuvent comporter un ou des labels / types / étiquettes
- Les relations
  - relient les noeuds par type et direction
- Les propriétés
  - paires nom-valeur possibles sur les noeuds et relations



Briques de base du graphe : résumé

- Les noeuds - entités et types de valeur complexes
- Les relations - connectent les entités et structure le domaine
- Les propriétés - attributs d'entité, qualités de relations, métadonnées
- Les labels - groupent les noeuds par role

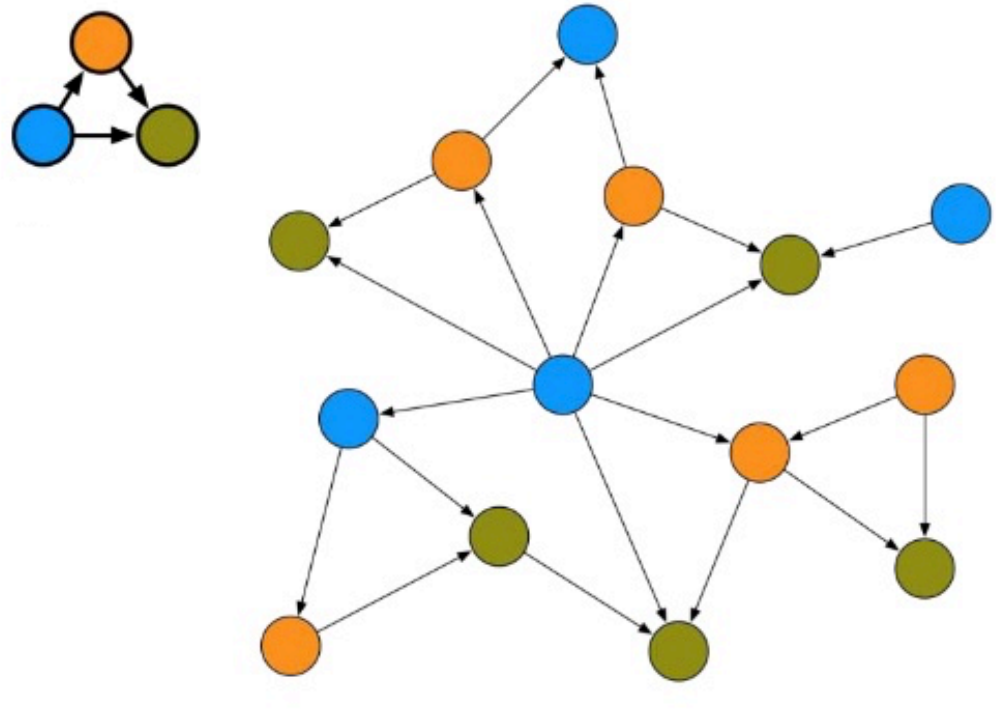
Langage de requête pour les graphes

Pourquoi pas SQL ?

- SQL inefficace pour exprimer des patterns de graphe complexes
- En particulier pour les requêtes
  - récursives
  - pouvant accepter des chemins de longueurs différentes
- Les patterns de graphes sont plus intuitifs et déclaratifs que les jointures
- SQL ne peut pas retourner de valeurs sur les chemins

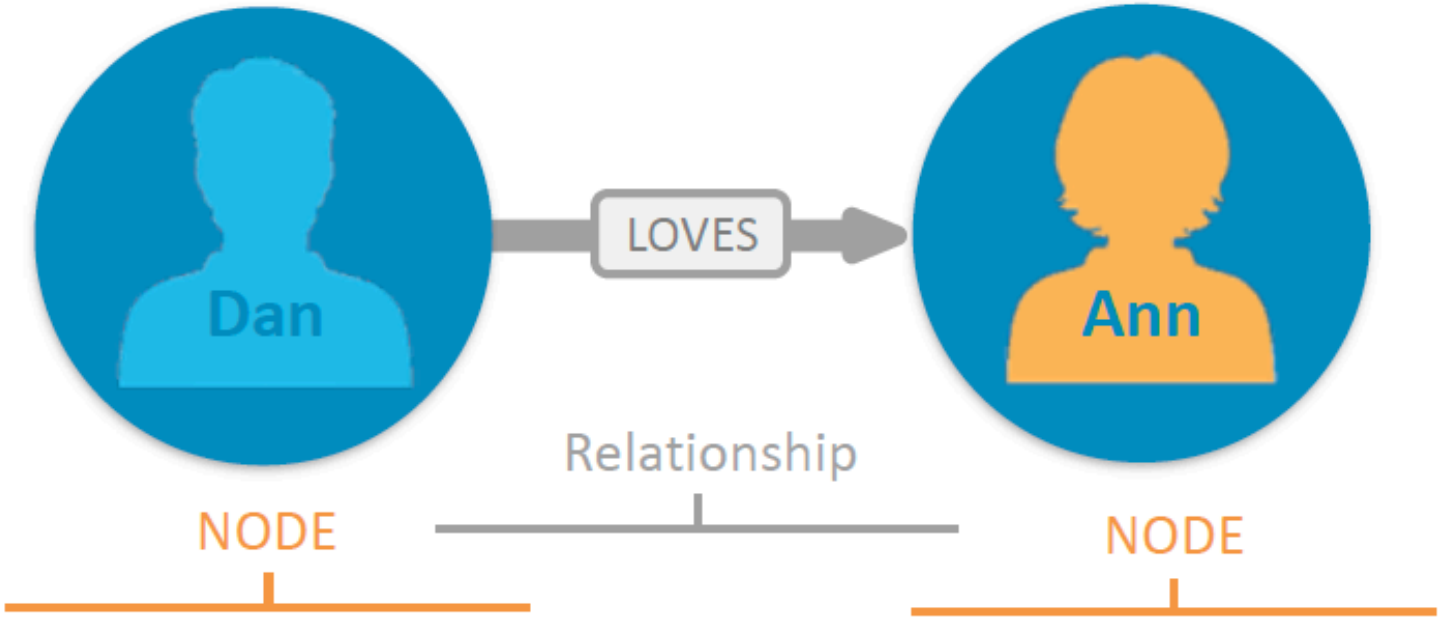
Cypher

- Un langage de requête fait pour les graphes basé sur les patterns de graphe
  - Déclaratif
  - Expressif
  - Pattern matching

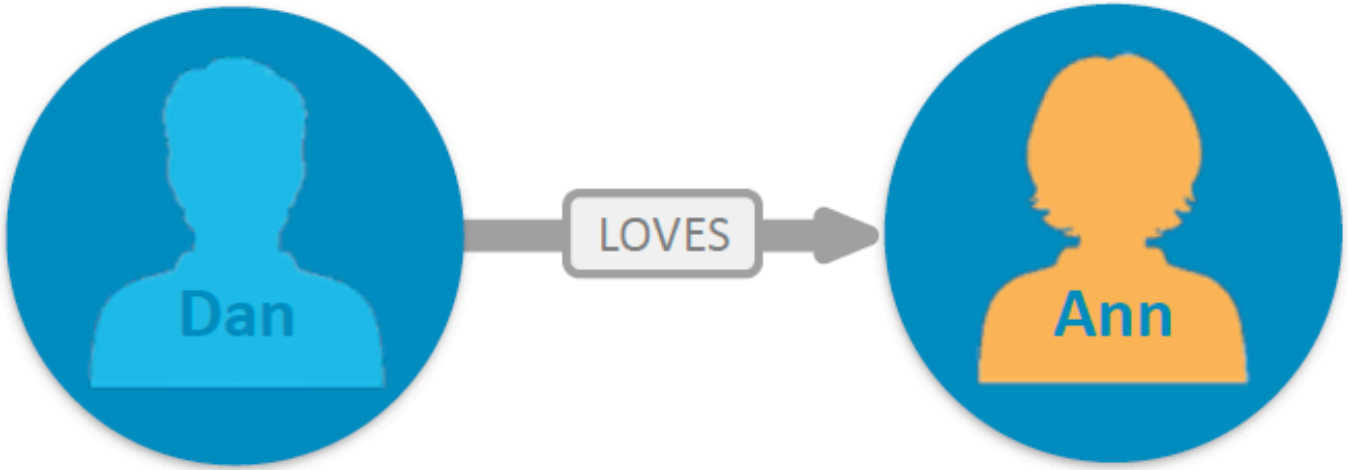


(Pattern matching = « *recherche de motif* » en français)

Les patterns dans notre modèle de graphe



Cypher : expression des patterns de graphe

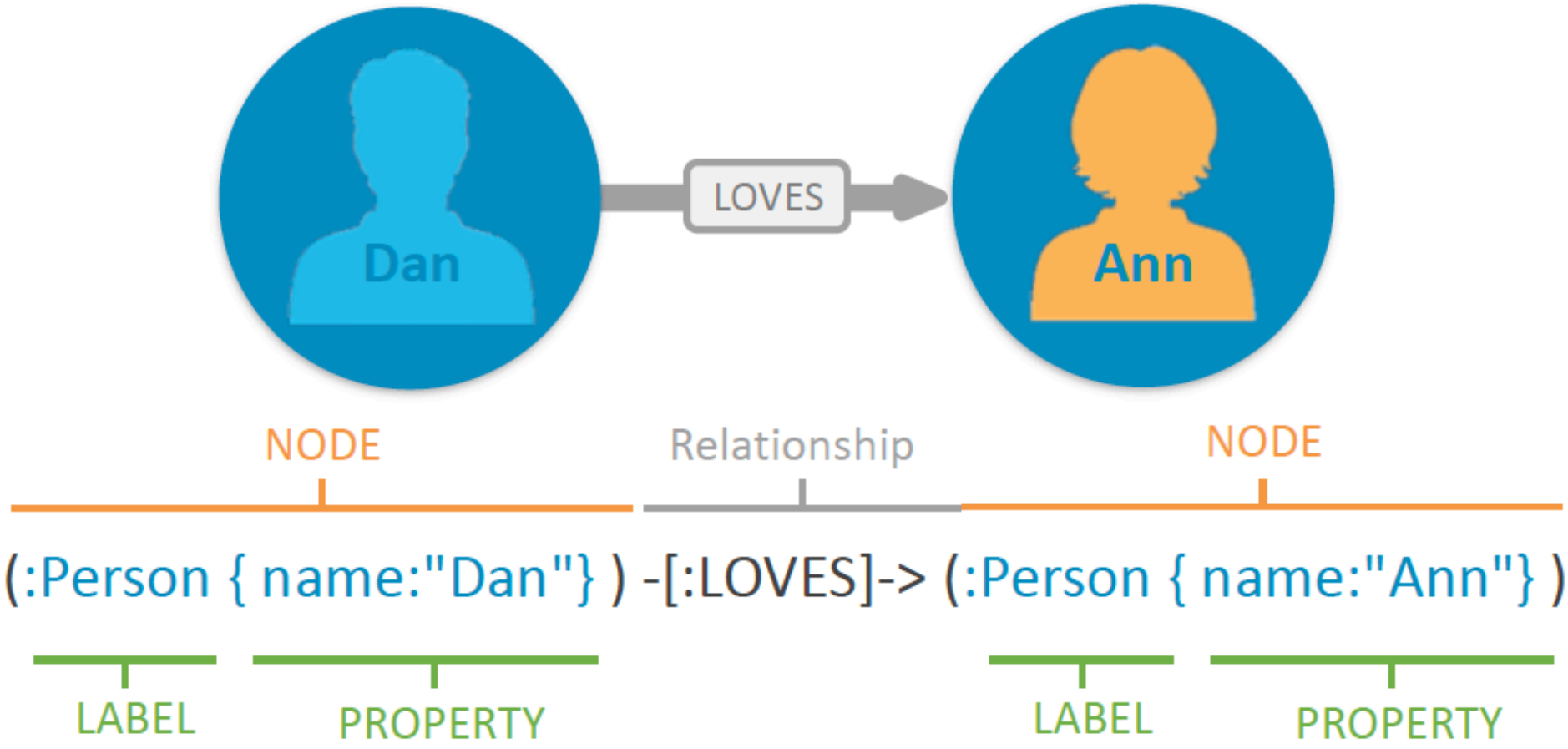


Relationship

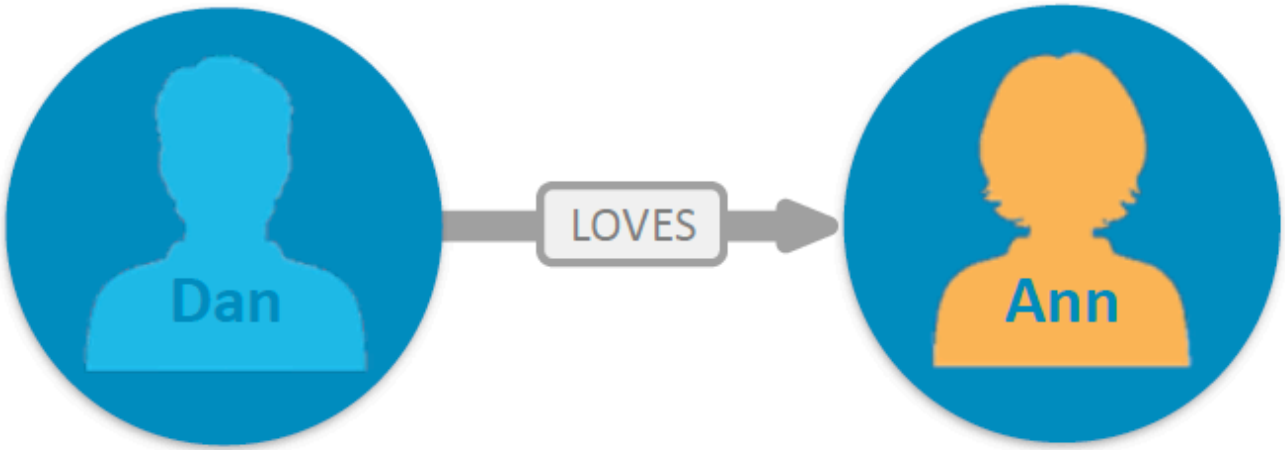
```
(:Person { name:"Dan"} ) -[:LOVES]-> (:Person { name:"Ann"} )
```

LABEL PROPERTY LABEL PROPERTY

Cypher : expression des patterns de graphe



Cypher : création des patterns de graphe

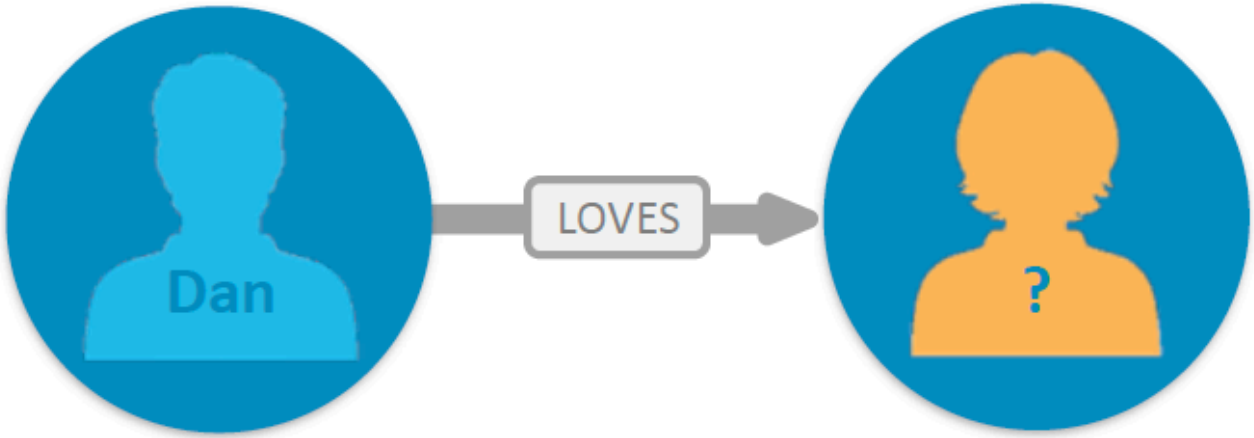


NODE Relationship NODE

```
CREATE (:Person { name:"Dan"} ) -[:LOVES]-> (:Person { name:"Ann"} )
```

LABEL PROPERTY LABEL PROPERTY

Cypher : match des patterns de graphe



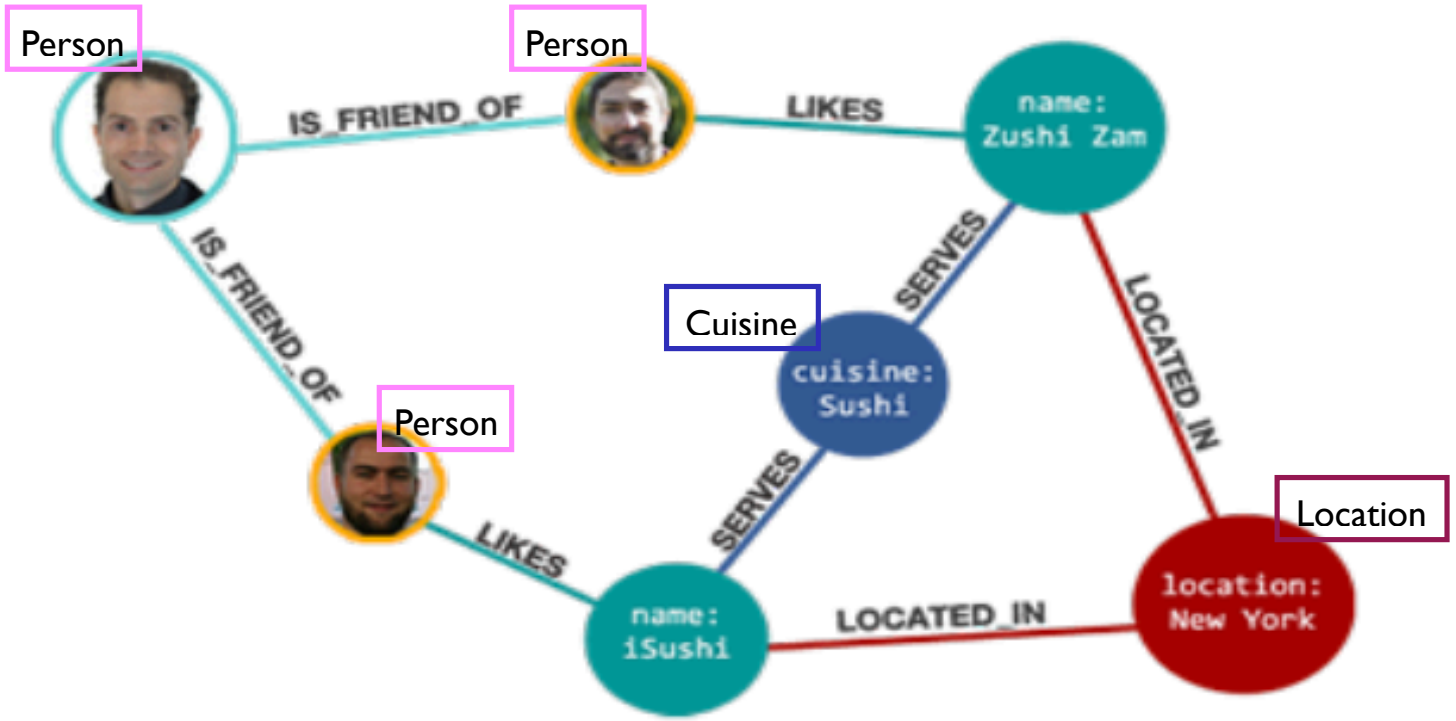
NODE Relationship NODE

```
MATCH (:Person { name:"Dan"} ) -[:LOVES]-> ( whom ) RETURN whom
```

