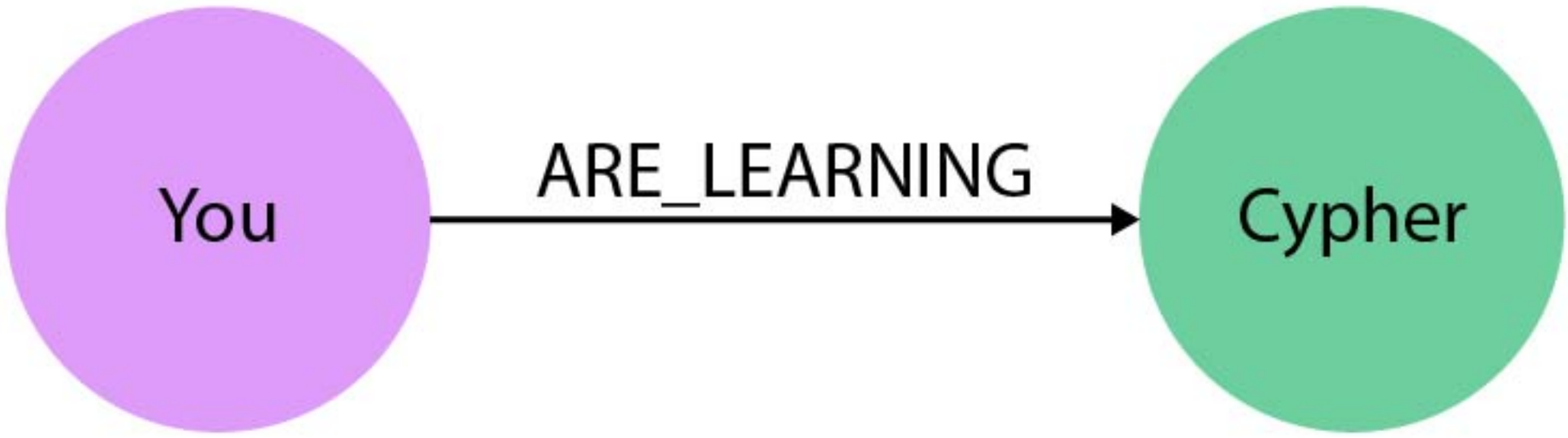
Bases de données spécialisées

Partie 2 : Bases de données orientées graphes
neo4j et Cypher



Amélie Gheerbrant
IRIF, Université Paris Diderot
amelie@irif.fr

Neo4j et Cypher

- Neo4j est le sgbd graphe le plus utilisé :

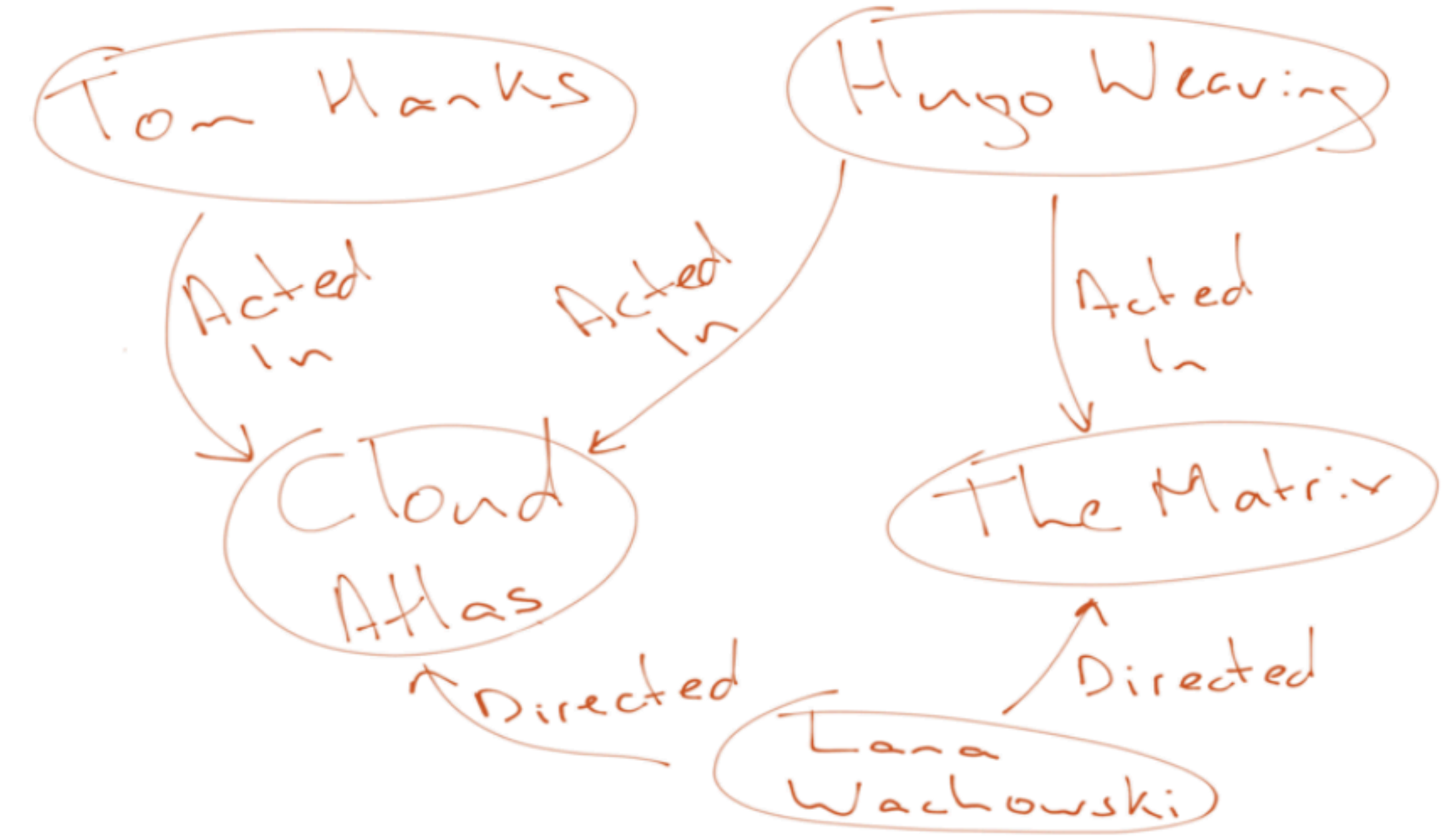
https://db-engines.com/en/ranking/graph+dbms

- 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

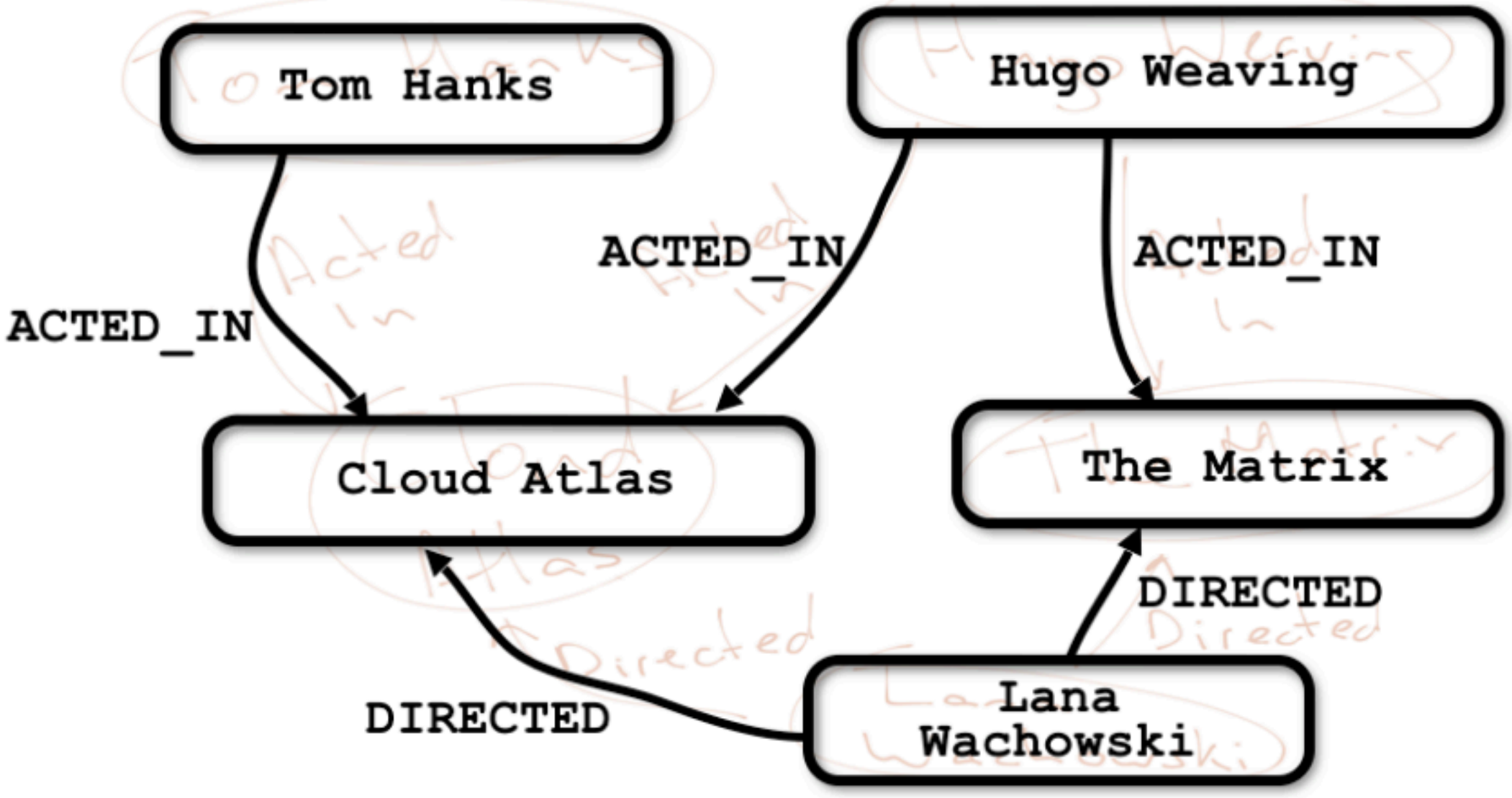
Whiteboard friendliness

Matrix - whiteboard model



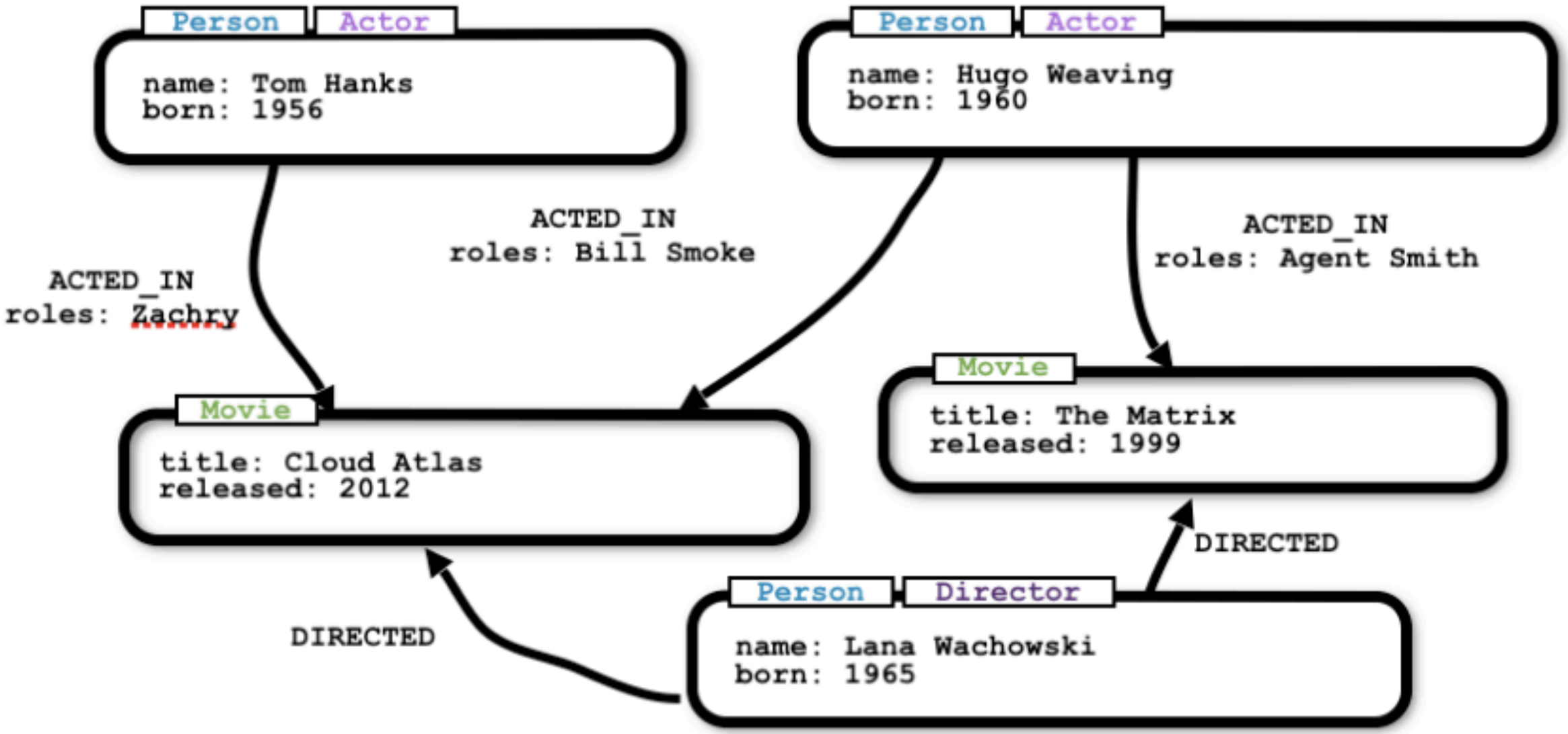
Whiteboard friendliness