LABEL PROPERTY VARIABLE

Un exemple de requête de graphe

Recommandation et réseaux sociaux



Recommandation et réseaux sociaux


```
MATCH (person:Person)-[:IS_FRIEND_OF]->(friend),
      (friend)-[:LIKES]->(restaurant),
      (restaurant)-[:LOCATED_IN]->(loc:Location),
      (restaurant)-[:SERVES]->(type:Cuisine)
WHERE person.name = 'Philip'
AND loc.location='New York'
AND type.cuisine='Sushi'
RETURN restaurant.name
```

La Syntaxe de Cypher

Les motifs de noeuds

- Les noeuds
  - dessinés avec des parenthèses

( )

Les motifs de noeuds

- Les noeuds
  - dessinés avec des parenthèses, on peut spécifier un type (label)

(:Type)

Les motifs de noeuds

- Les noeuds
  - dessinés avec des parenthèses, on peut spécifier un type
    - Expression de type autorisées :

(:(A&B)&!(B&C))

    - Opérateurs logiques pour les expressions de type :
      - & | !
    - Wildcard (au moins une etiquette) : %

Remarque : avant Neo4J 5 seulement “and” autorisé pour les labels de noeuds et avec une autre syntaxe (:A:B)

Les motifs de relation

- Les relations
  - dessinés avec des flèches, détails entre crochets

```
- - >

-[]->

- [:DIRECTED] ->
```

  - Matchent une seule relation du graphe. Entre crochets les propriétés pour filtrer les relations matchées

Les motifs de relation

- Les relations
  - dessinés avec des flèches, détails entre crochets
  - expressions de type et wildcard autorisés

$$- [:(!R\&!S)|T] ->$$

  - Les expressions de type ont la même syntaxe que l'expressions d'etiquette (pour le noeuds)
    - (mais à difference d'un noeud, une relation a exactement une etiquette)

Les motifs de chemin (simple path patterns)

- Simple path patterns
  - dessinés en connectant les noeuds et les relations avec des tirets,

    la spécification de directions avec > et < est optionnelle

```
( ) - - ( )
( ) - [ ] - ( )
( ) - [ ] - >( )
( ) <- [ ] - ( )
( ) <- [ ] -> ( )
```

Les motifs de chemin simples (simple path patterns)

- Un simple path pattern commence et termine avec un noeud et alterne strictement noeuds et relations
- Il doit avoir au mons un noeud

( ) - [ ] - >( ) - - ( )<- [] - ( ) <- [ ] - ( )

Les variables

- Un pattern cypher peut utiliser des variables pour nommer noeuds, relations et chemins

(n :A) - [r:Directed] - >(m)

- n, r et m sont les variables du patterns, elles precedent les éventuelles etiquettes
  - Elles servent à obtenir une reference aux noeuds et relations matchées
- :A est un label de noeud
- :Directed est un label de relation

Les propriétés

- Les propriétés de noeuds et relations sont décrites entre { }

(n :P {name :'John', office: 34}) - [r:Directed {since :1992}] - >(m)

- n, r et m sont les variables du pattern, elles precedent les éventuelles etiquettes
- :P est un label de noeud
- :Directed est un label de relation
- "name" et "office" sont des propriétés du noeud n, "since" une propriété de la relation r

Les motifs de graphe (graph patterns)

- Un motif de graphe est formé par plusieurs path patterns séparés par ‘,’
- Ils peuvent utiliser des variables en commun

(p : Person) - [r : ACTED_IN] -> (m : Movie)<- [: ACTED_IN] - (c : Person) ,

(m) <- [:DIRECTED]- (d : Person)

Intuitivement : ce motif matche toutes les parties du graphe décrivant un film, un couple d’acteurs de ce film ainsi que le réalisateur de ce film

Les composants d'une requête Cypher

MATCH (m : Movie)

RETURN m

MATCH est suivi d'un graph pattern (le motif à chercher dans le graphe)

RETURN est suivi des parties du pattern (noeuds, relations, chemins et/ou propriétés) à retourner

MATCH et **RETURN** sont des mot clefs Cypher

Les composants d'une requête Cypher

Exemples :

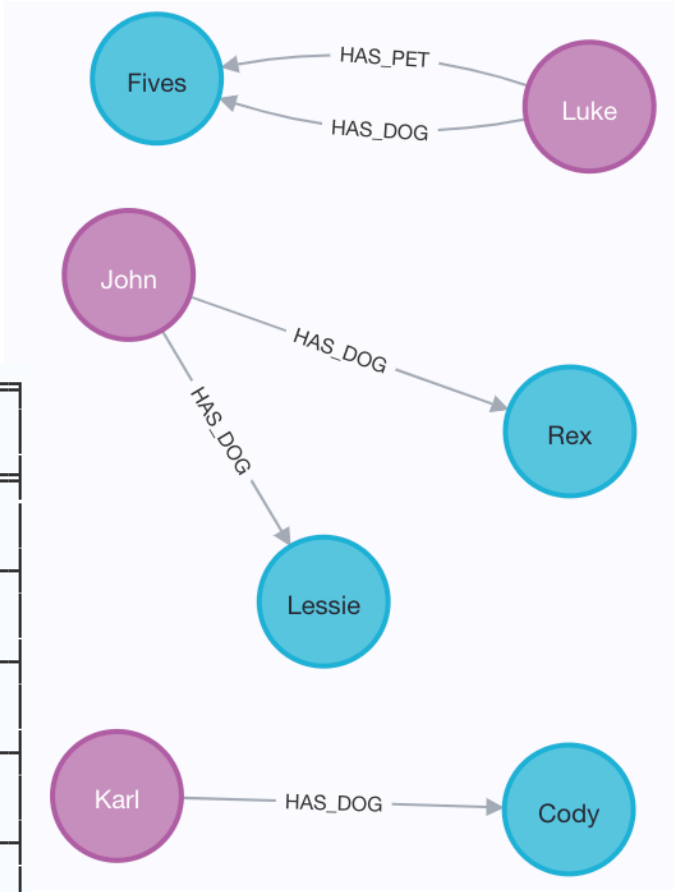
```
MATCH (p : Person) - [r :ACTED_IN] -> (m : Movie)
RETURN p, r, m.title
```

```
MATCH (p : Person) - [r1 :ACTED_IN] -> (m : Movie) ,
(m) <- [r2 :ACTED_IN] -  (c : Person)
RETURN p, r1, c
```

```
MATCH
(p : Person) - [r1 :ACTED_IN] -> (m : Movie) <- [r2 :ACTED_IN] -  (c :
Person), (m) <- [ :DIRECTED]- (d : Person)
RETURN p.name, c.name, r1.fee, d
```

Matchs

- Un match d'un graph pattern P (dont les path patterns sont simples) sur un graphe G est une affectation v des variables de P tel que v(P) est dans G. De plus v(r1)≠v(r2) pout toute paire r1≠r2 de relations
- MATCH P calcule tous les matchs de P dans G



MATCH (person :Person)-[r]->(d:Dog)

d	person	r
(:Dog {name: "Lessie"})	(:Person {name: "John"})	[:HAS_DOG]
(:Dog {name: "Rex"})	(:Person {name: "John"})	[:HAS_DOG]
(:Dog {name: "Cody"})	(:Person {name: "Karl"})	[:HAS_DOG]
(:Dog {name: "Fives"})	(:Person {name: "Luke"})	[:HAS_DOG]
(:Dog {name: "Fives"})	(:Person {name: "Luke"})	[:HAS_PET]

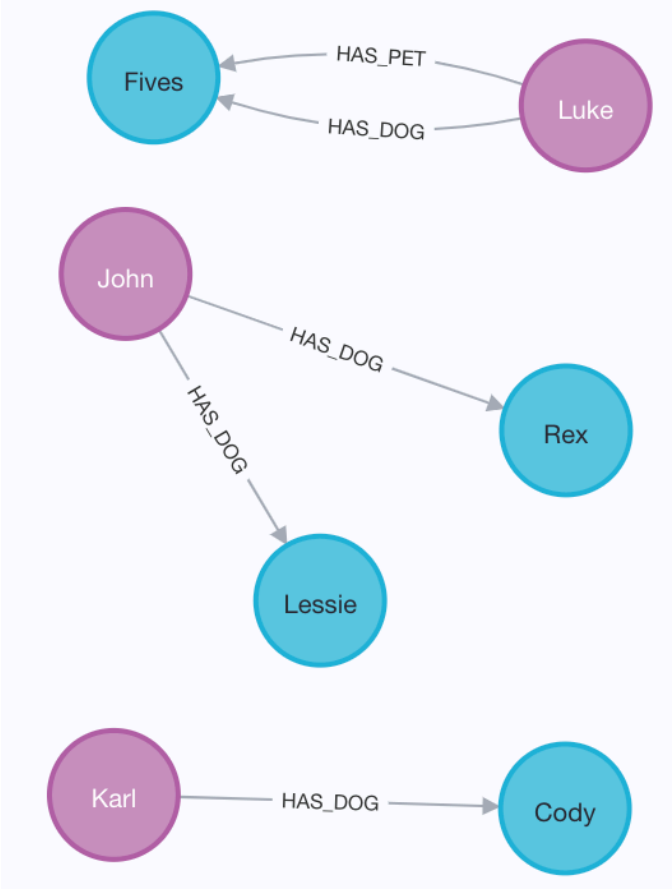
Matchs : variables anonymes

- Les variables sans nom se comportent comme les variables avec nom (toutes distinctes)

MATCH (person :Person)-[]->(:Dog)

RETURN *

person
(:Person {name: "John"})
(:Person {name: "John"})
(:Person {name: "Karl"})
(:Person {name: "Luke"})
(:Person {name: "Luke"})



Remarques : doublons pas éliminés dans les résultats retournés

* retourne toutes les variables

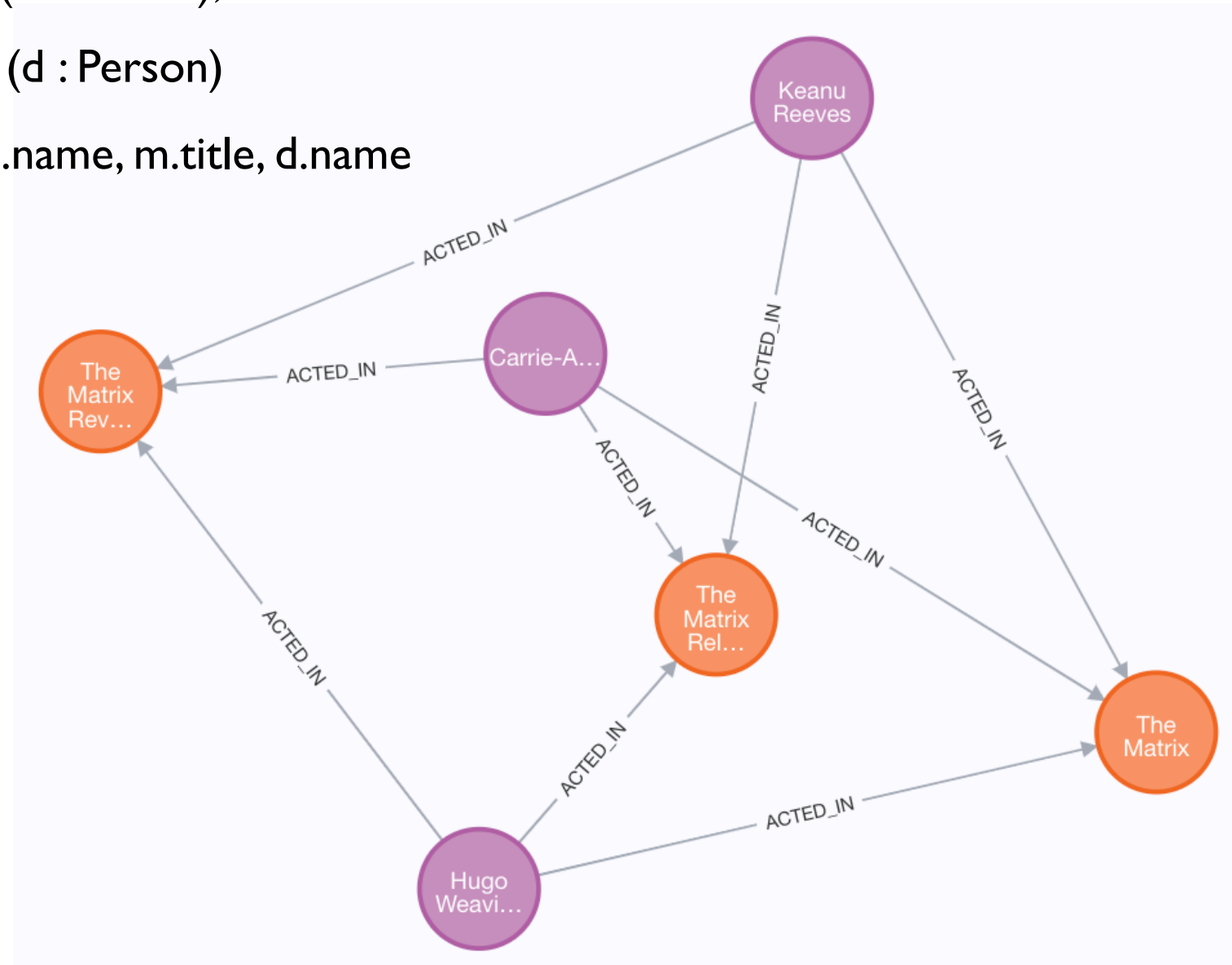
Matchs : sémantique “Trail”

MATCH (p : Person) - [r1 :ACTED_IN] -> (m : Movie)

<- [r2 :ACTED_IN] - (c : Person),

(m) <- [:ACTED_IN]- (d : Person)

RETURN p.name, c.name, m.title, d.name



Matchs : sémantique “Trail”

MATCH (p : Person) - [r1 :ACTED_IN] -> (m : Movie)

<- [r2 :ACTED_IN] - (c : Person),

(m) <- [:ACTED_IN]- (d : Person)

RETURN p.name, c.name, m.title, d.name

p.name	c.name	m.title	d.name
"Keanu Reeves"	"Carrie-Anne Moss"	"The Matrix"	"Hugo Weaving"
"Carrie-Anne Moss"	"Keanu Reeves"	"The Matrix"	"Hugo Weaving"
"Keanu Reeves"	"Hugo Weaving"	"The Matrix"	"Carrie-Anne Moss"
"Hugo Weaving"	"Keanu Reeves"	"The Matrix"	"Carrie-Anne Moss"
"Carrie-Anne Moss"	"Hugo Weaving"	"The Matrix"	"Keanu Reeves"
"Hugo Weaving"	"Carrie-Anne Moss"	"The Matrix"	"Keanu Reeves"

Matchs : sémantique “Trail”

Remarquer qu’on n’obtient pas la ligne

"Keanu Reeves"	"Carrie-Anne Moss"	"The Matrix”	"Keanu Reeves"

Ni la ligne

"Keanu Reeves"	"Keanu Reeves”	"The Matrix”	"Hugo Weaving"

Car ces matchs nécessiteraient d’utiliser plusieurs fois la même relation (pour matcher différentes parties du graph pattern)

Matchs : sémantique “Trail”

Donc par défaut dans Cypher les matchs où la même relation est incluse plusieurs fois ne sont pas retenus (*relationships isomorphism*)

Raisons :

- réduction de la taille des résultats
- élimination des traversées infinies

D’autres sémantiques existent. (c.f., manuel Cypher p8)

Les composants d'une requête Cypher

MATCH path = (p : Person) - [: HAS_DOG] - > ()

RETURN path

path est une variable de chemin

path
(:Person {name: "John"})-[:HAS_DOG]->(:Dog {name: "Lessie"})
(:Person {name: "John"})-[:HAS_DOG]->(:Dog {name: "Rex"})
(:Person {name: "Karl"})-[:HAS_DOG]->(:Dog {name: "Cody"})
(:Person {name: "Luke"})-[:HAS_DOG]->(:Dog {name: "Fives"})

Valeur retournés

- On peut retourner des noeuds

```
MATCH (m : Movie)
RETURN m
```

- Des relations

```
MATCH  (person :Person )-[r:HAS_DOG]->(d:Dog)
RETURN r
```

- Des propriétés

```
MATCH (m : Movie)
RETURN m.title, m.released
```

On accède aux propriétés avec {variable}.{property_key}

Valeur retournés

- Des chemins

```
MATCH path = (p : Person) - [] - > (d : Dog)
RETURN path
```

- Toutes les variables (de noeuds, relation et chemin) de la requête

```
MATCH path = (p : Person) - [] - > ()
RETURN *
```

Valeur retournés

```
MATCH path = (p : Person) - [] - > ()
RETURN *
```

p	path
(:Person {name: "John"})	(:Person {name: "John"})-[:HAS_DOG]->(:Dog {name: "Lessie"})
(:Person {name: "John"})	(:Person {name: "John"})-[:HAS_DOG]->(:Dog {name: "Rex"})
(:Person {name: "Karl"})	(:Person {name: "Karl"})-[:HAS_DOG]->(:Dog {name: "Cody"})
(:Person {name: "Luke"})	(:Person {name: "Luke"})-[:HAS_DOG]->(:Dog {name: "Fives"})
(:Person {name: "Luke"})	(:Person {name: "Luke"})-[:HAS_PET]->(:Dog {name: "Fives"})

Valeur retournés

- Des fonctions peuvent être appliquées aux valeurs retournés (cf. https://neo4j.com/docs/cypher-manual/current/functions/)

```
MATCH path = (p : Person) - [r] - > ()
RETURN length(path), type(r)
```

- Plus en general n'importe quelle expression peut être retournée
  - Expression : constantes, variables, propriétés, appel de fonction, expression arithmétique, motifs de chemin, etc…

    MATCH (m:Movie)

    RETURN "Early release : ", m.released < 2012

  - cf. https://neo4j.com/docs/cypher-manual/current/queries/expressions/

Sensibilité à la casse

Sensible à la casse

- les labels de noeud
- Les types de relation
- Les clefs de propriété

Insensible à la casse

- les mots clefs Cypher

Sensibilité à la casse

Sensible à la casse

- : Person
- :ACTED_IN
- name

Insensible à la casse

- MaTcH
- return

L’écriture de requêtes

La sous-clause WHERE

```
MATCH (m : Person)
WHERE m.age > 60 XOR m.country = 'France'
RETURN m.name
```

Fait partie de MATCH, (ainsi que de OPTIONAL MATCH et WITH, cf. plus loin) :

- ajoute des contraintes au pattern
- seulement les matchs qui satisfont le WHERE sont retenus

Connecteurs logiques : NOT, OR, XOR, AND

Tests booléens : =, >,<, <=, >=, expressions régulières (=~ 'regexp'), fonctions booléennes (e.g. STARTS, CONTAINS, ENDS WITH,…), prédicat de label ("x : Label"), IS (NOT) NULL, prédicat de chemin, etc…

La sous-clause WHERE

Fonctions booléennes (sur les chaînes de caractères)

```
MATCH (m : Person)
WHERE m.name STARTS WITH 'A'
RETURN m.name
```