Matrix - match node and relationship format of property graph model

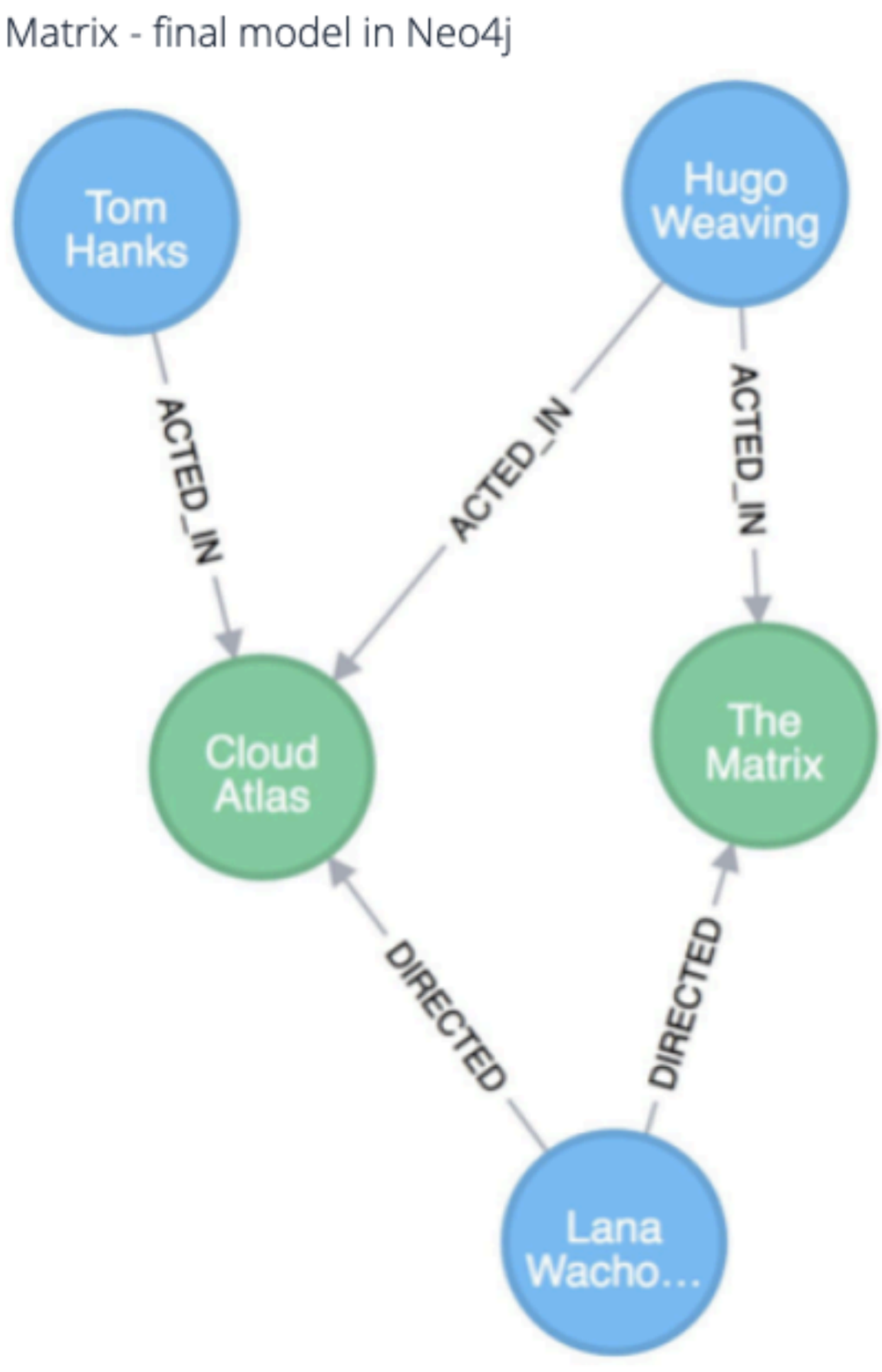


Whiteboard friendliness

Matrix - add labels and properties

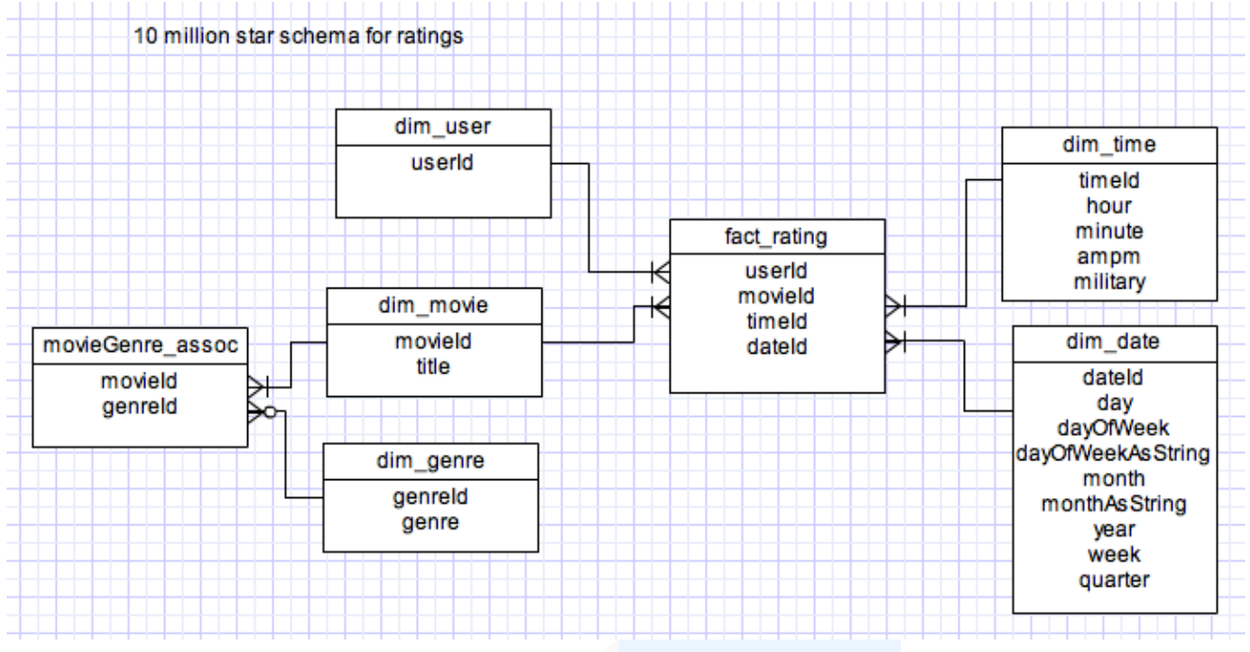


Whiteboard friendliness



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