La sous clause WHERE

Fonctions booléennes (sur les chaînes de caractères)

```
MATCH (m : Person)
WHERE m.name ENDS WITH 'li'
RETURN m.name
```

La sous clause WHERE

Fonctions booléennes (sur les chaînes de caractères)

```
MATCH (m : Person)
WHERE m.name CONTAINS 'li'
RETURN m.name as nom
```

La sous clause WHERE

Expressions régulières

```
MATCH (m : Person)
WHERE m.name =~ 'Am.*'
RETURN m.name
```

Hérité de la syntaxe des expressions régulières Java

Flags inclus, par exemple (?i) insensibilité à la casse

```
MATCH (m : Person)
WHERE m.name =~ '(?i)Am.*'
RETURN m.name
```

La sous-clause WHERE

prédicat de label (“:”) :

```
MATCH (m)
WHERE m : Person
RETURN m.name
```

```
MATCH (m)
WHERE m : Employee  : Student
RETURN m.name
```

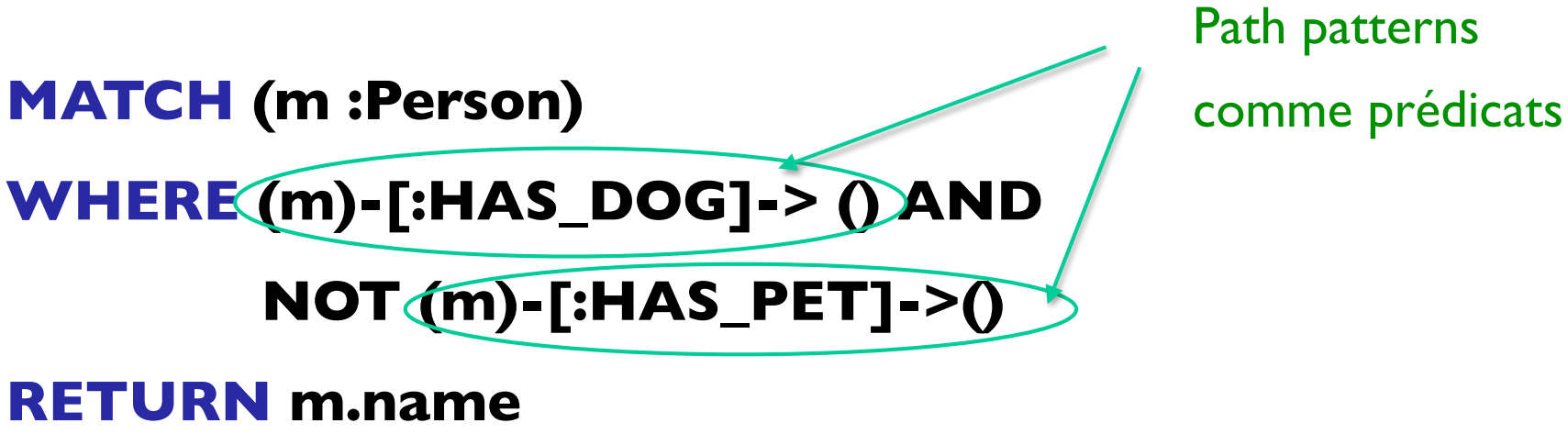
La sous-clause WHERE

- IS (NOT) NULL vérifie si une propriété existe (ou pas)

```
MATCH (m :Client)
WHERE m.email IS NOT NULL
RETURN m.email
```

La sous-clause WHERE

- **Prédicat de chemin** : un path pattern dans la clause WHERE est considéré un prédicat booléen :



- Il ne peut pas introduire de nouvelles variables : (m)-[:HAS_DOG]-> (x)
- Si p est un path pattern qui utilise les variables $m_1 \ldots m_k$
  p retourne vrai pour $m_1 = v_1, \ldots, m_k = v_k$ si $m_1 = v_1 \ldots m_k = v_k$ est un match pour p dans le graphe

La sous-clause WHERE

- WHERE peut aussi être utilisé dans un élément (noeud ou relation) d'un pattern :

```
MATCH (a:Person)-[r:HAS_DOG  WHERE r.since < 2009]->(b:Dog)
RETURN r.since
```

Dans ce cas la condition de WHERE peut seulement mentionner les variables du noud/relation dans lequel il se trouve

Les sous clauses SKIP et LIMIT

```
MATCH (m : Person)
WHERE m.name CONTAINS 'li'
RETURN m.name as nom
LIMIT 5
```

Sous-clauses de RETURN (ou de WITH, cf. plus loin)

5 premiers résultats seulement

Limite le branchement d'un chemin de recherche

Les sous clauses SKIP et LIMIT

```
MATCH (m : Person)
WHERE m.name CONTAINS 'li'
RETURN m.name as nom
SKIP 5
```

A partir du 6ème résultat

Utile en combinaison avec LIMIT pour l'affichage de résultats page par page

Les sous clauses SKIP et LIMIT

```
MATCH (m : Person)
WHERE m.name CONTAINS 'li'
RETURN m.name as nom
SKIP 5
LIMIT 2
```

Le 6ème et 7ème résultat

La sous clause ORDER BY

```
MATCH (m : Person)
WHERE m.name CONTAINS 'li'
RETURN m.name as nom
ORDER BY m.name DESC
```

Suit RETURN (ou WITH), trie les résultats avant de les retourner

Modificateur par défaut : ASC (ordre croissant)

DESC : ordre décroissant

Souvent utile en combinaison avec LIMIT

La clause CREATE

```
CREATE (m : Movie {title : 'Mystic River', released : 2003})
RETURN m
```

- Ajoute au graphe les noeuds et relations spécifiées
- Les variables servent à garder une référence aux elements créés, pour les utiliser plus loin
- Meme syntaxe que la clause MATCH mais pas de relations / path pattern quantifiés, ni de relations de longueur variable (cf plus loin)

La clause CREATE

```
MATCH (m : Movie {title : 'Mystic River'}),
      (p : Person {name : 'Kevin Bacon'})
CREATE (p) - [r : ACTED_IN {roles: ['Sean']}] -> (m)
RETURN p, r, m
```

La clause DELETE

Attention ! A n'utiliser que si le noeud n'intervient dans aucune relation.

```
MATCH (p : Person {name : 'Your Name'})
DELETE p
```

Pour supprimer également toutes les relations associées :

```
MATCH (p : Person {name : 'Your Name'})
DETACH DELETE p
```

La clause DELETE

Pour supprimer une relation (mais pas les noeuds y participant) :

```
MATCH (p {name : 'Andy'}) - [r: KNOWS]-> ()
DELETE r
```

Pour supprimer tout le contenu de la base de données :

```
MATCH (p)
DETACH DELETE p
```

La clause SET

MATCH (m : Movie {title : 'Mystic River'})

SET m.tagline = 'We bury our sins here, Dave. We wash them clean.'

RETURN m

- Modifie chaque match comme spécifié
- Peut modifier
  - sur les noeud : etiquettes et propriétés
  - sur les relations : propriétés

La clause SET

- Ajouter des etiquettes de noeud :

```
match (n:Person {name:"John"} )
SET n:User
return labels(n)
```

labels (n)
["Person", "User"]

- Ajouter des propriétés :

```
match (n:Person {name:"John"} )
SET n += {age: 38}
return n
```

n
(:Person:User {name: "John",age: 38})

- Supprimer des propriétés

```
match (n:Person {name:"John"} )
SET n.age = null
return n
```

n
(:Person:User {name: "John"})

La clause SET

- Remplacer toutes les des propriétés :

```
match (n:Person {name:”John"} )
SET n = {firstName: ”John"}
return n
```

n
(:Person:User {firstName: "John"})

La clause REMOVE

```
MATCH (p : Person {name : 'Your Name'})
REMOVE p.age
RETURN p.name, p.age
```

Pour supprimer seulement la propriété âge,

la valeur retournée pour p.age sera alors nulle

La clause REMOVE

Attention, REMOVE ne peut pas être utilisé pour supprimer toutes les propriétés, à la place :

```
MATCH (p : Person {name : 'Your Name'})
SET p={}
RETURN p.name, p.age
```

La clause REMOVE

```
MATCH (p : Person {name : 'Your Name'})
REMOVE p:German:Swedish
RETURN p.name, labels(p)
```

Pour supprimer des labels sur un noeud.

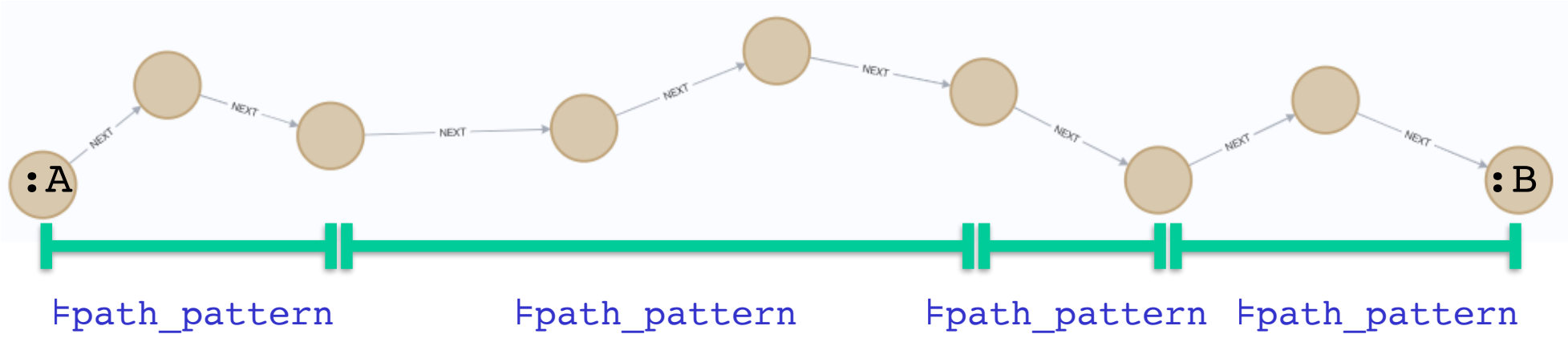
L'ensemble de labels retourné sera vide [ ]

Path patterns quantifiés

- La version 5.9 de Cypher a introduit les ‘quantified path patterns’ et les 'quantified relationships’
- Path pattern quantifié :

```
(p:A)  (path_pattern){min..max}  (m:B)
```

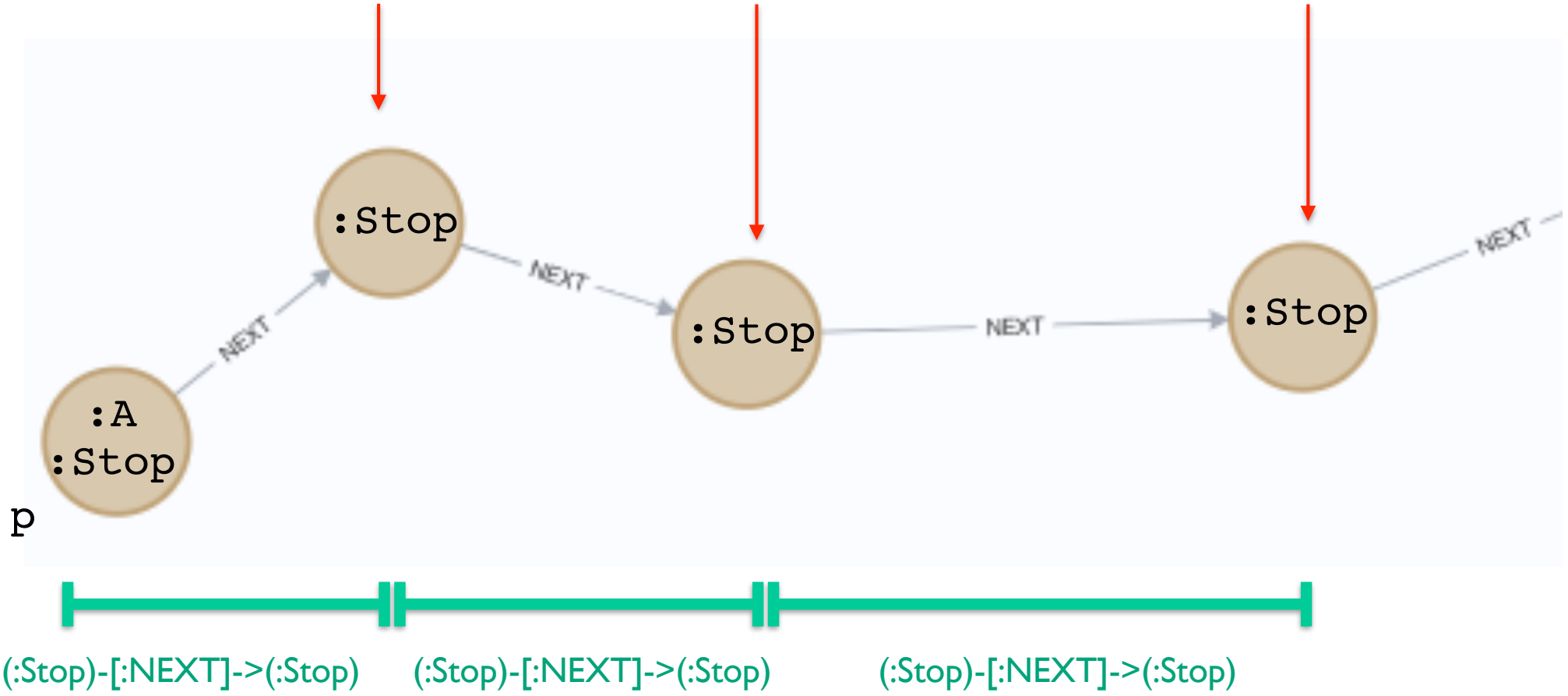
  - Un chemin dans le graphe est un match pour ce path pattern quantifié si le noeud de depart a etiquette :A, le noeud d’arrivée a etiquette :B et ils sont reliés par la concatenation de n chemins, $min \leq n \leq max$, où chaque chemin est un match pour `path_pattern`
  - `Sémantique trail :` pas de repetition de relation dans un match d’un path pattern quantifié ; pas de repetition de relation dans les matchs des différents path patterns d’un graph pattern



Path patterns quantifiés

- **MATCH** (p:A) ((:Stop)-[:NEXT]->(:Stop)){1,3} (m)
- **RETURN** m

Noeuds retournés



Path patterns quantifiés

Variant	Description
{m,n}	Between m and n iterations.
+	1 or more iterations.
*	0 or more iterations.
{n}	Exactly n iterations.
{m,}	m or more iterations.
{,n}	Between 0 and n iterations.
{,}	0 or more iterations.

Path patterns quantifiés

- On peut avoir une alternance quelconque de path pattern simples et path patterns quantifiés (mais pas deux path patterns simples de suite) :

(:C)-[]->(:D) ((:A)-[:R]->(:B)){2} ((:X)<--(:Y)){1,2} (:E)-[:R]->(:F)

est un path pattern et est equivalent à l'union des deux path patterns :

(:C)-[]->(:D&A)-[:R]->(:B&A) -[:R]->(:B&X)<--(:Y&E)-[:R]->(:F)

(:C)-[]->(:D&A)-[:R]->(:B&A) -[:R]->(:B&X)<--(:Y&X)<--(:Y&E)-[:R]->(:F)

Relations quantifiées

- Relations quantifiées : un raccourci pour les path patterns quantifiés qui ne spécifient que la repetition d'un motif de relation :
- Relation quantifiée : Motif de relation suivi d'un quantificateur (même quantificateurs que les path patterns quantifiés)
  - Peut remplacer une motif de relation dans un path pattern simple
- Path pattern simple :
  - (:A)-[r:Route{type:'autoroute'}]->(:B)
- Path pattern avec relation quantifiée :

  (:A)-[r:Route{type:'autoroute'}]->{2,3}(:B)

  Est equivalent à :

  (:A) (()-[r:Route{type:'autoroute'}]->()){2,3} (:B)

Path patterns quantifiés et relations quantifiées

- Variables de groupe
  - En utilisant des listes on peut avoir des references aux noeuds intermédiaires matchés par un path pattern quantifié (cf. plus loin)

Relations de longueur variable

- Avant Neo4J 5.9 la seule façon de représenter des motifs de longueur variable
- Les relations de longueur variables sont maintenues pour compatibilité
  - Les relations quantifiées peuvent exprimer plus
- Syntaxe différente :

```
(:A)-[r :Route*..5 {type:'autoroute'}]->(:B)
```

est une relation de longueur variable équivalente à :

```
(:A)-[r:Route{type:'autoroute'}]->{1,5}(:B)
```

- Nécessaire de les utiliser dans des contextes où les quantified path patterns ne sont pas encore supportés (e.g dans shortestPath cf. Plus loin)

Relations de longueur variable

Quantificateur de longueur	Description
*	1 or more iterations.
*n	Exactly n iterations.
*m..n	Between m and n iterations.
*m..	m or more iterations.
*..n	Between 1 and n iterations.

quantificateur de longueur entre l'expression d'etiquette et l'expression de proprieté-valeur :

(a)-[r :Route*..5 {type: 'autoroute'}]->(b)

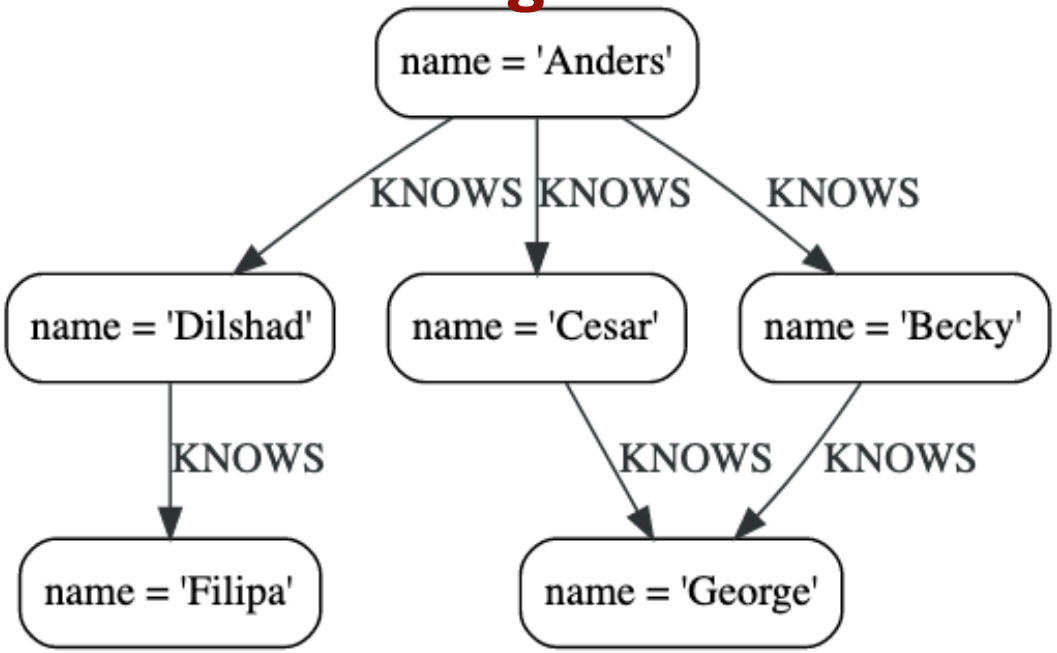
(a) et (b) matchent deux noeuds dans le graphe connectés par un chemin d'au moins 1 relation et au plus 5, tel que toutes les relations dans le chemin ont etiquette 'Route' et propriété 'type' égale à 'autoroute'

Relations de longueur variable

```
MATCH (me) - [*min...max] - > (autre_noeud)
WHERE me.name = `Filipa'
RETURN autre_noeud
```

```
MATCH (me) - [: knows*] - > (remote_friend)
WHERE me.name = `Filipa'
RETURN remote_friend
```

Relations de longueur variable



Query **cypher** Copy to Clipboa

```
MATCH (me)-[:KNOWS*1..2]-(remote_friend)
WHERE me.name = 'Filipa'
RETURN remote_friend.name
```

Table 1. Result

remote_friend.name
"Dilshad"
"Anders"

Plus courts chemins

`allShortestPaths(path_pattern)`

MATCH (m: Person{name: 'Martin Sheen'}),(o: Person{name : 'Oliver Stone'}),

p = allShortestPaths((m) -[*] - (o))

RETURN p

allShortestPaths restreint les matchs possibles de (m) -[*] - (o) : un match (chemin) est gardé seulement s'il est le plus court entre son origine et sa destination

- Taille d'un match (chemin) : nombre de relations
- Remarque : il peut y avoir plusieurs plus courts chemins
- path_pattern ne peut pas être arbitraire : une seule relation (typiquement de longueur variable)

Plus courts chemins

`allShortestPaths(path_pattern)`

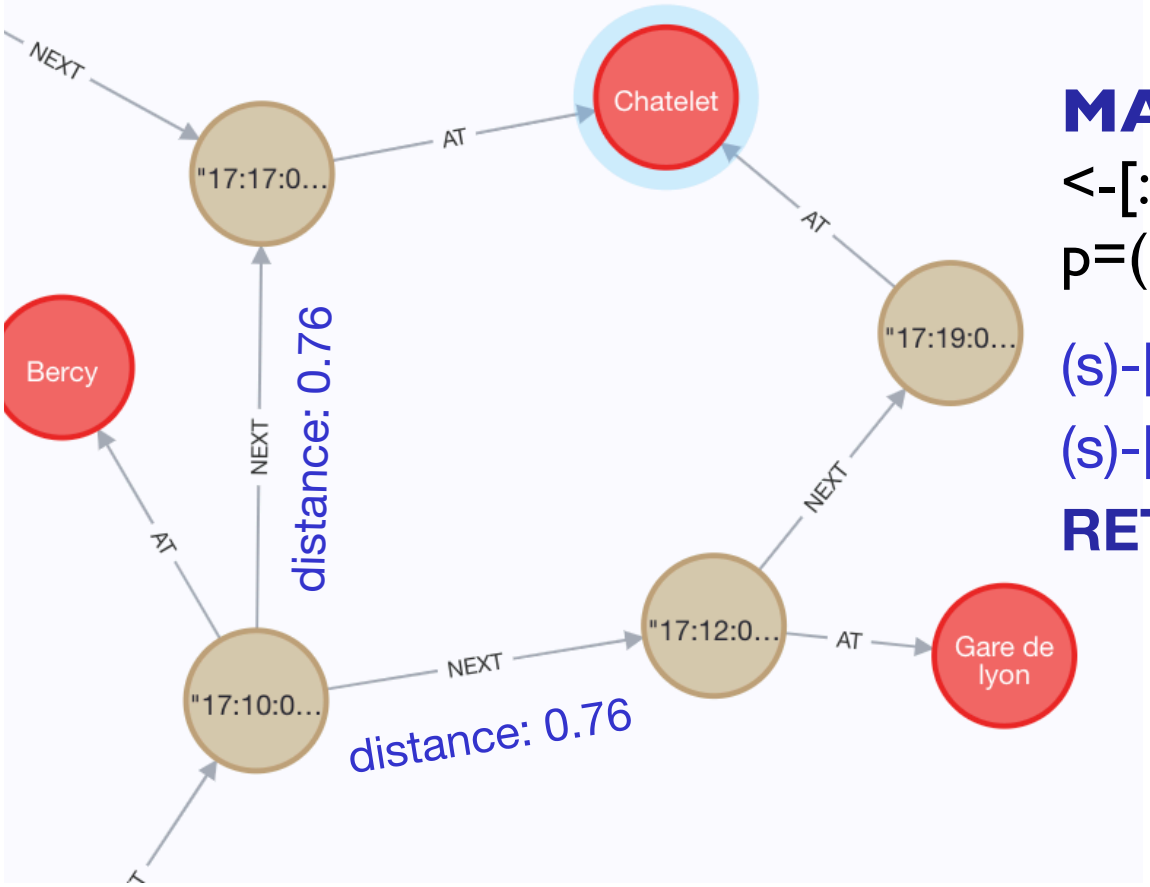
MATCH (m: Person{name: 'Martin Sheen'}),(o: Person{name : 'Oliver Stone'}),

p = allShortestPaths((m) -[*] - (o))

RETURN p

- Remplace la sémantique Trail pour path_pattern
  - Toutefois un plus court chemin est aussi un trail (pas de repetition de relation)
- Mais repetition de relation possible entre le plus court chemin et les chemins qui matchent les autres path patterns de la requête

Plus courts chemins

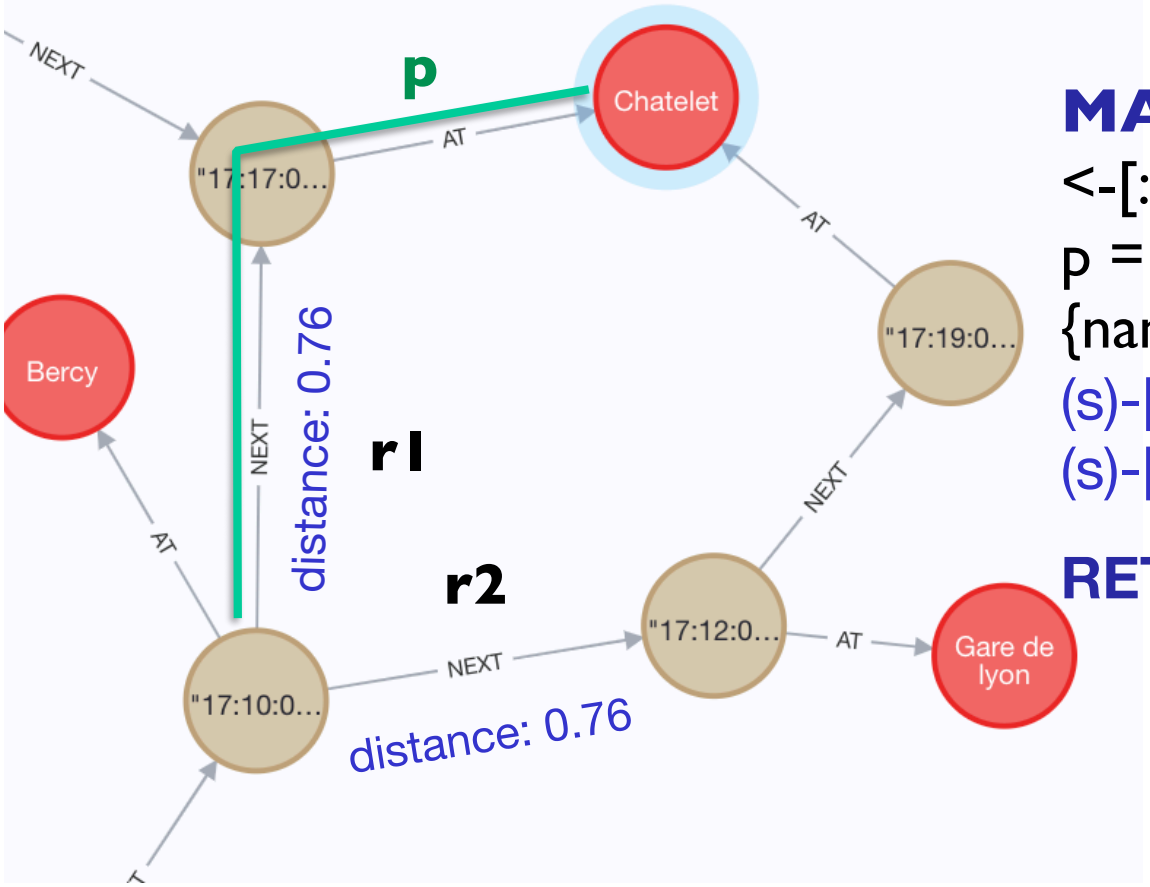



```
MATCH (: Station {name:"Bercy"} )
<-[:AT]-(s),
p=(s)-[*]- (t: Station {name:"Chatelet"} ) ,
(s)-[r1:NEXT {distance: 0.76}]-(),
(s)-[r2:NEXT {distance: 0.76}]-()
RETURN p, r1,r2
```

Resultat vide

- Les deux derniers chemins doivent partir de Bercy et parcourir un tronçon de distance 0.76
- Le premier chemin doit aller de Bercy à Châtelet : tous les chemins de Bercy à Châtelet passent par un tronçon de distance 0.76
- Pas de solution dans la sémantique "Trail" (trois tronçons distincts)

Plus courts chemins




```
MATCH (: Station {name:"Bercy"} )
<-[:AT]-(s),
p = allShortestPaths( (s)-[*]- (t: Station
{name:"Chatelet"} ) ) ,
(s)-[r1:NEXT {distance: 0.76}]-(),
(s)-[r2:NEXT {distance: 0.76}]-()

RETURN p, r1, r2
```

Resultat non vide

Plus court chemin

```
MATCH (m: Person{name: 'Martin Sheen'}),(o: Person{name : 'Oliver Stone'}),
p = shortestPath((m) -[*…15] - (o))
RETURN p
```

Un seul plus court chemin est gardé parmi tous les matchs avec la même origine et la même destination (choisi arbitrairement, s'il il y en a plusieurs)

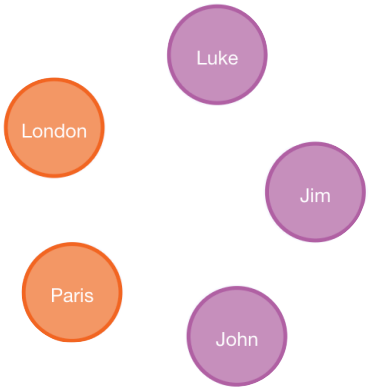
Composition de clauses

- Une requête Cypher est obtenue par enchainement (*chaining*) de plusieurs clauses

```
MATCH (p: Person)
Where p.name STARTS with “J”

MATCH (c: City)
WHERE c.name = p.city

CREATE (p)-[r:LIVES_IN]->(c)

REMOVE p.city

RETURN p, c, r
```

Composition de clauses

- La première clause a pour input l'état du graphe et produit une table : chaque ligne de la table spécifie une assignation des variables de la clause



MATCH (p: Person)

WHERE p.name STARTS with "J")

.

.

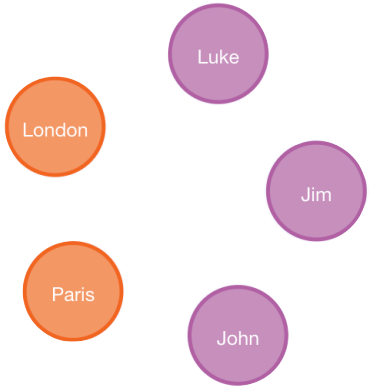
.

p
(:Person {city: "London",name: "John"})
(:Person {city: "Paris",name: "Jim"})

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.

```
MATCH (p: Person)
WHERE p.name STARTS with "J"
```



p
(:Person {city: "London",name: "John"})
(:Person {city: "Paris",name: "Jim"})

C {

```
MATCH (c: Place)
WHERE c.name = p.city
```

.

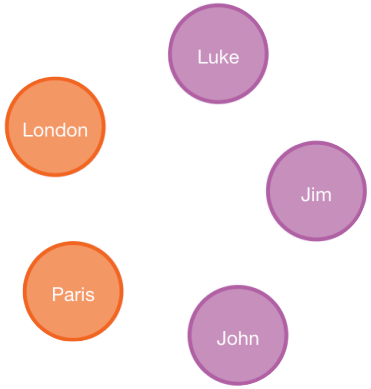
.

.

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.

```
MATCH (p: Person)
WHERE p.name STARTS with "J"
```



p
(:Person {city: "London",name: "John"})
(:Person {city: "Paris",name: "Jim"})

C

```
MATCH (c: Place)
WHERE c.name = p.city
```

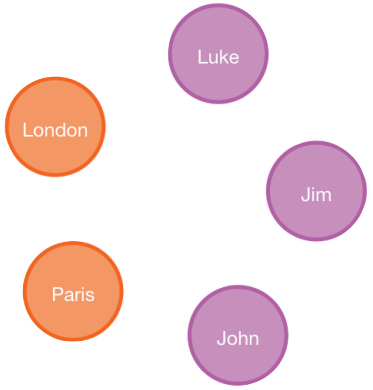
.
.
.

p	c
(:Person {city: "London",name: "John"})	(:Place {name: "London"})
(:Person {city: "Paris",name: "Jim"})	(:Place {name: "Paris"})

- En sortie : La table résultat est étendue avec les nouvelles variables de C. Chaque ligne de l'ancienne table est concatenée avec les résultats de C pour cette ligne

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.



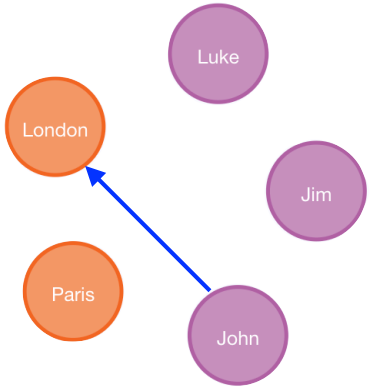

```
MATCH (p: Person)
WHERE p.name STARTS with "J"
MATCH (c: Place)
WHERE c.name = p.city
```

p	c
(:Person {city: "London",name: "John"})	(:Place {name: "London"})
(:Person {city: "Paris",name: "Jim"})	(:Place {name: "Paris"})

C { **CREATE (p)-[r: LIVES_IN]->(c)**

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.



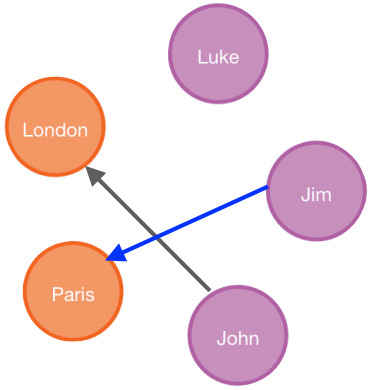

```
MATCH (p: Person)
WHERE p.name STARTS with "J"
MATCH (c: Place)
WHERE c.name = p.city
```

p	c
(:Person {city: "London",name: "John"})	(:Place {name: "London"})
(:Person {city: "Paris",name: "Jim"})	(:Place {name: "Paris"})

C { **CREATE (p)-[r: LIVES_IN]->(c)**

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.



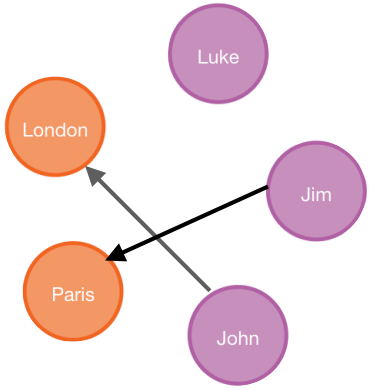

```
MATCH (p: Person)
WHERE p.name STARTS with "J"
MATCH (c: Place)
WHERE c.name = p.city
```

p	c
(:Person {city: "London",name: "John"})	(:Place {name: "London"})
(:Person {city: "Paris",name: "Jim"})	(:Place {name: "Paris"})

C { **CREATE (p)-[r: LIVES_IN]->(c)**

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.




```
MATCH (p: Person)
WHERE p.name STARTS with "J"
MATCH (c: Place)
WHERE c.name = p.city
```

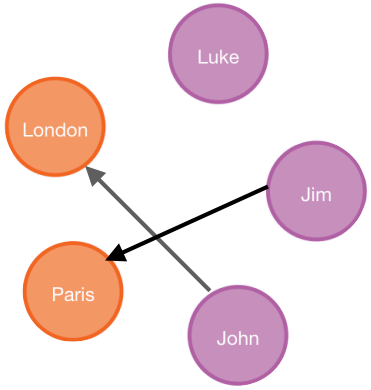
p	c
(:Person {city: "London",name: "John"})	(:Place {name: "London"})
(:Person {city: "Paris",name: "Jim"})	(:Place {name: "Paris"})

C { **CREATE (p)-[r: LIVES_IN]->(c)**

p	c	r
(:Person {city: "London",name: "John"})	(:Place {name: "London"})	[:LIVES_IN]
(:Person {city: "Paris",name: "Jim"})	(:Place {name: "Paris"})	[:LIVES_IN]

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.




```
MATCH (p: Person)
WHERE p.name STARTS with "J"
MATCH (c: Place)
WHERE c.name = p.city
CREATE (p)-[r: LIVES_IN]->(c)
```

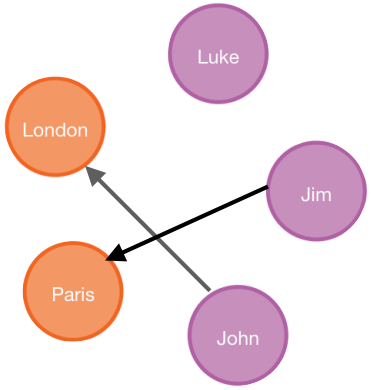
p	c	r
(:Person {city: "London",name: "John"})	(:Place {name: "London"})	[:LIVES_IN]
(:Person {city: "Paris",name: "Jim"})	(:Place {name: "Paris"})	[:LIVES_IN]

C { **REMOVE p.city**

p	c	r
(:Person {name: "John"})	(:Place {name: "London"})	[:LIVES_IN]
(:Person {name: "Jim"})	(:Place {name: "Paris"})	[:LIVES_IN]

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.




```
MATCH (p: Person)
WHERE p.name STARTS with "J"
MATCH (c: Place)
WHERE c.name = p.city
CREATE (p)-[r: LIVES_IN]->(c)
REMOVE p.city
```

C { **RETURN p, c, r**

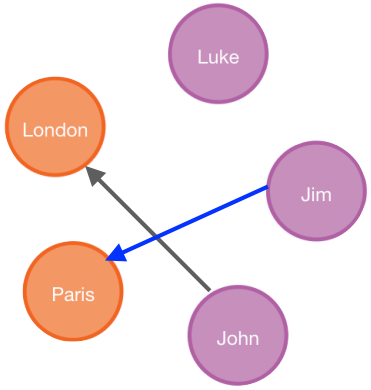
p	c	r
(:Person {name: "John"})	(:Place {name: "London"})	[:LIVES_IN]
(:Person {name: "Jim"})	(:Place {name: "Paris"})	[:LIVES_IN]

p	c	r
(:Person {name: "John"})	(:Place {name: "London"})	[:LIVES_IN]
(:Person {name: "Jim"})	(:Place {name: "Paris"})	[:LIVES_IN]