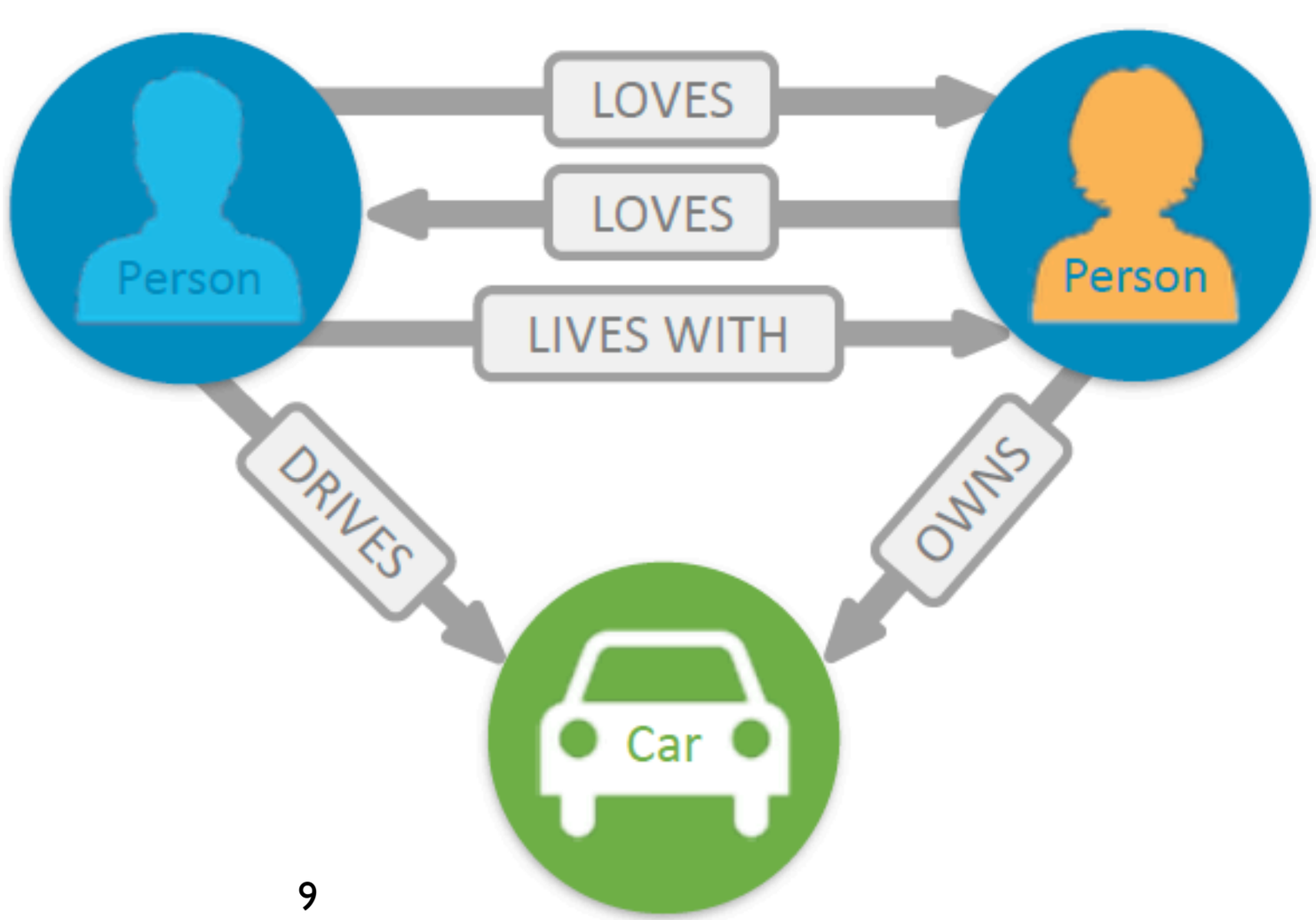
The #1 Database for Connected Data

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

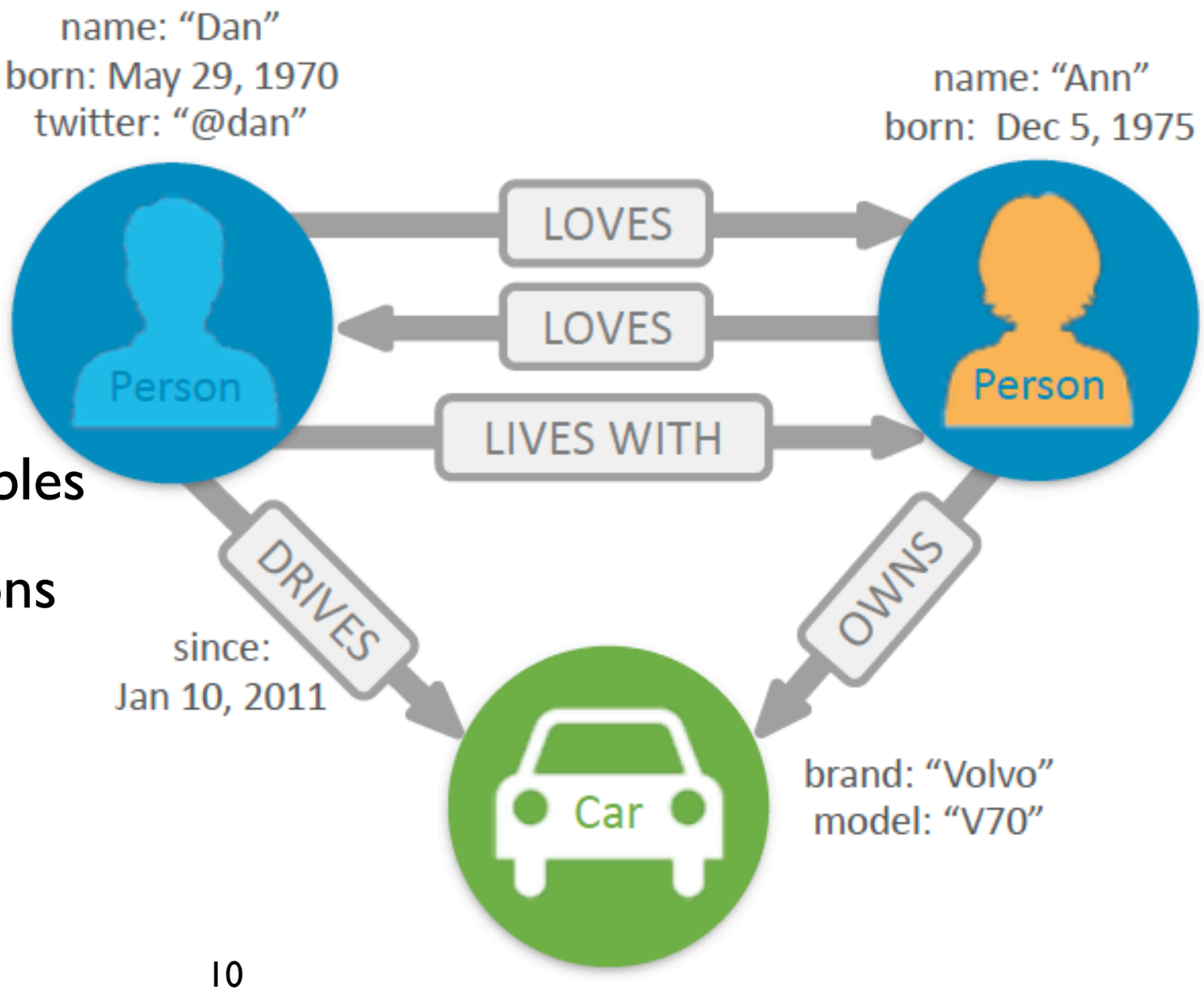
Les composants du modèle de graphe de propriété

- 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

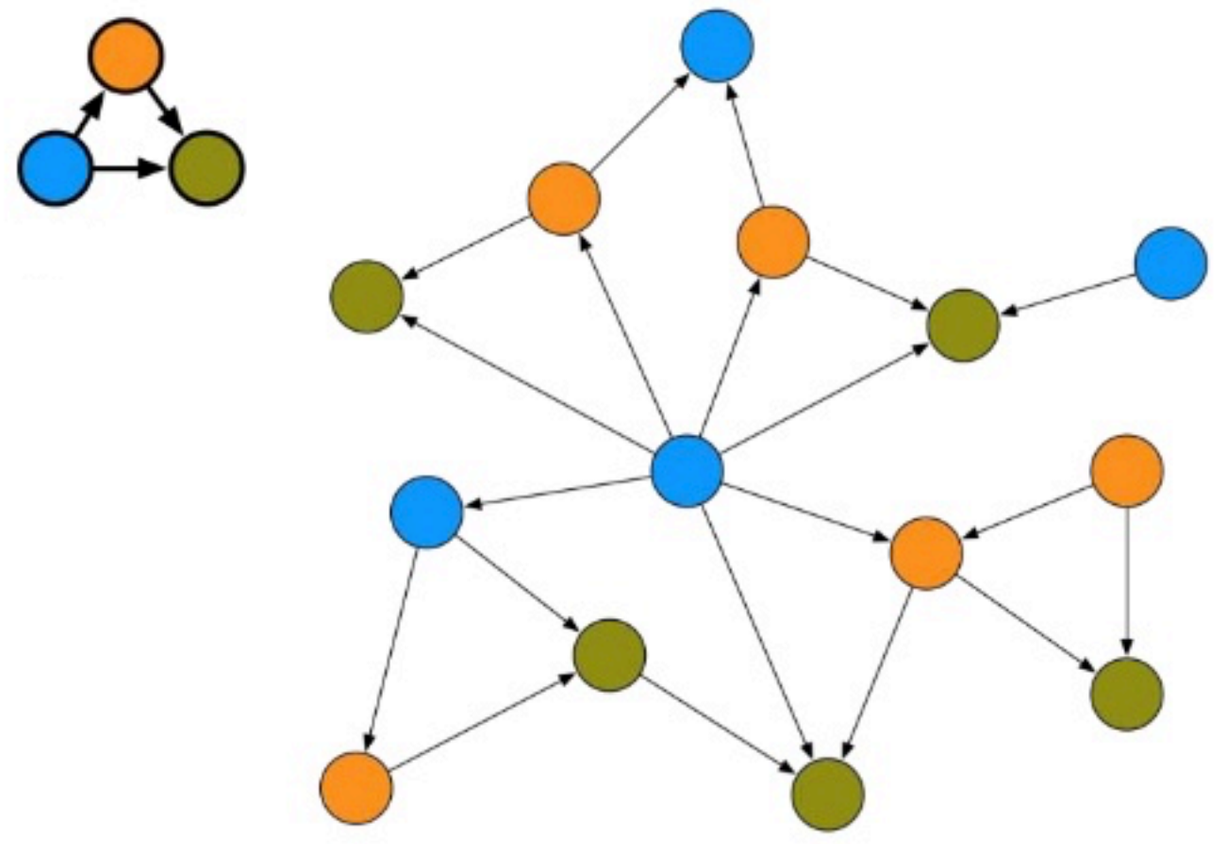
- En particulier pour les requêtes
  - récursives
  - pouvant accepter des chemins de longueurs différentes