Composition de clauses

- Chaque clause suivante (C) a comme input le graphe **et** la table des résultats de la clause précédente
- Cette clause C est évaluée une fois pour chaque ligne de la table des résultats intermédiaires.

```
MATCH (p: Person)
WHERE p.name STARTS with “J”
MATCH (c: Place)
WHERE c.name = p.city
CREATE (p)-[r: LIVES_IN]->(c)
REMOVE p.city
RETURN p, c, r
```



p	c	r
(:Person {name: "John"})	(:Place {name: "London"})	[:LIVES_IN]
(:Person {name: "Jim"})	(:Place {name: "Paris"})	[:LIVES_IN]

Composition de clauses

- Autre exemple :

```
MATCH (p : Person {firstName: “John"})
SET p.name="Doe"
REMOVE p:German
RETURN p.name, labels(p)
```

La clause MERGE

MERGE (p : Person {name : 'Tom Hanks'})

RETURN p

Cela se comporte comme :

MATCH (p : Person {name : 'Tom Hanks'})

Mais, si aucun match n'est trouvé, cela se comporte comme

CREATE (p : Person {name : 'Tom Hanks'})

Attention le pattern entier doit avoir un match, un match partiel ne suffit pas, et déclenche la création de nouveaux noeuds

La clause MERGE

```
MERGE (p : Person {name : 'Tom Hanks'})
RETURN p
```

- Sur un graphe vide donc crée un noeud Person de nom Tom Hanks :

La clause MERGE

```
MERGE (p : Person {name : 'Tom Hanks'})
SET p.age = 67
RETURN p
```

Il y a un match, Merge se comporte comme MATCH

Ajoute une propriété au noeud existant

La clause MERGE

```
MERGE (p : Person {name : 'Tom Hanks', oscar: true})
RETURN p
```

La clause MERGE

```
MERGE (p : Person {name : 'Tom Hanks', oscar: true})
RETURN p
```

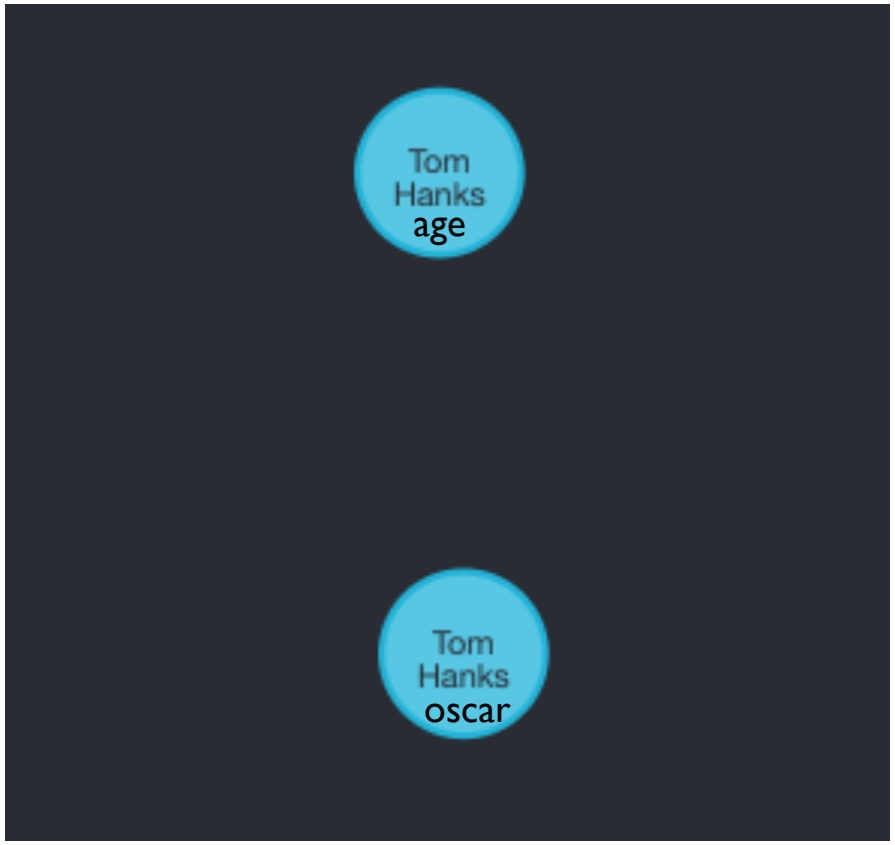
Il n'y a pas de noeud p: Person avec à la fois name: 'Tom Hanks' et oscar: true dans le graphe…

La clause MERGE

MERGE (p : Person {name : 'Tom Hanks', oscar: true})

RETURN p

Un nouveau noeud est créé…



La clause MERGE

```
MERGE (p : Person {name : 'Tom Hanks'}) - [r: ACTED_IN]
                ->(m : Movie {title: 'The Terminal'})
RETURN p, r, m
```

La clause MERGE

```
MERGE (p : Person {name : 'Tom Hanks'}) - [r: ACTED_IN]
                ->(m : Movie {title: 'The Terminal'})
RETURN p, r, m
```

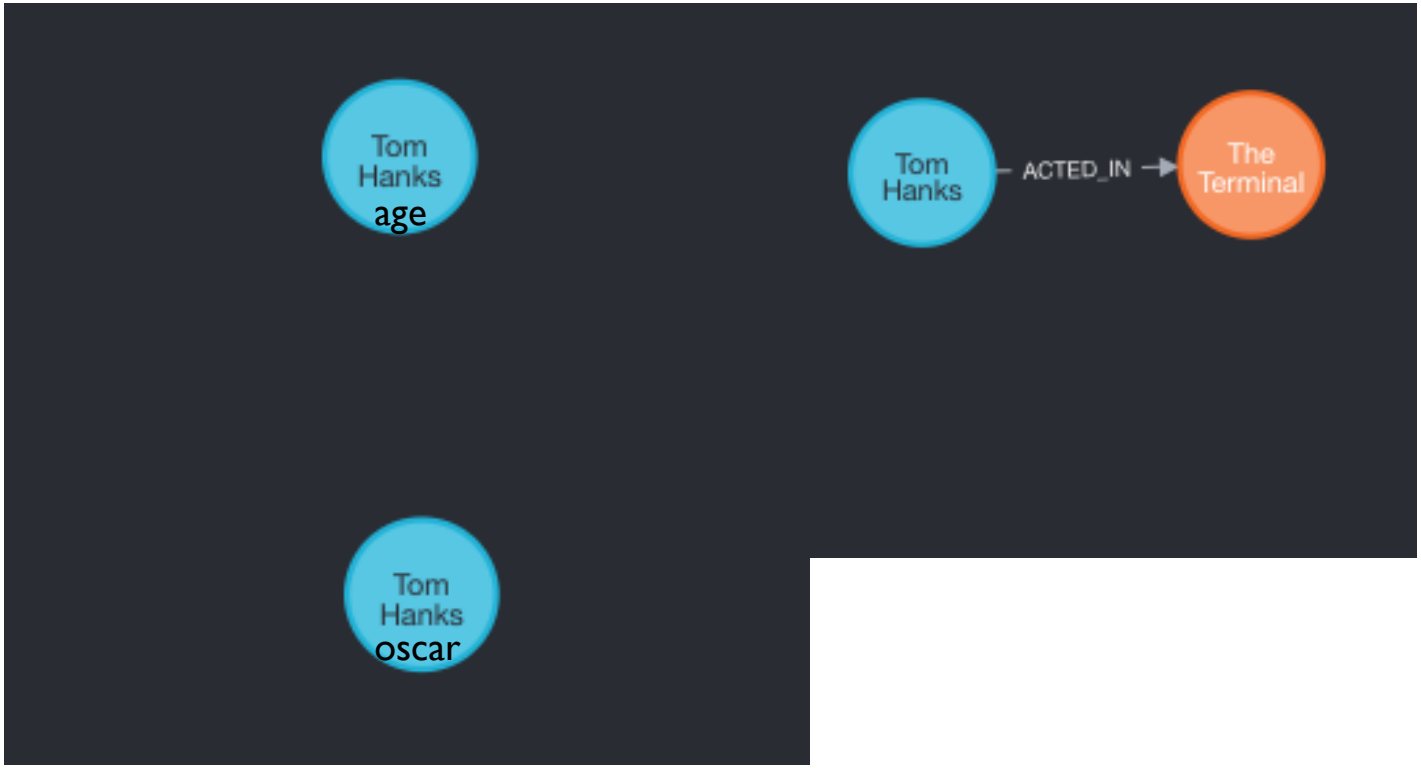
Il n'y a pas de noeud a: Movie avec pour titre : 'The Terminal' dans le graphe, mais il y a un noeud :Person avec name: 'Tom Hanks'

A votre avis, que va-t-il se passer ?

La clause MERGE

```
MERGE (p : Person {name : 'Tom Hanks'}) - [r: ACTED_IN]
        ->(m : Movie {title: 'The Terminal'})
RETURN p, r, m
```

Deux nouveau noeuds et une relation sont créés…



La clause MERGE

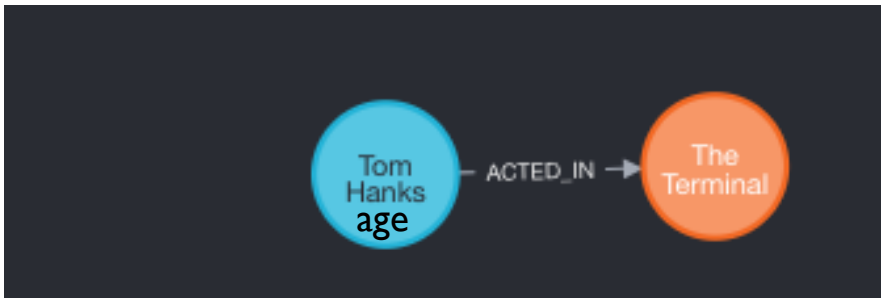
Pour fusionner le nouveau noeud avec le noeud existant :

MERGE (p : Person {name : 'Tom Hanks'})

MERGE (m : Movie {title: 'The Terminal'})

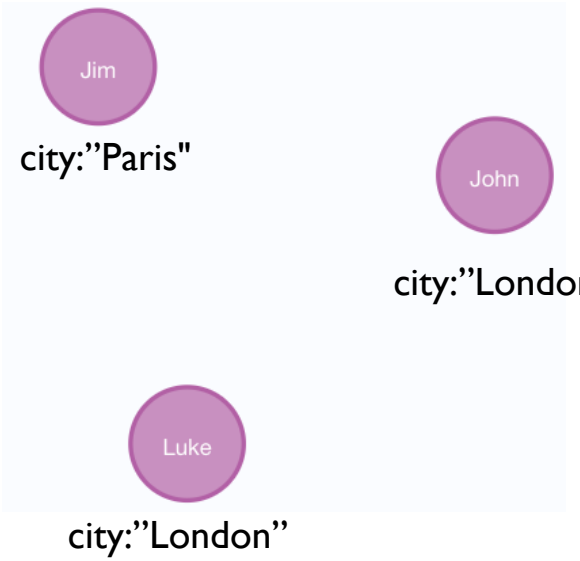
MERGE (p) - [r: ACTED_IN]-> (m)

RETURN p, r, m



La clause MERGE

- Utile pour créer des nouveaux noeuds seulement s'ils n'ont pas déjà été crées :

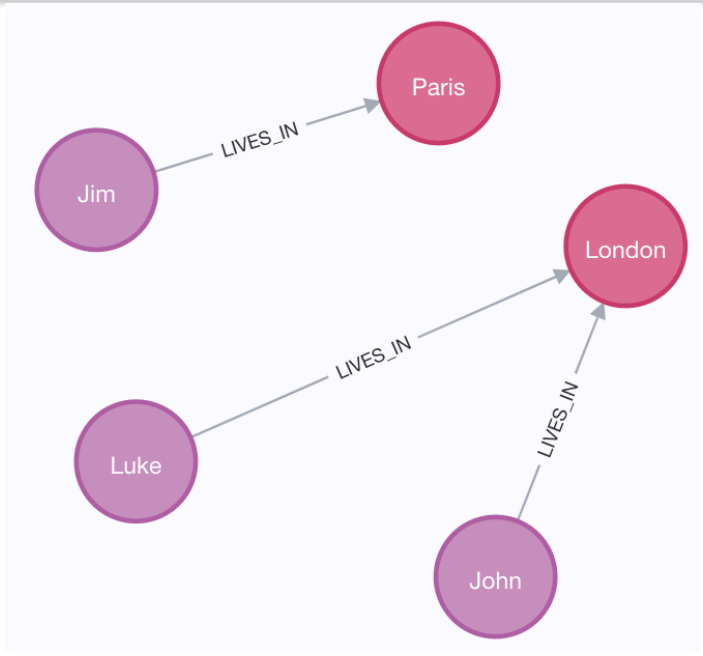


MATCH (p: Person)

MERGE (c : City {name: p.city})

MERGE (p) -[r: LIVES_IN]->(c)

REMOVE p.city



ON CREATE et ON MATCH

```
MERGE (p : Person {name : 'Your Name'})
 ON CREATE SET p.created = timestamp(), p.updated = 0
 ON MATCH SET p.updated = p.updated + 1
RETURN p.created, p.updated
```

Agrégation

Fonctions d'agrégation usuelles : avg(), count(), max(), min(), sum(), stDev(), collect()…

MATCH (p: Person)

p
(:Person:User {name: "Doe"})
(:Person {name: "Karl",age: 21})
(:Person {name: "John",age: 30})
(:Person {name: "Jim",age: 28})
(:Person {name: "Luke",age: 25})

RETURN avg (p.age)

avg(p.age)
26.0

Calcule la moyenne d'age dans la colonne p du résultat.

Agrégation

On peut retourner des expressions sans agrégation en plus des expressions avec agrégation. Dans ce cas les expressions sans agrégation définissent une clef de groupage

MATCH (p: Person)-[:FRIEND]->(f)

p	f
(:Person {name: "Karl",age: 21})	(:Person {name: "John",age: 30})
(:Person {name: "Karl",age: 21})	(:Person {name: "Jim",age: 25})
(:Person {name: "John",age: 30})	(:Person {name: "Jim",age: 28})
(:Person {name: "Jim",age: 28})	(:Person {name: "Karl",age: 21})
(:Person {name: "Jim",age: 28})	(:Person {name: "Jim",age: 25})
(:Person {name: "Jim",age: 28})	(:Person {name: "John",age: 30})
(:Person {name: "Jim",age: 25})	(:Person {name: "Jim",age: 28})

RETURN p.name, avg (f.age)

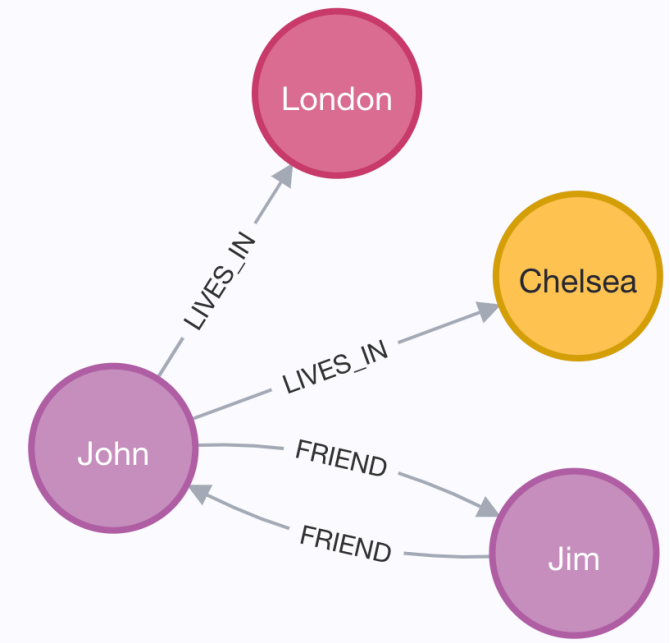
Similaire à GROUP BY p.name en SQL

p.name	avg (f.age)
"Karl"	27.5
"John"	28.0
"Jim"	26.0

Agrégation

```
MATCH (n {name: "John"})-[r]->()
RETURN type(r), count(*)
```

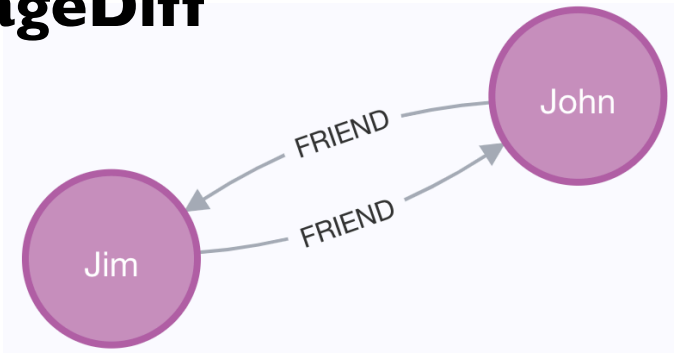
type(r)	count(*)
"LIVES_IN"	2
"FRIEND"	1



La clause WITH

WITH sert à manipuler le résultat temporaire passé d'une clause à l'autre, dans l'enchainement de plusieurs clauses

```
MATCH (n)-[r:FRIEND]->(friend)
WITH n, r, n.age - friend.age AS ageDiff
SET r.ageDiff = ageDiff
RETURN n, ageDiff
```



https://neo4j.com/docs/cypher-manual/current/clauses/with/

La clause WITH

WITH sert à manipuler le résultat temporaire passé d'une clause à l'autre, dans l'enchainement de plusieurs clauses

MATCH (n)-[r:FRIEND]->(friend)

n	r	friend
(:Person {name: "Jim",age: 28})	[:FRIEND]	(:Person {name: "John",age: 30})
(:Person {name: "John",age: 30})	[:FRIEND]	(:Person {name: "Jim",age: 28})

WITH n, r, n.age - friend.age AS ageDiff

n	r	ageDiff
(:Person {name: "Jim",age: 28})	[:FRIEND]	-2
(:Person {name: "John",age: 30})	[:FRIEND]	2

SET r.ageDiff = ageDiff

n	r	ageDiff
(:Person {name: "Jim",age: 28})	[:FRIEND {ageDiff: -2}]	-2
(:Person {name: "John",age: 30})	[:FRIEND {ageDiff: 2}]	2

RETURN n.name, ageDiff

n.name	ageDiff
"Jim"	-2
"John"	2

La clause WITH

- Attention : si WITH est present dans l'enchainement, seulement les variables spécifiées dans WITH sont passées à la suite se la requête :

MATCH (n)-[r:FRIEND]->(friend)

n	r	friend
(:Person {name: "Jim",age: 28})	[:FRIEND]	(:Person {name: "John",age: 30})
(:Person {name: "John",age: 30})	[:FRIEND]	(:Person {name: "Jim",age: 28})

WITH r, n.age - friend.age AS ageDiff

r	ageDiff
[:FRIEND]	-2
[:FRIEND]	2

SET r.ageDiff = ageDiff

r	ageDiff
[:FRIEND {ageDiff: -2}]	-2
[:FRIEND {ageDiff: 2}]	2

RETURN n.name, ageDiff

ERROR

La clause WITH

```
MATCH (n {name : 'John'})-[:FRIEND]-(friend)
WITH n, count(friend) AS friendsCount
SET n.friendsCount= friendsCount
RETURN n.friendsCount
```

Mise à jour du graphe par ajout des données agrégées

WITH permet ici de passer de la lecture à l'écriture

https://neo4j.com/docs/cypher-manual/current/clauses/with/

La clause WITH

- Sous-clauses de WITH :
  - **ORDER BY, LIMIT** (comme de RETURN)
  - **WHERE** (comme pour MATCH)