- Les patterns de graphes sont plus intuitifs et déclaratifs que les jointures

- SQL ne peut pas retourner de valeurs sur les chemins

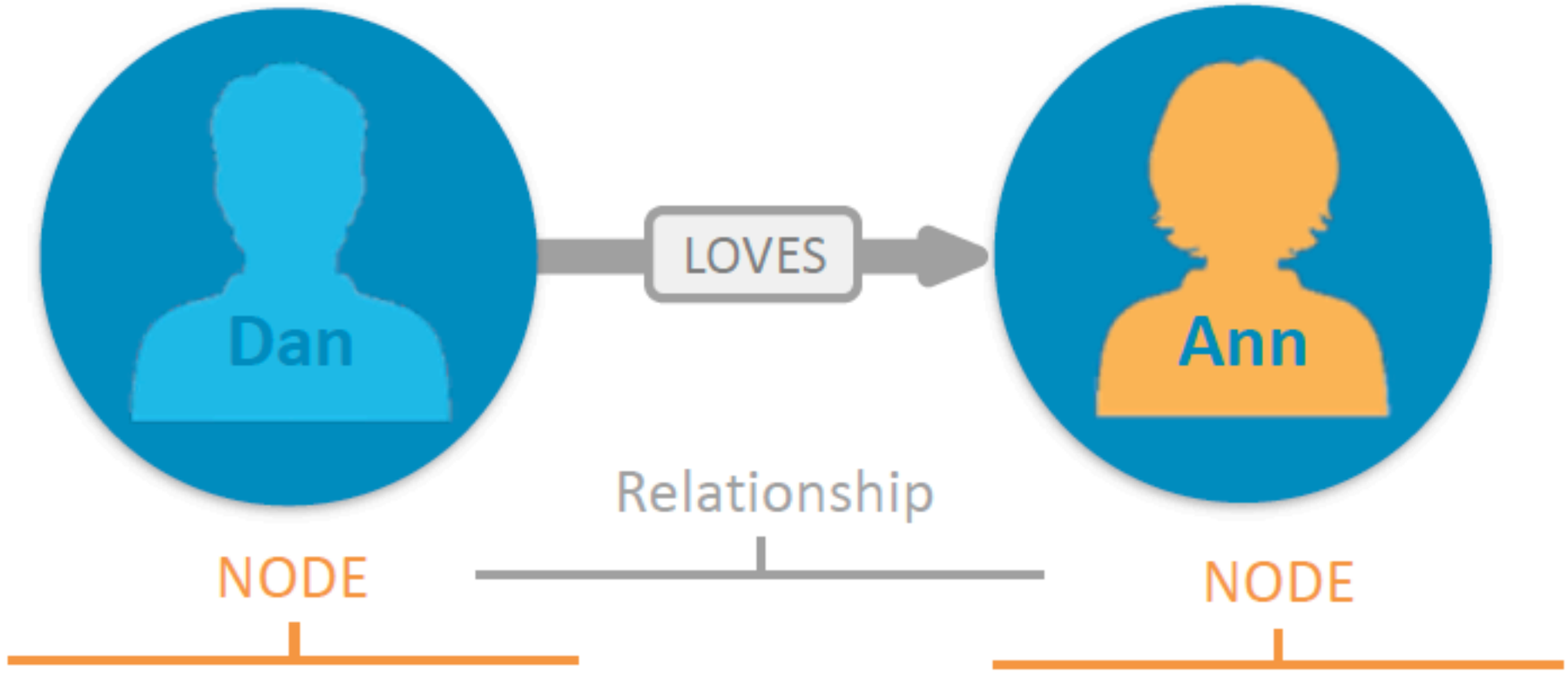
Cypher

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

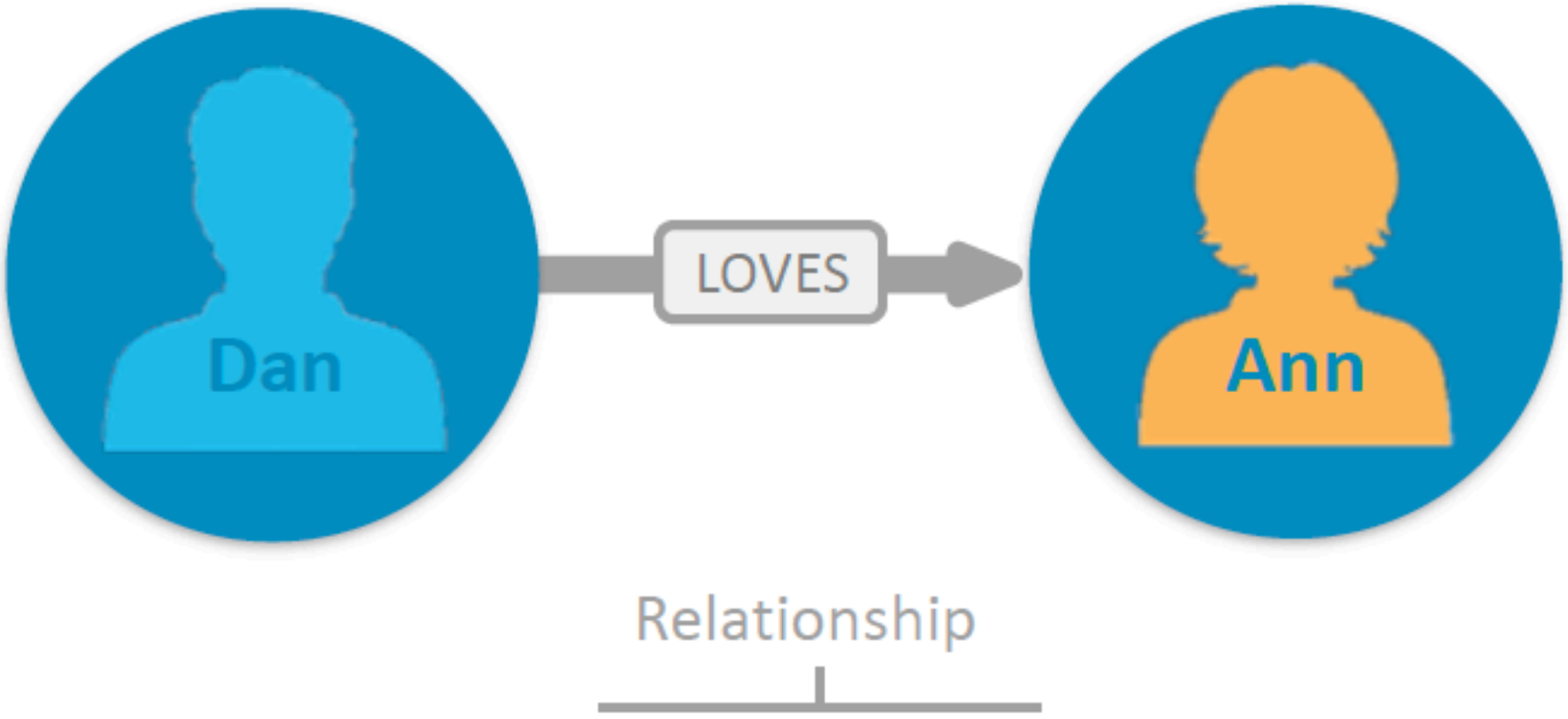


(Pattern matching = « *recherche de motif* » en français)

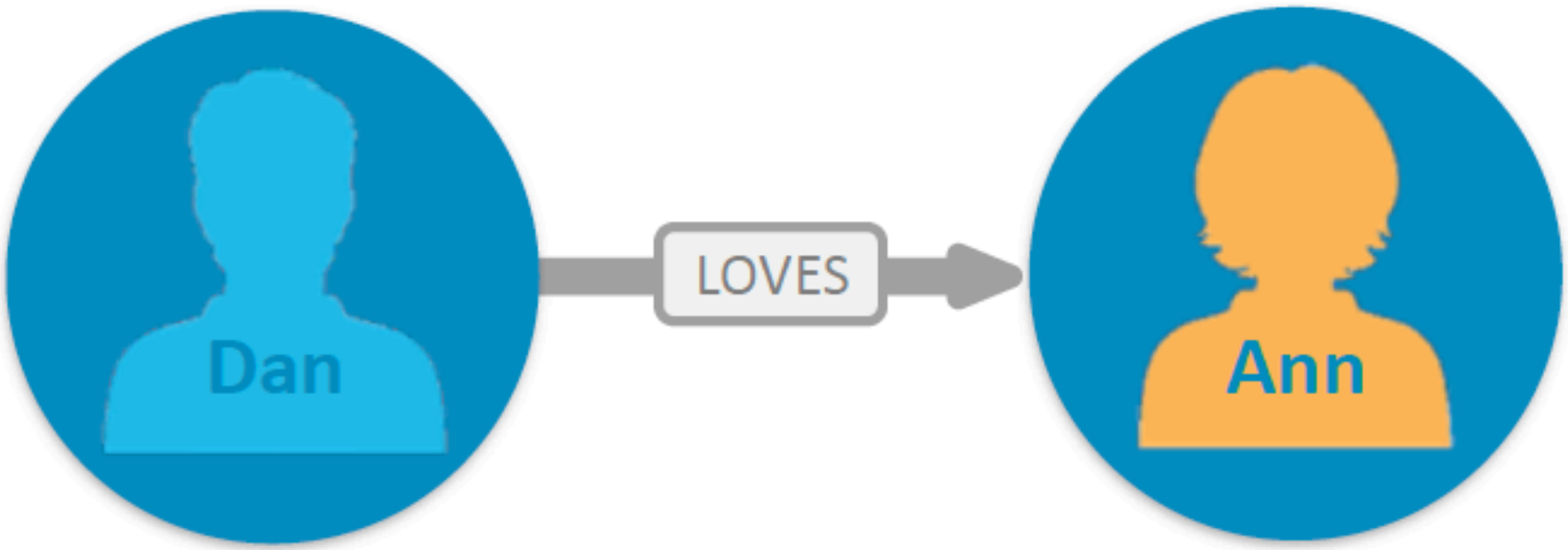
Les patterns dans notre modèle de graphe



Cypher : expression des patterns de graphe



Cypher : expression des patterns de graphe

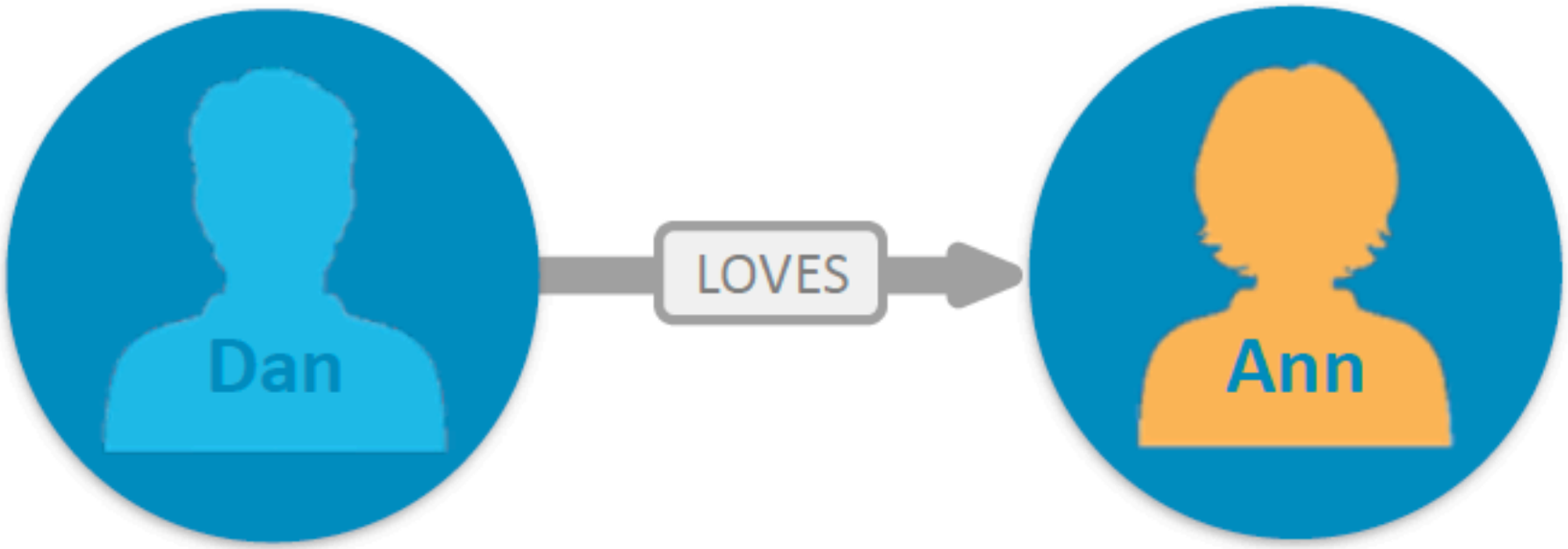


Relationship

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

LABEL PROPERTY LABEL PROPERTY

Cypher : expression des patterns de graphe

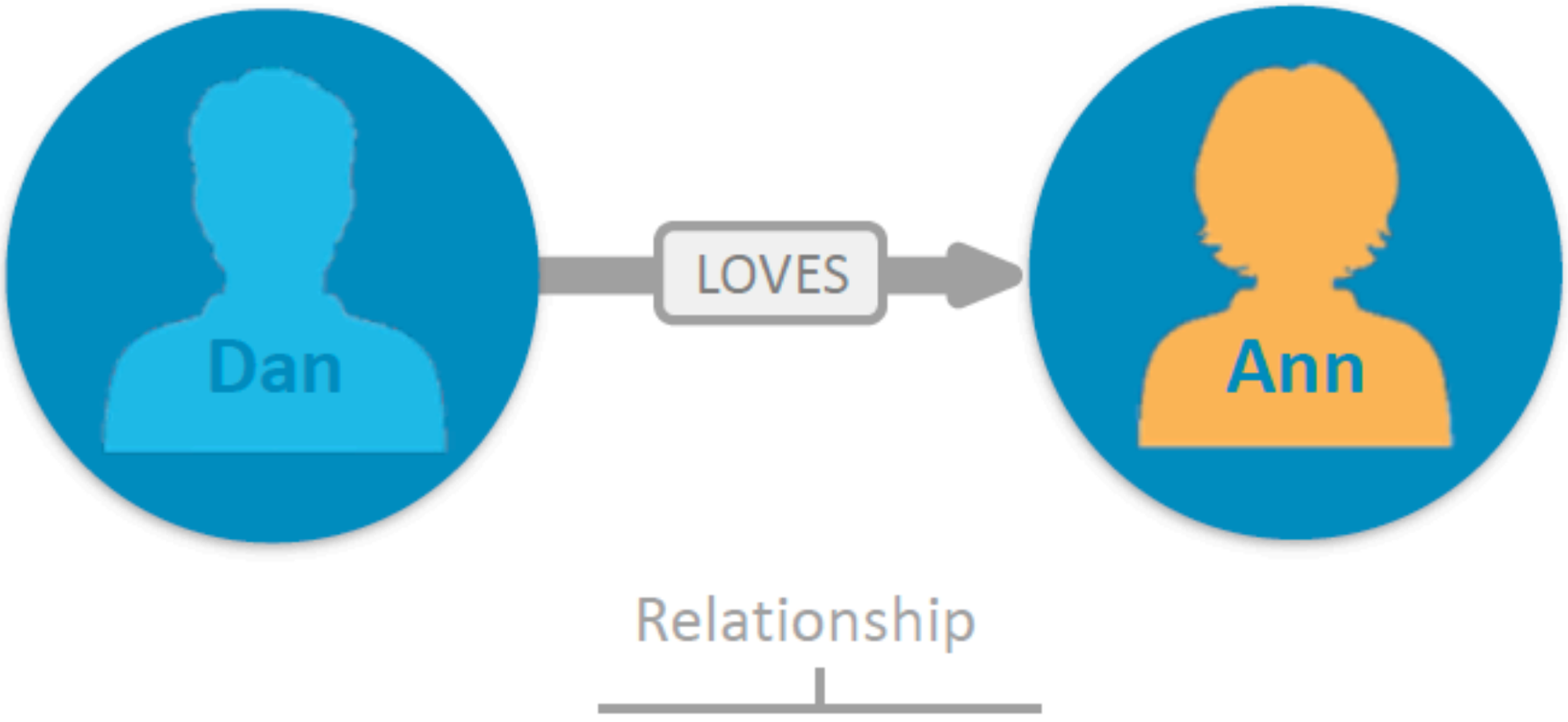


NODE — Relationship — NODE

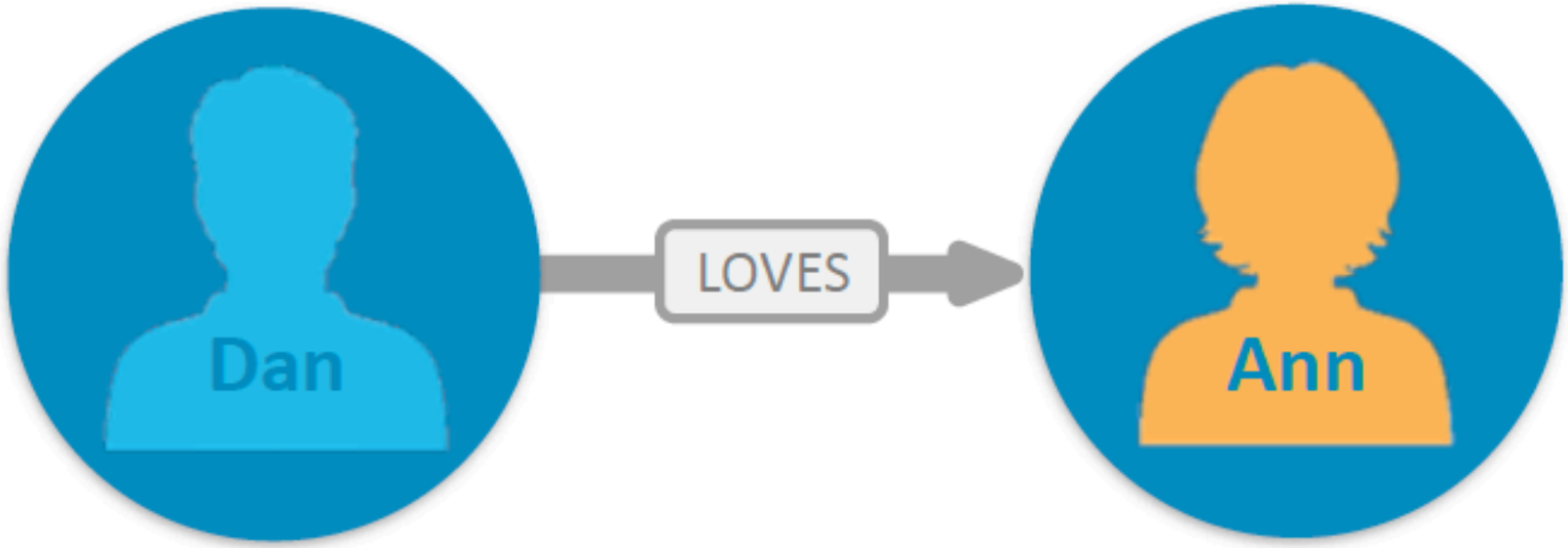
```
(:Person { name:"Dan"} ) -[:LOVES]-> (:Person { name:"Ann"} )
```