La clause WITH

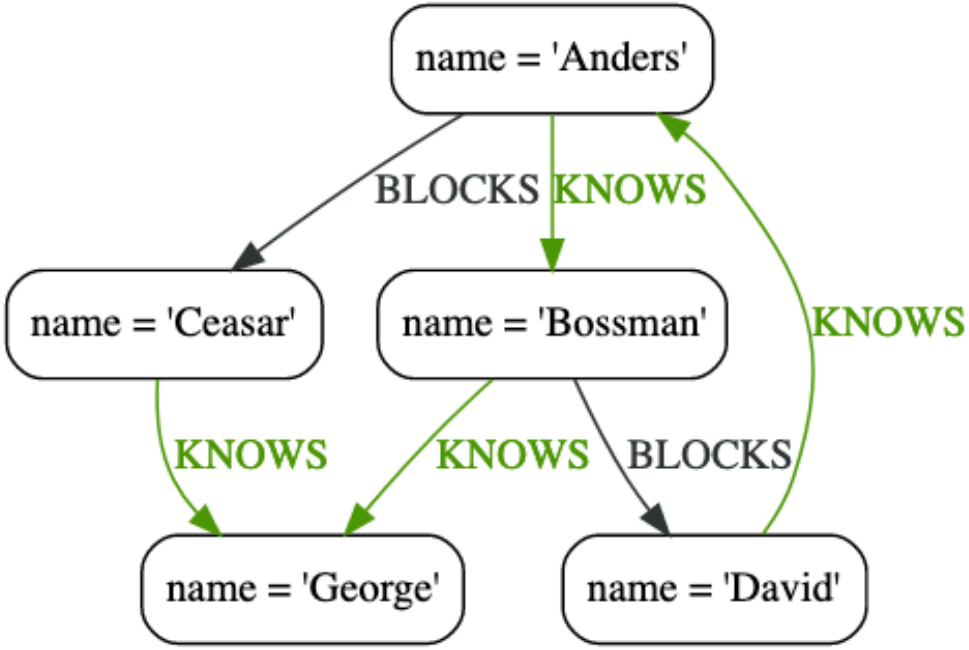
```
MATCH (n {name:”Jim"})
WITH n
ORDER BY n.age
LIMIT 1
DETACH DELETE n
```

La clause WITH

Résultats agrégés passent par un WITH pour être filtrés par le WHERE :

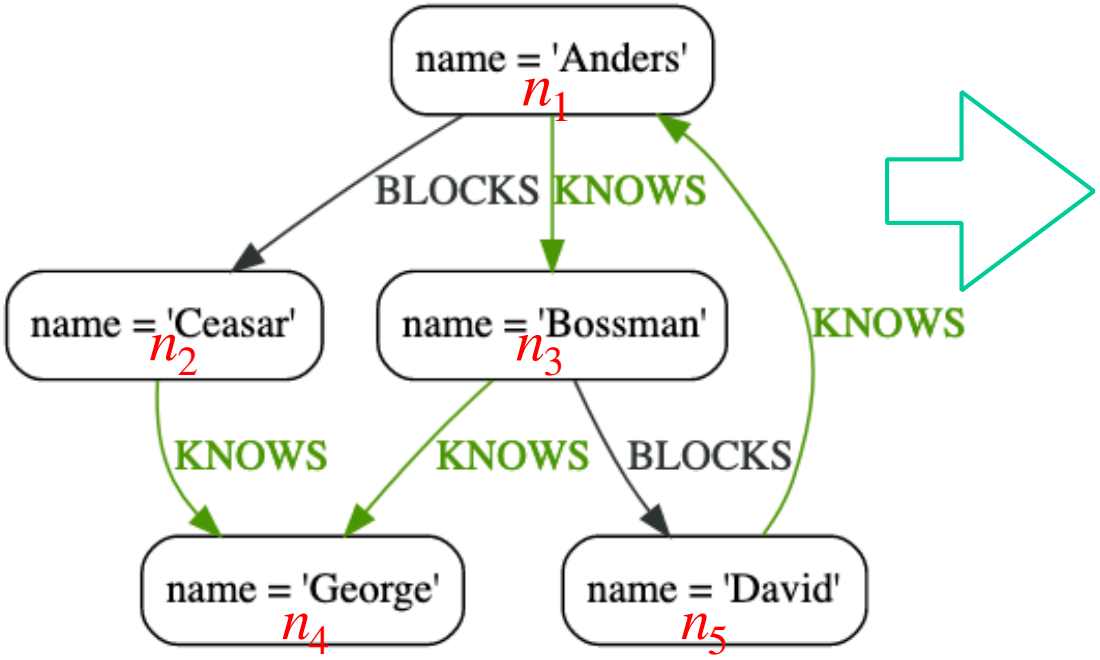
```
MATCH (n {name : 'David'}) - - (other) - -> ()
WITH other, count(*) AS fof
WHERE fof > 1
RETURN other.name
```

other.name
'Anders'



La clause WITH

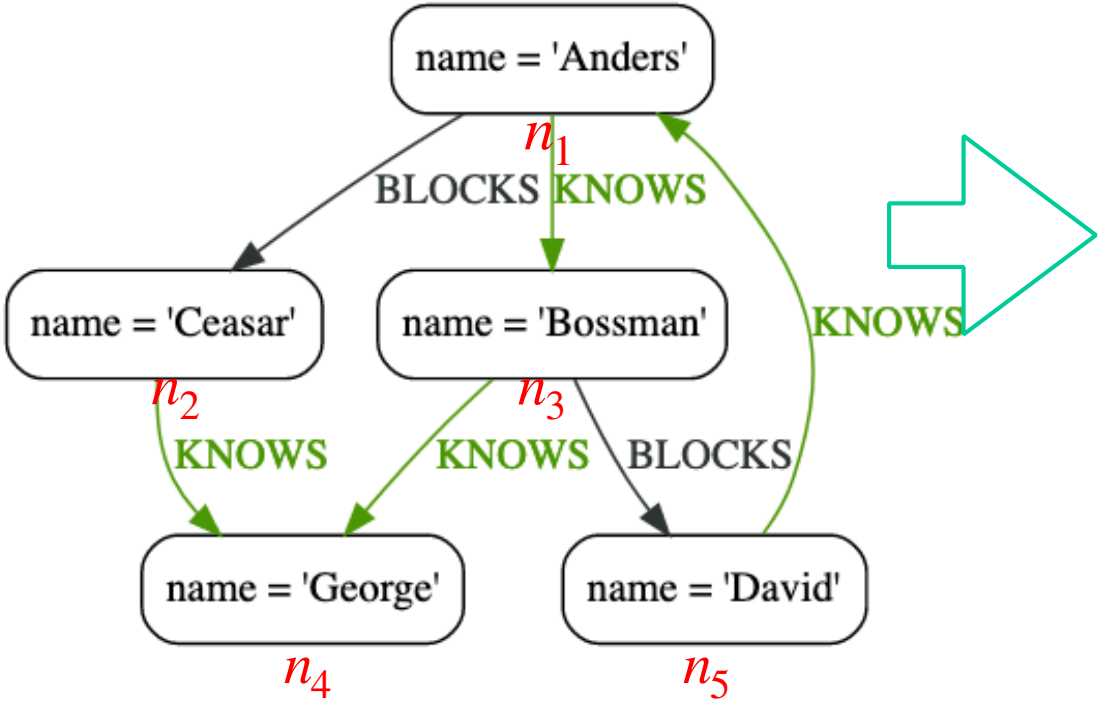
```
MATCH (n {name : 'David'}) - - (other) - -> ()
WITH other, count(*) AS fof
WHERE fof > 1
RETURN other.name
```



n	other	fof
n_5	n_3	1
n_5	n_1	2

La clause WITH

```
MATCH (n {name : 'David'}) - - (other) - -> ()
WITH other, count(*) AS fof
WHERE fof > 1
RETURN other.name
```



n	other	fof
n_5	n_3	1
n_5	n_1	2

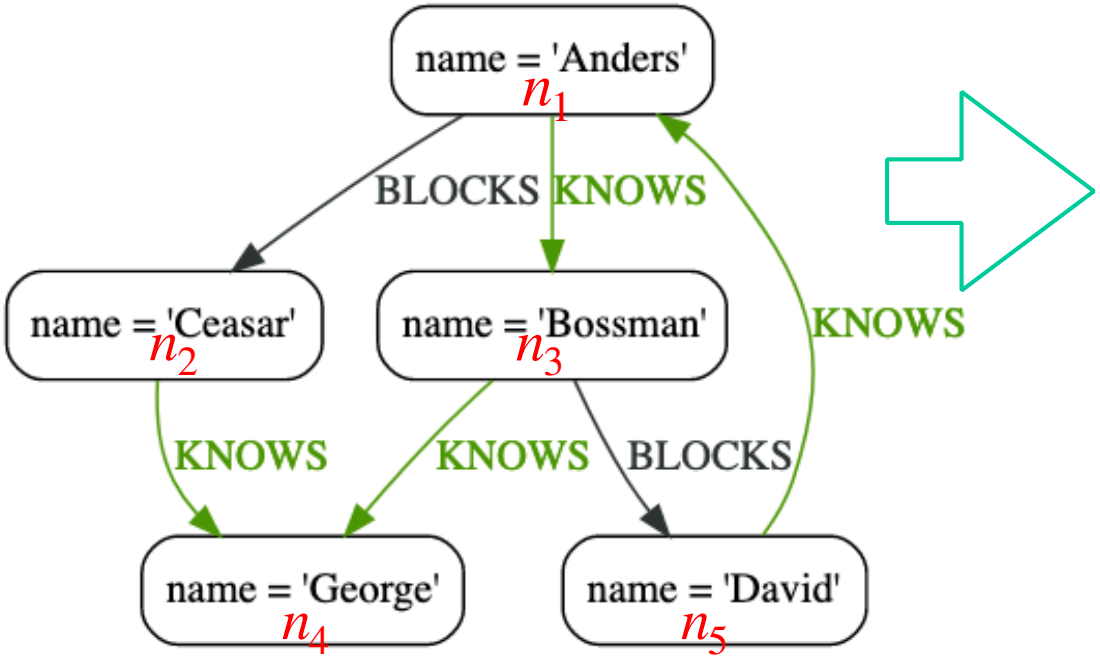
Premier match : seconde arrête sortante uniquement vers n_4 (fof = 1)

ATTENTION : semantique TRAIL

Second match : seconde arrête sortante vers n_2 et n_3 (fof = 2)

La clause WITH

```
MATCH (n {name : 'David'}) - - (other) - -> ()
WITH other, count(*) AS fof
WHERE fof > 1
RETURN other.name
```



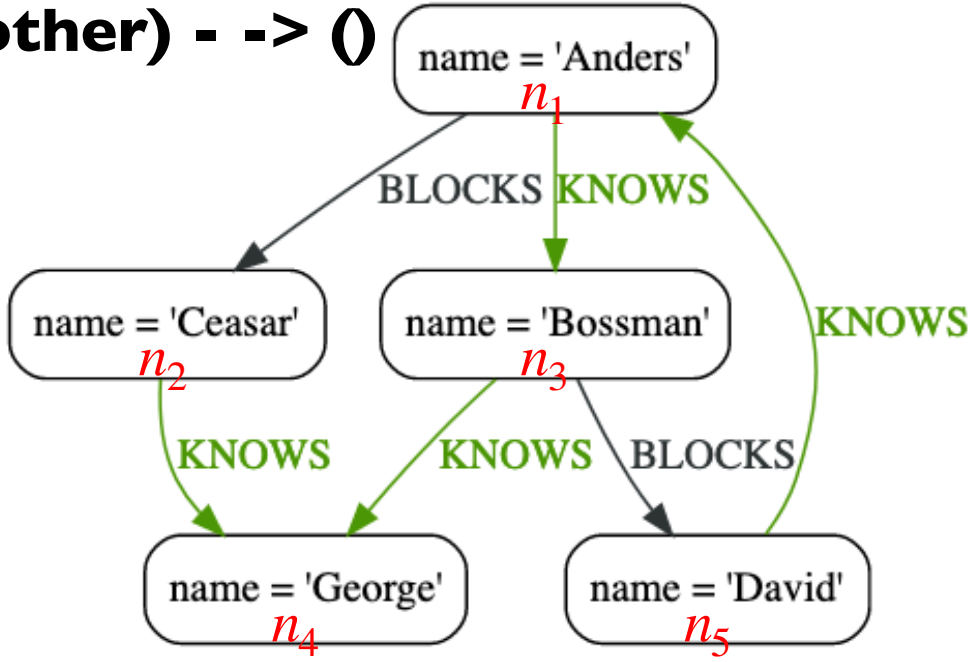
n	other	fof
n_5	n_3	1
n_5	n_1	2

WITH other (Cypher) ≃ GROUP BY other (SQL)

La clause WITH

Les résultats agrégés sont finalement filtrés par le WHERE :

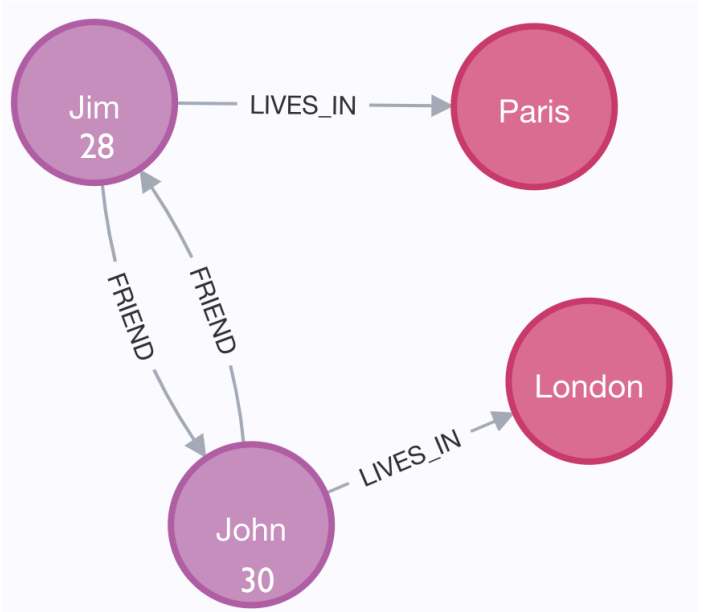
```
MATCH (n {name : 'David'}) - - (other) - -> ()
WITH other, count(*) AS fof
WHERE fof > 1
RETURN other.name
```



n	other	fof	other.name
n_5	n_1	2	*'Anders'*

La clause OPTIONAL MATCH

```
MATCH (a:Person)
WHERE a.age > 27
OPTIONAL MATCH (a)-[r:LIVES_IN]->(c)
WHERE c.name<>'Paris'
RETURN a, c
```

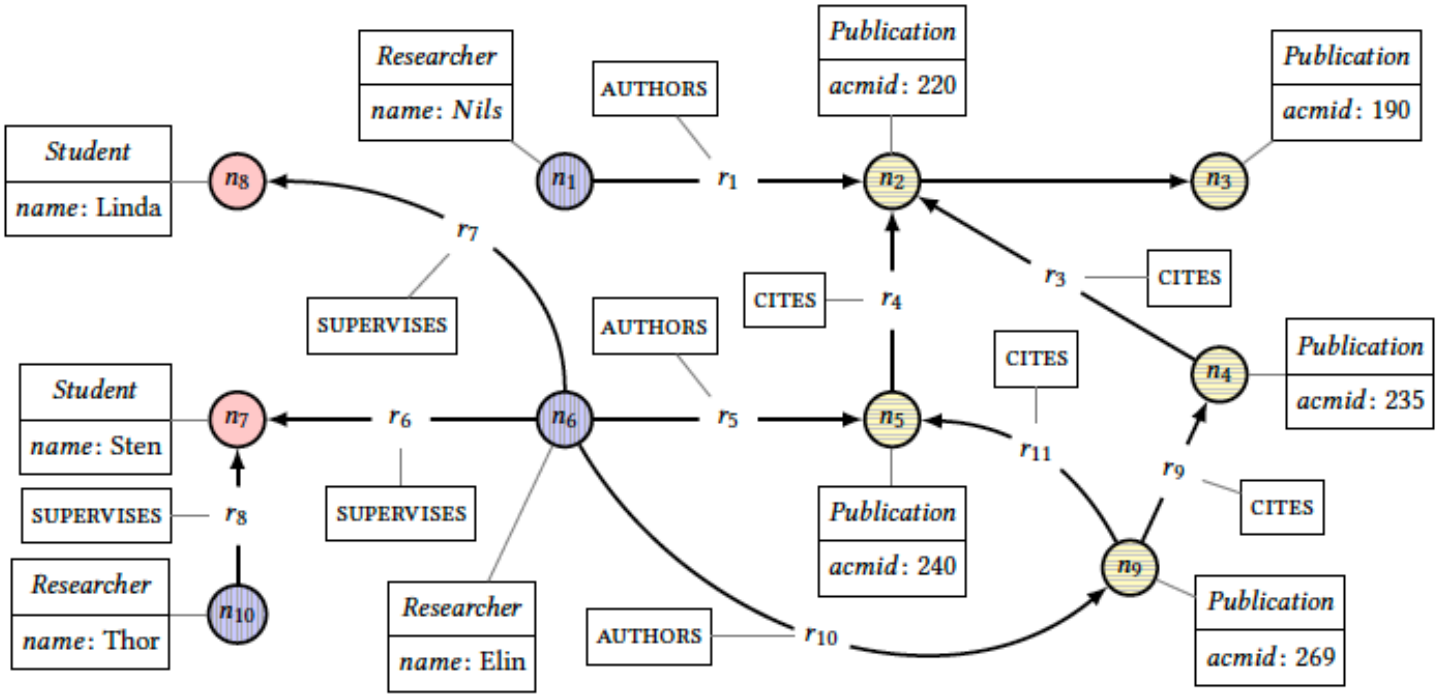


- OPTIONAL MATCH est suivi d'un pattern
  - Possiblement incluant une clause WHERE
- Si ce pattern n'a pas de match, OPTIONAL MATCH retourne un résultat avec valeur <null> pour les variables qui n'ont pas de binding

a	c
(:Person {name: "John",age: 30})	(:City {name: "London"})
(:Person {name: "Jim",age: 28})	null

- OPTIONAL MATCH ≃ OUTER JOIN en SQL

La clause OPTIONAL MATCH



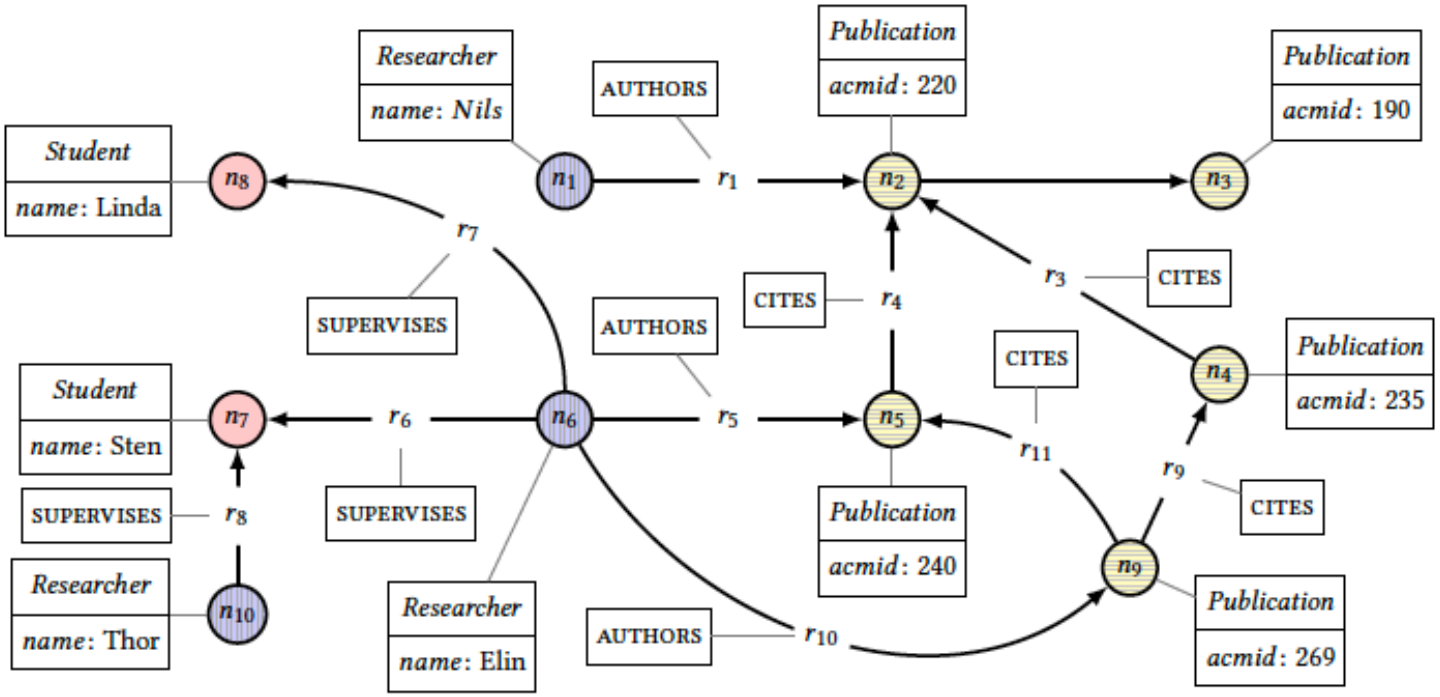

```
MATCH (r:Researcher)
OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
WITH r, count(s) AS studentsSupervised
MATCH (r)-[:AUTHORS]->(p1:Publication)
OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
RETURN r.name, studentsSupervised,
       count(DISTINCT p2) AS citedCount
```

r.name	studentsSupervised	citedCount
Nils	0	3
Elin	2	1

résultat de l'évaluation de la requête

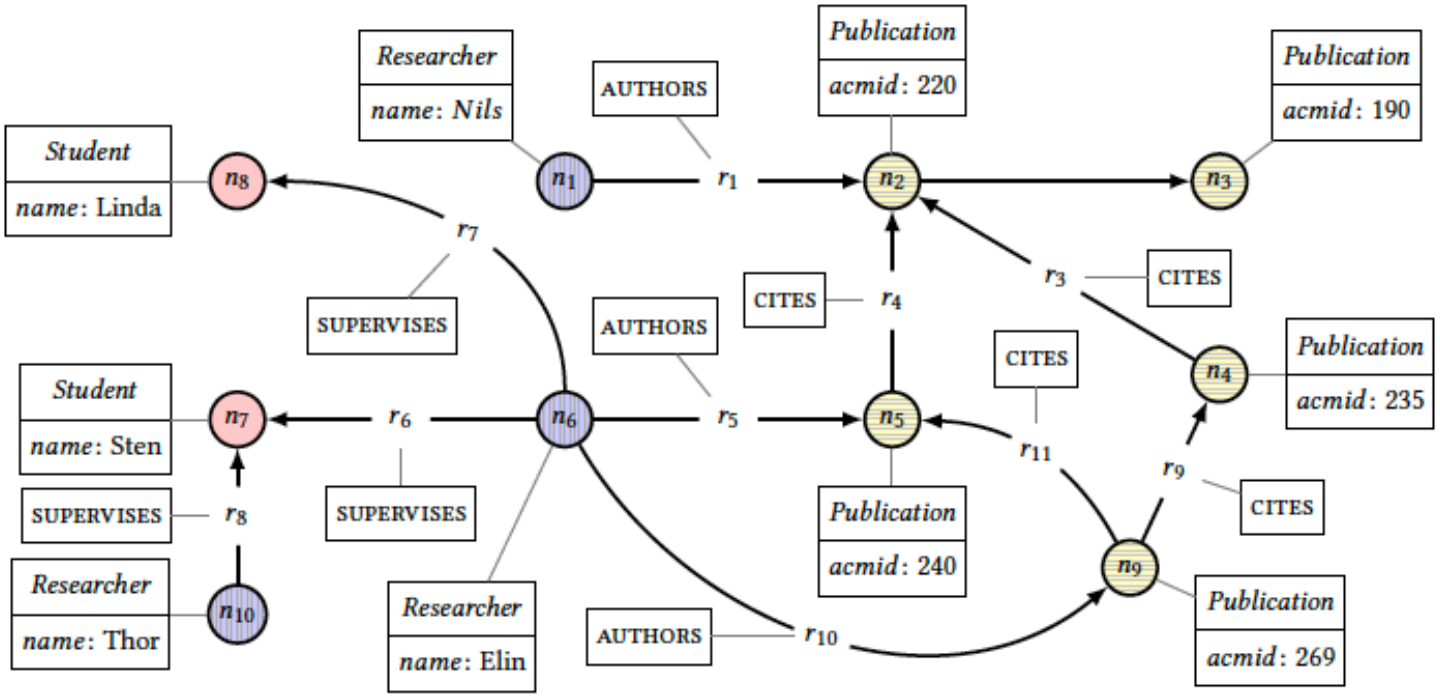
Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, Andrés Taylor: Cypher: An Evolving Query Language for Property Graphs. SIGMOD Conference 2018:

La clause OPTIONAL MATCH




```
1 MATCH (r:Researcher)
2 OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
3 WITH r, count(s) AS studentsSupervised
4 MATCH (r)-[:AUTHORS]->(p1:Publication)
5 OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
6 RETURN r.name, studentsSupervised,
7        count(DISTINCT p2) AS citedCount
```

La clause OPTIONAL MATCH




```
1 MATCH (r:Researcher)
2 OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
3 WITH r, count(s) AS studentsSupervised
4 MATCH (r)-[:AUTHORS]->(p1:Publication)
5 OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
6 RETURN r.name, studentsSupervised,
7        count(DISTINCT p2) AS citedCount
```

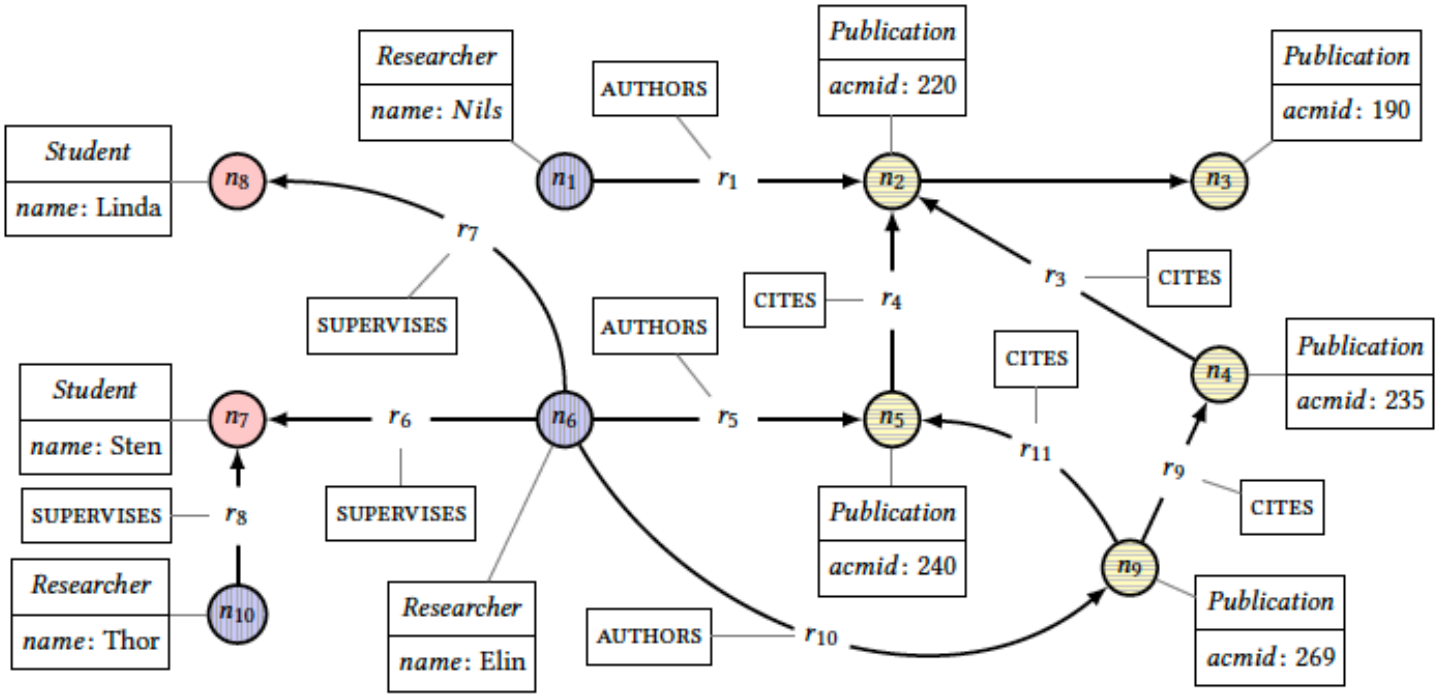
variable binding

lignes 1 & 2

r	s
n_1	null
n_6	n_7
n_6	n_8
n_{10}	n_7

Cypher optional match ≃ SQL outer join

La clause OPTIONAL MATCH



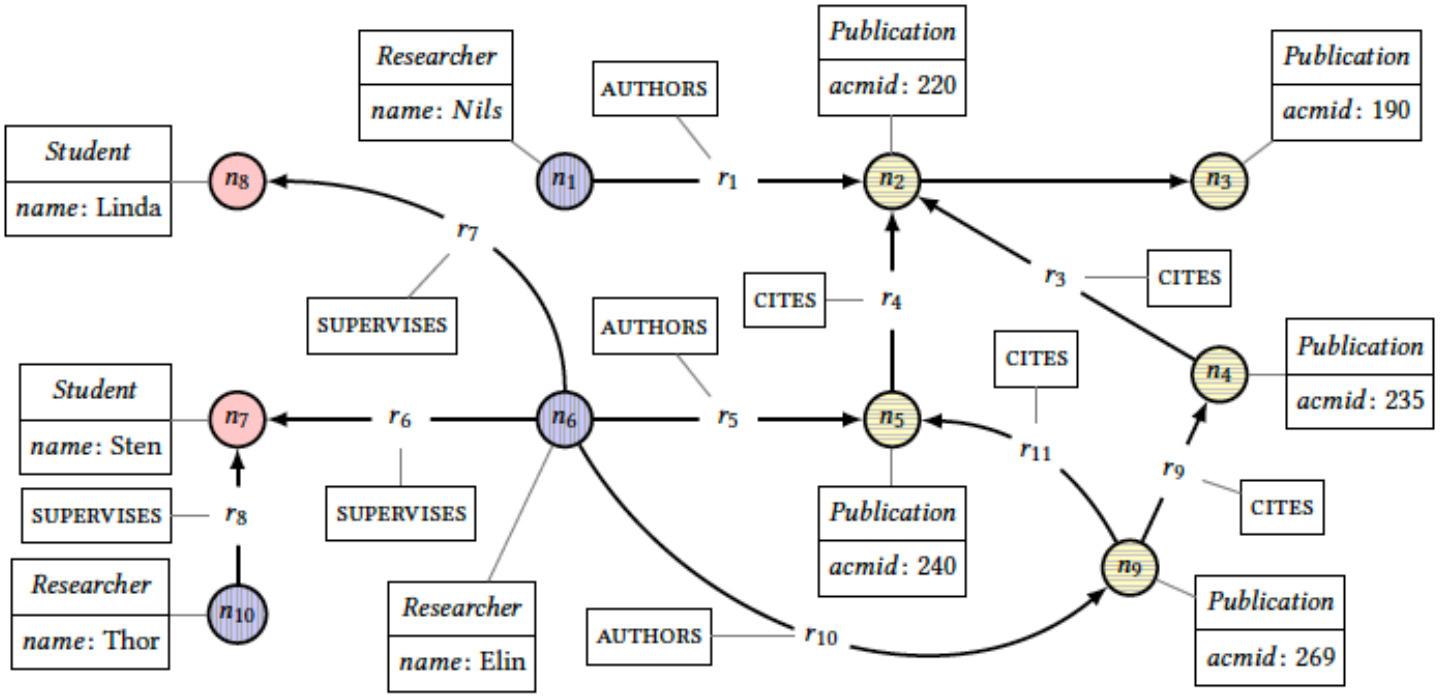

```
1 MATCH (r:Researcher)
2 OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
3 WITH r, count(s) AS studentsSupervised
4 MATCH (r)-[:AUTHORS]->(p1:Publication)
5 OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
6 RETURN r.name, studentsSupervised,
7        count(DISTINCT p2) AS citedCount
```

variable binding

ligne 3

r	studentsSupervised
n_1	0
n_6	2
n_{10}	1

La clause OPTIONAL MATCH




```
1 MATCH (r:Researcher)
2 OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
3 WITH r, count(s) AS studentsSupervised
4 MATCH (r)-[:AUTHORS]->(p1:Publication)
5 OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
6 RETURN r.name, studentsSupervised,
7        count(DISTINCT p2) AS citedCount
```

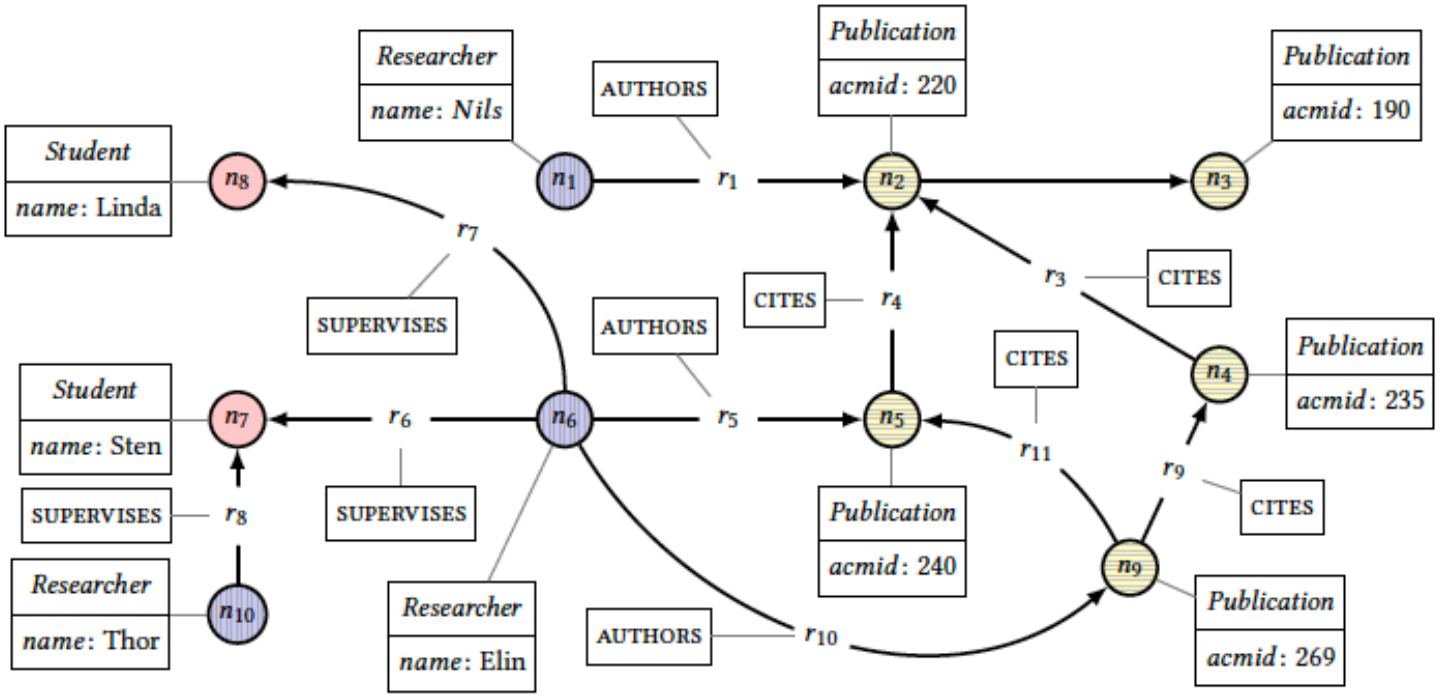
r	studentsSupervised	p1
n_1	0	n_2
n_6	2	n_5
n_6	2	n_9

variable binding

ligne 4

n_{10} - qui n'a rien publié - ne figure plus parmi les valeurs possibles de r (≠ optional match)

La clause OPTIONAL MATCH




```
1 MATCH (r:Researcher)
2 OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
3 WITH r, count(s) AS studentsSupervised
4 MATCH (r)-[:AUTHORS]->(p1:Publication)
5 OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
6 RETURN r.name, studentsSupervised,
7        count(DISTINCT p2) AS citedCount
```

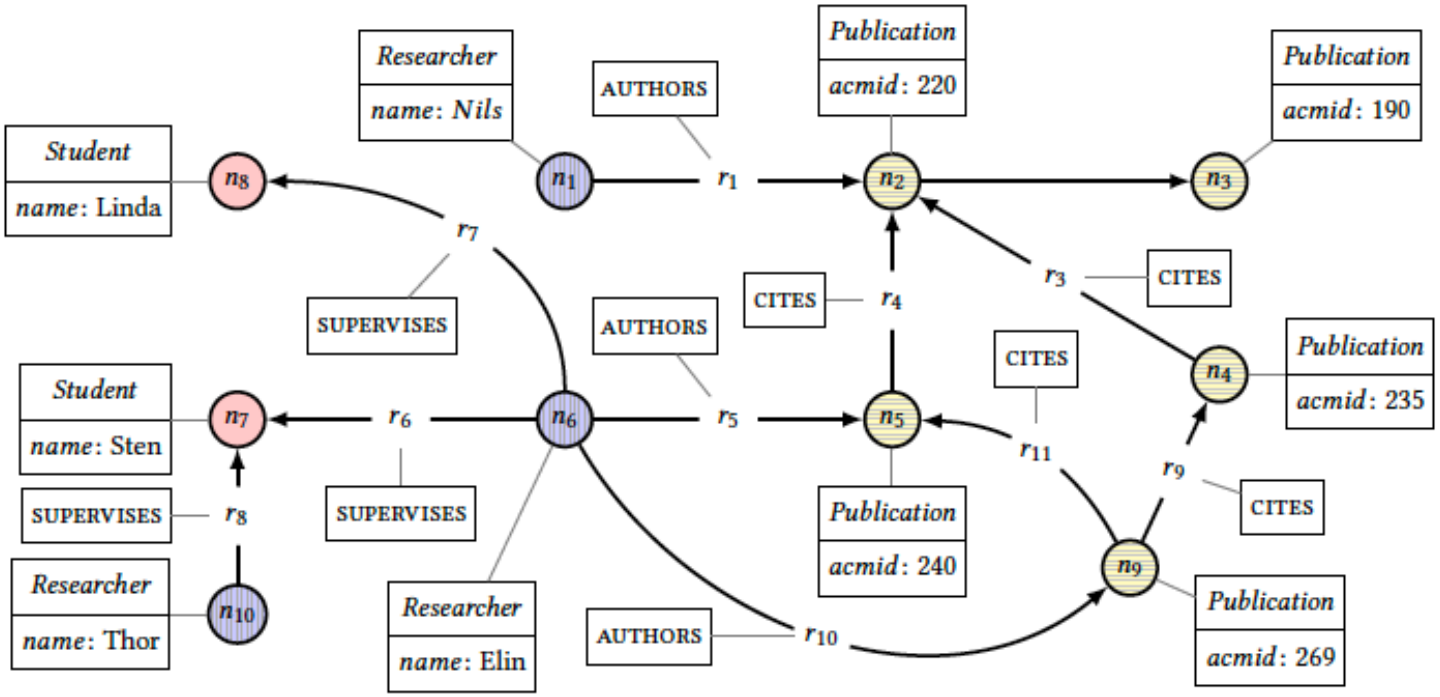
variable binding

ligne 5

r	studentsSupervised	p1	p2	
n_1	0	n_2	n_4	
n_1	0	n_2	n_9	†
n_1	0	n_2	n_5	
n_1	0	n_2	n_9	†
n_6	2	n_5	n_9	
n_6	2	n_9	null	

† = doublons

La clause OPTIONAL MATCH




```
1 MATCH (r:Researcher)
2 OPTIONAL MATCH (r)-[:SUPERVISES]->(s:Student)
3 WITH r, count(s) AS studentsSupervised
4 MATCH (r)-[:AUTHORS]->(p1:Publication)
5 OPTIONAL MATCH (p1)<-[:CITES*]-(p2:Publication)
6 RETURN r.name, studentsSupervised,
7        count(DISTINCT p2) AS citedCount
```

r.name	studentsSupervised	citedCount
Nils	0	3
Elin	2	1

projection du RETURN

lignes 6 et 7

Remarque : `count()` ignore les valeurs `null`

Identifiants de noeuds

En Cypher chaque noeud et chaque relation a un identifiant unique

```
MATCH (n : City)
RETURN elementId(n)
```

elementId(n)
"4:11423029-717d-4eb6-82bb-5e6a6f8e7be1:35"
"4:11423029-717d-4eb6-82bb-5e6a6f8e7be1:36"

On peut y accéder directement avec la fonction elementId(), mais le sgbd réutilise ses identifiants internes quand les noeuds et relations sont supprimés, donc risqué..

On verra bientôt comment attribuer des contraintes d'unicité

Listes

- En Cypher les listes sont des valeurs,
  - Elles peuvent être crée, modifiées, affectées, retournées
- Creation de liste : [ ]

[val1, val2, …, valn]

est une liste comportant la séquence des valeurs val1, val2, …valn

- Creation de liste : range ()

range(n,n+m)

est une liste comportant la séquence des valeurs n, n+1, n+2 …n+m

https://neo4j.com/docs/cypher-manual/current/values-and-types/lists/#cypher-lists-general

Listes

Exemples :

- **RETURN** [2, 3, 4, 5] AS numberlist

numberlist
[2, 3, 4, 5]

- **WITH** range(5,10) AS list
  RETURN list[0]

list[0]
5

Listes de listes possibles aussi [[1,2],3]

Opérateurs de liste

- Concatenation :

```
RETURN [1,2,3]+[4,5,6] as myList
```

myList
[1, 2, 3, 4, 5, 6]

- Acces

```
RETURN range(0,10)[3]
```

range(0,10)[3]
3

- Appartenance

```
MATCH (p:Person)
WHERE p.age IN [26,27,28]
RETURN p
```

p
(:Person {name: "Jim",age: 28})

Opérateurs de liste

- UNWIND : ajoute une nouvelle colonne au résultat temporaire, contenant tous les elements de la liste

```
WITH [1,2,3] as nblist
```

nblist
[1, 2, 3]

```
UNWIND nblist as nb
```

nblist	nb
[1, 2, 3]	1
[1, 2, 3]	2
[1, 2, 3]	3

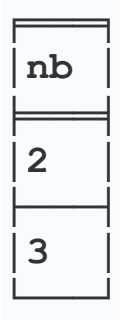
```
RETURN nb
```

nb
1
2
3

Opérateurs de liste

- UNWIND : utile suivi d'un WITH pour parcourir les elements de la liste

```
WITH [1,2,3] as nblist
UNWIND nblist as nb
WITH nb
WHERE nb >=2
RETURN nb
```

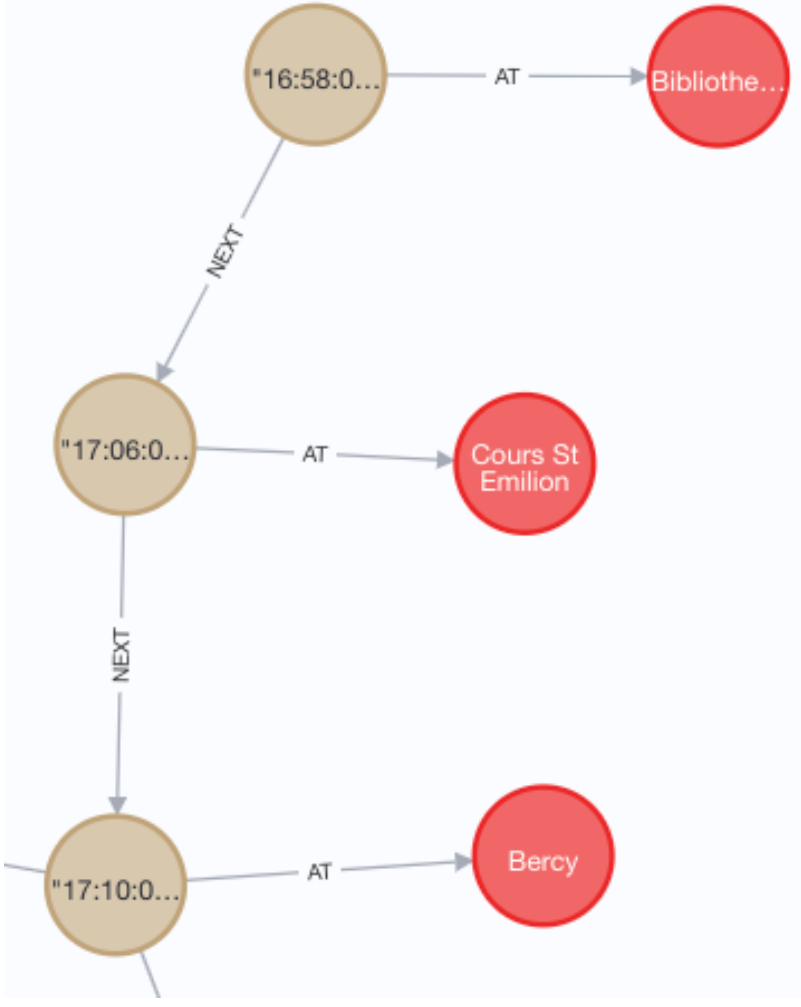


Opérateurs de liste

- On peut récupérer les noeuds d'un chemin dans une liste : `nodes(path_var)`

```
MATCH (begin)-[:AT]-> ({name: 'Bibliotheque'}),
p=(begin)-[:NEXT*]->(end),
(end)-[:AT]-> ({name: 'Bercy'})
UNWIND nodes(p) AS n
RETURN n.arrives
```

n.arrives
"16:58:00Z"
"17:06:00Z"
"17:10:00Z"



La clause FOREACH

Utilisée pour mettre à jour les données d'une liste

```
MATCH p=(begin)-[*]->(end)
WHERE begin.name='départ' AND
end.name='arrivée'
FOREACH (n IN nodes(p) | SET n.marked = TRUE)
```

Compréhension sur les listes

Création de listes à partir d'autres listes par compréhension ensembliste

RETURN [x IN range(0,10) WHERE x%2 = 0 | x^2] AS result

result
[0.0, 4.0, 16.0, 36.0, 64.0, 100.0]

Compréhension sur les patterns

Création de listes à partir des matchs d'un path pattern :

```
MATCH (a: Person)

RETURN a.name, [(a)-[:LIVES_IN]->(c) WHERE c: City | c.name]
AS villes
```

Le path pattern **(a)-[:LIVES_IN]->(c) WHERE c: City** est evalué comme dans une clause MATCH

Pour chaque match, c.name est ajouté à liste

a.name.	villes.
"John"	["London","Chelsea"]
"Jim"	["Paris"]

Fonction collect() pour les listes

Fonction d'agrégation qui construit une liste avec les valeurs dans une colonne :

```
MATCH (a: Person {name :  'Keanu Reeves'})-- >(b)
RETURN collect(b.released)
```

Fonction collect() pour les listes

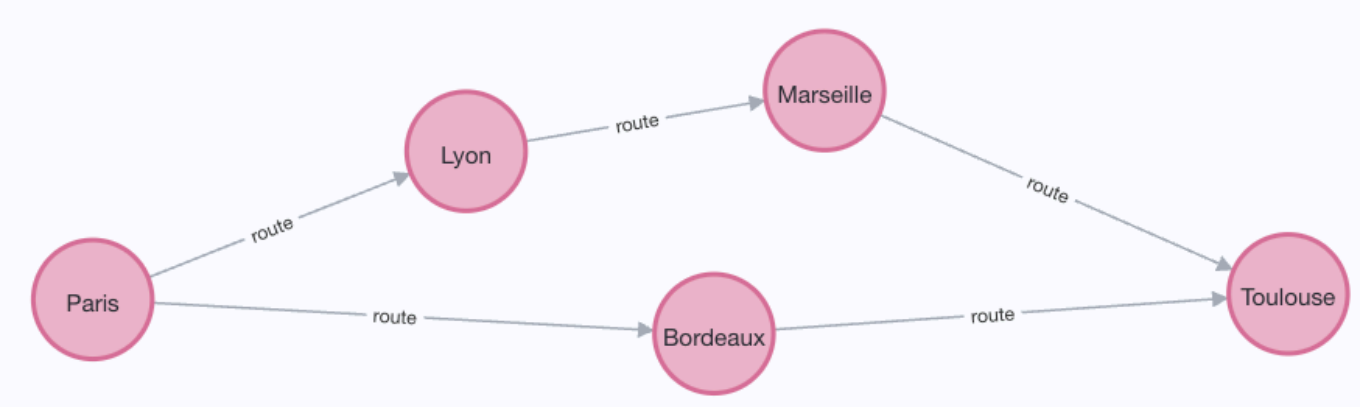
La même requête sans doublons :

```
MATCH (a: Person {name :  'Keanu Reeves'})-- >(b)
RETURN collect(distinct b.released)
```

Listes et variables de groupe

- Le variables utilisées dans un path pattern quantifié sont appelées variables de groupe
- Pour chaque match du pattern quantifié elles retournent la liste des noeuds qu'elles matchent

```
MATCH (:City{name:"Paris"}) ((start)-[:route]->(end))+
      (:City{name:"Toulouse"})
RETURN start, end
```



start	end
[(:City {name: "Paris"}), (:City {name: "Bordeaux"})]	[(:City {name: "Bordeaux"}), (:City {name: "Toulouse"})]
[(:City {name: "Paris"}), (:City {name: "Lyon"}), (:City {name: "Marseille"})]	[(:City {name: "Lyon"}), (:City {name: "Marseille"}), (:City {name: "Toulouse"})]

Clause de sous requête avec CALL

- CALL { … }, comme toutes les clauses est exécutée une fois pour chaque ligne t du résultat intermédiaire
- Execution de CALL{ …} sut une ligne t :
  - évalue la sous requête sur t
  - Modifie le résultat intermédiaire : associe à t le résultat de la sous-requête

```
UNWIND [0, 1, 2] as x
CALL {
  MATCH (n : Film)
  RETURN count(n) AS nombre
}
RETURN nombre
```

https://neo4j.com/docs/cypher-manual/current/clauses/call-subquery/

Clause de sous requête avec CALL

```
UNWIND [0, 1, 2] as x
```

x
0
1
2

```
CALL {
  MATCH (n : Film)
  RETURN count(n) AS nombre
}
```

x	nombre
0	345
1	345
2	345

```
RETURN nombre
```

nombre
345
345
345

(Ici pour générer des doublons)

Clause de sous requête avec CALL

- Nécessaire de commencer par WITH si on veut utiliser les variables de la requête principale

```
UNWIND [0, 1, 2] as note
CALL {
  WITH note
  MATCH (n : Film)
  WHERE n.note =  note
  RETURN count(n) AS nombre
}
RETURN note, nombre
```

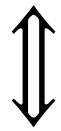
note	nombre
0	25
1	300
2	20

Attention : pas d'alias ou d'expression dans le WITH (juste WITH x, y), la sous requête doit finir par RETURN

Clause de sous requête avec CALL

- Parfois le même résultat pet être obtenu sans CALL

```
UNWIND [0, 1, 2] as x
CALL {
  WITH x
  RETURN x * 10 AS y
}
RETURN x, y
```



x.	y.
0	0
1	10
2	20

```
UNWIND [0, 1, 2] as x
RETURN x, x * 10 AS y
```

Clause de sous requête avec CALL

- D'autre fois le même résultat pet être obtenu sans CALL, mais pas de façon immediate

```
UNWIND [0, 1, 2, 3] as note
CALL {
  WITH note
  MATCH (n : Film)
  WHERE n.note =  note
  RETURN count(n) AS nombre
}
RETURN note, nombre
```

note	nombre
0	25
1	300
2	20
3	0

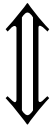
```
UNWIND [0, 1, 2, 3] as note
MATCH (n : Film)
WHERE n.note =  note
RETURN note, count(n) AS nombre
```

note	nombre
0	25
1	300
2	20

Clause de sous requête avec CALL

- D'autre fois le même résultat pet être obtenu sans CALL, mais pas de façon immediate

```
UNWIND [0, 1, 2, 3] as note
CALL {
  WITH note
  MATCH (n : Film)
  WHERE n.note =  note
  RETURN count(n) AS nombre
}
RETURN note, nombre
```



```
UNWIND [0, 1, 2, 3] as note
OPTIONAL MATCH (n : Film)
WHERE n.note =  note
RETURN note, count(n) AS nombre
```

note	nombre
0	25
1	300
2	20
3	0

Sous requête avec CALL post UNION

Avec les opérateurs ensemblistes, remplacer CALL est possible mais parfois l'écriture avec CALL est plus compacte

Seuls opérateurs ensemblistes Cypher : UNION et UNION ALL

```
CALL {
  MATCH (p: Person) RETURN p ORDER BY p.born
  ASC LIMIT 1
    UNION
  MATCH (p: Person) RETURN p ORDER BY p.born
  DESC LIMIT 1
}
RETURN p.name, p.born ORDER BY p.name
```

Sous requête avec CALL post UNION

Avec les opérateurs ensemblistes, remplacer CALL est possible mais parfois l'écriture avec CALL est plus compacte

Pas toujours :

```
MATCH (p: Person)
CALL {
  WITH p OPTIONAL MATCH (p)-[:DIRECTED]->(movie)
  RETURN movie
    UNION
  WITH p OPTIONAL MATCH (p)-[:ACTED_IN]->(movie)
  RETURN movie
}
RETURN DISTINCT p.name, count(movie)
```

Sous requête avec CALL post UNION

Avec les opérateurs ensemblistes, remplacer CALL est possible mais parfois l'écriture avec CALL est plus compacte

Pas toujours :

la requête précédente n'est pas équivalente à :

```
MATCH (p: Person)-[:DIRECTED|ACTED_IN]->(movie)
RETURN DISTINCT p.name, count(movie)
```

Ici les personnes n'ayant ni dirigé un film, ni joué dans un film ne sont pas comptées (la requête précédente incluait par exemple les critiques de films).

Sous requête avec CALL post UNION

Avec les opérateurs ensemblistes, remplacer CALL est possible mais parfois l'écriture avec CALL est plus compacte

Pas toujours :

Ici utiliser OPTIONAL MATCH permet d'éviter CALL **et** rend la requête plus compacte :

```
MATCH (p: Person)
OPTIONAL MATCH (p)-[:DIRECTED|ACTED_IN]->(movie)
RETURN DISTINCT p.name, count(movie)
```

Sous requête avec CALL post UNION

Attention à :

MATCH (p: Person)- ->(movie)

RETURN DISTINCT p.name, count(movie)

Le résultat dépend de la structure des données (problème si des arrêtes d'un nouveau type sont ensuite ajoutées), attention donc à la sémantique de la requête que vous avez en tête

Sous requête avec CALL et agrégation

CALL peut simplifier l'écriture de requêtes aussi en présence d'agrégation (évite le regroupement)

```
MATCH (p: Person) WHERE p.born IS NOT NULL
CALL {
  WITH p
  MATCH (other :Person) WHERE other.born < p.born
  RETURN count(other) AS youngerPersonCount
}
RETURN p.name, youngerPersonCount
```

Sous requête avec CALL et agrégation

Requête equivalent sans CALL et avec regroupement :

```
MATCH (p :Person)
WHERE p.born IS NOT NULL
OPTIONAL MATCH (other :Person)
WHERE other.born < p.born
RETURN p.name, count(other) AS
youngerPersonCount
```

Sous requête avec CALL et agrégation

Attention à positionner le WHERE sur le pattern qu'il est censé filtrer :

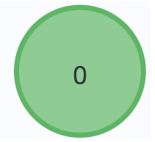
```
MATCH (p :Person)
OPTIONAL MATCH (other :Person)
WHERE p.born IS NOT NULL
AND other.born < p.born
RETURN p.name, count(other) AS youngerPersonCount
```

Le résultat contiendra des noms de personnes dont la date de naissance n'est pas renseignée.

Sous requête avec CALL et mise à jour

CALL peut permet d'accéder aux étapes intermédiaires d'une mise à jour

```
UNWIND [0, 1, 2] AS x
CALL {
  MATCH (n:Counter)
    SET n.count = n.count + 1
  RETURN n.count AS innerCount
}
RETURN x, innerCount
```

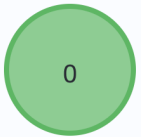


x	innerCount
0	1
1	2
2	3

Sous requête avec CALL et mise à jour

CALL peut permet d'accéder aux étapes intermédiaires d'une mise à jour

Sans CALL les mises à jours intermédiaires sont perdues



UNWIND [0, 1, 2] AS x

MATCH (n:Counter)

x	n
0	(:Counter {count: 0})
1	(:Counter {count: 0})
2	(:Counter {count: 0})

SET n.count = n.count + 1

RETURN x, n.count

x	n.count
0	3
1	3
2	3

Call et procédures

Aussi utilisée pour appeler des procédures (c.f. prochains cours)

CALL db.labels ()

CALL db.propertyKeys()

CALL db.schema.visualization()

CALL db.indexes()

CALL... YIELD var appelle la procedure et retourne les résultats "unwinded" dans la colonne var

Exemple :

```
CALL db.labels() YIELD label
RETURN count(label) AS numLabels
```