LABEL — PROPERTY — LABEL — PROPERTY

Cypher : création des patterns de graphe



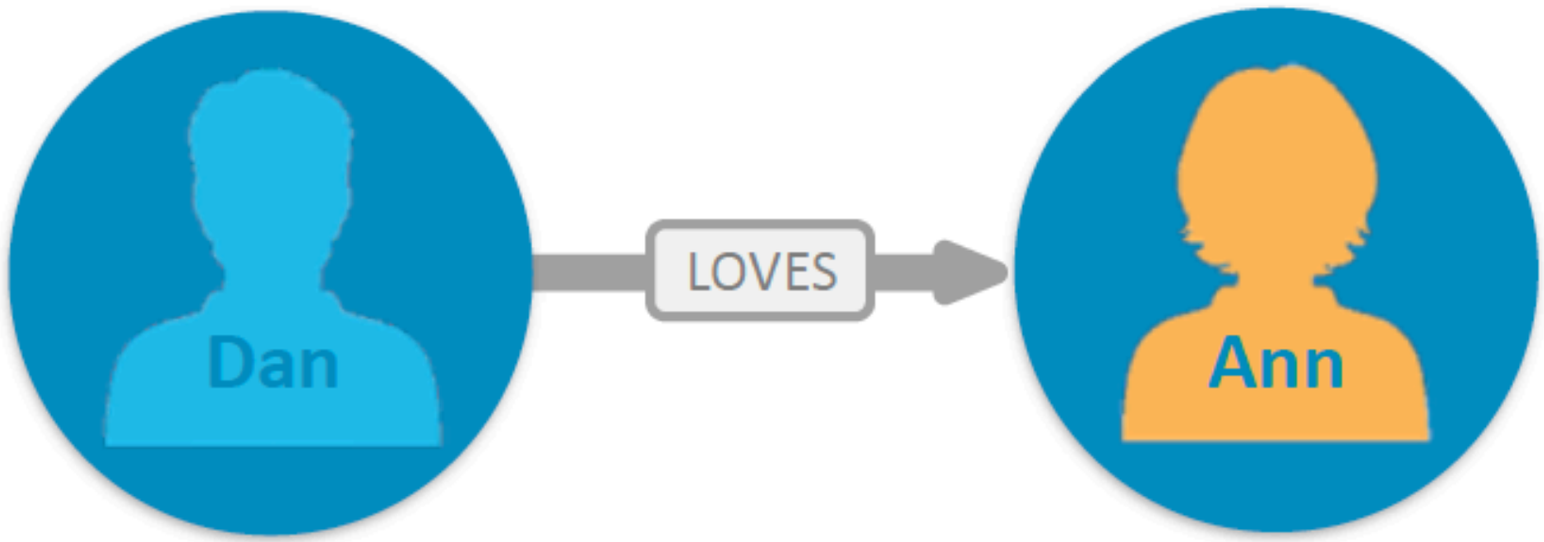
Cypher : création des patterns de graphe



Relationship

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

Cypher : création des patterns de graphe

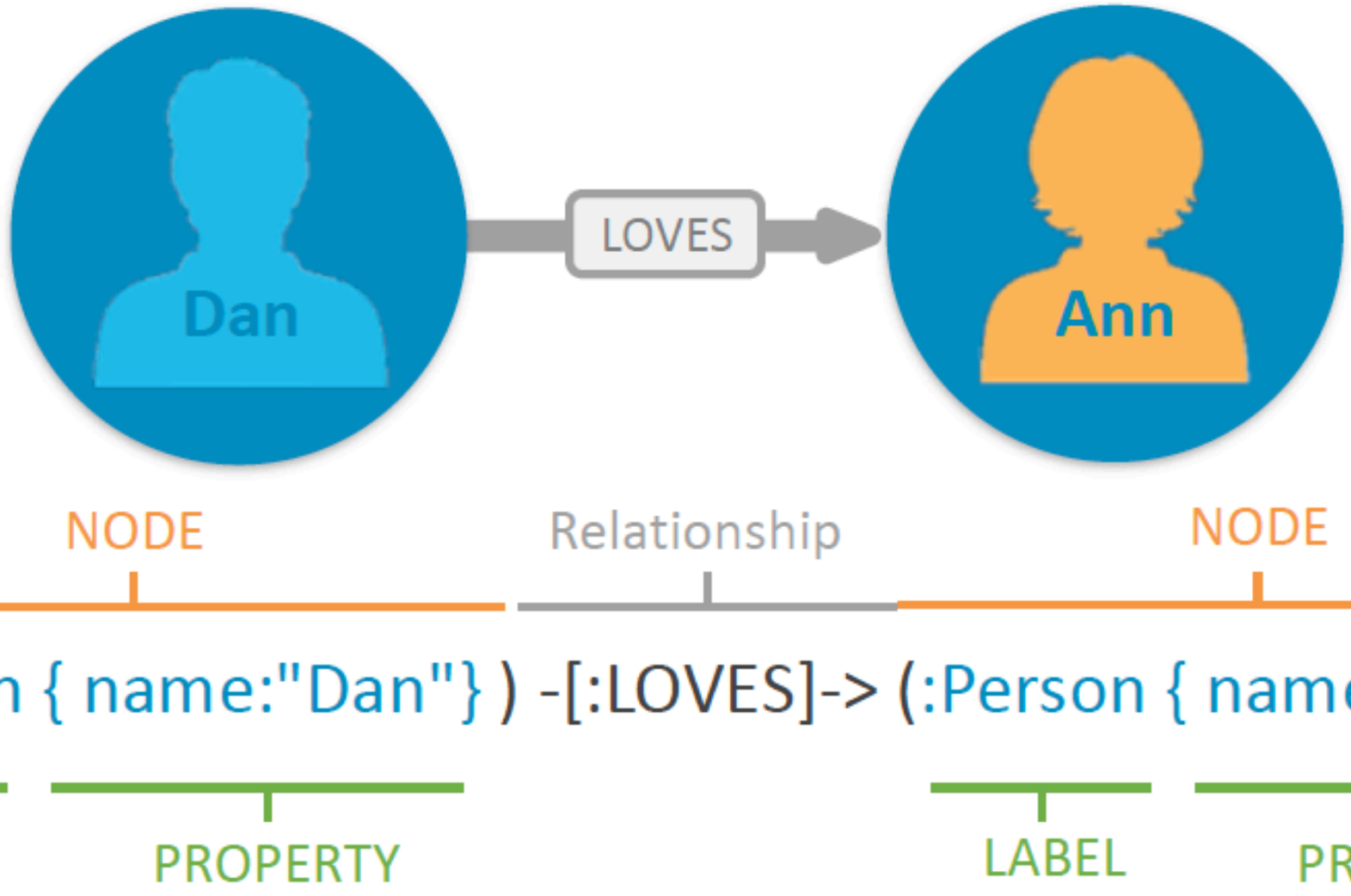


Relationship

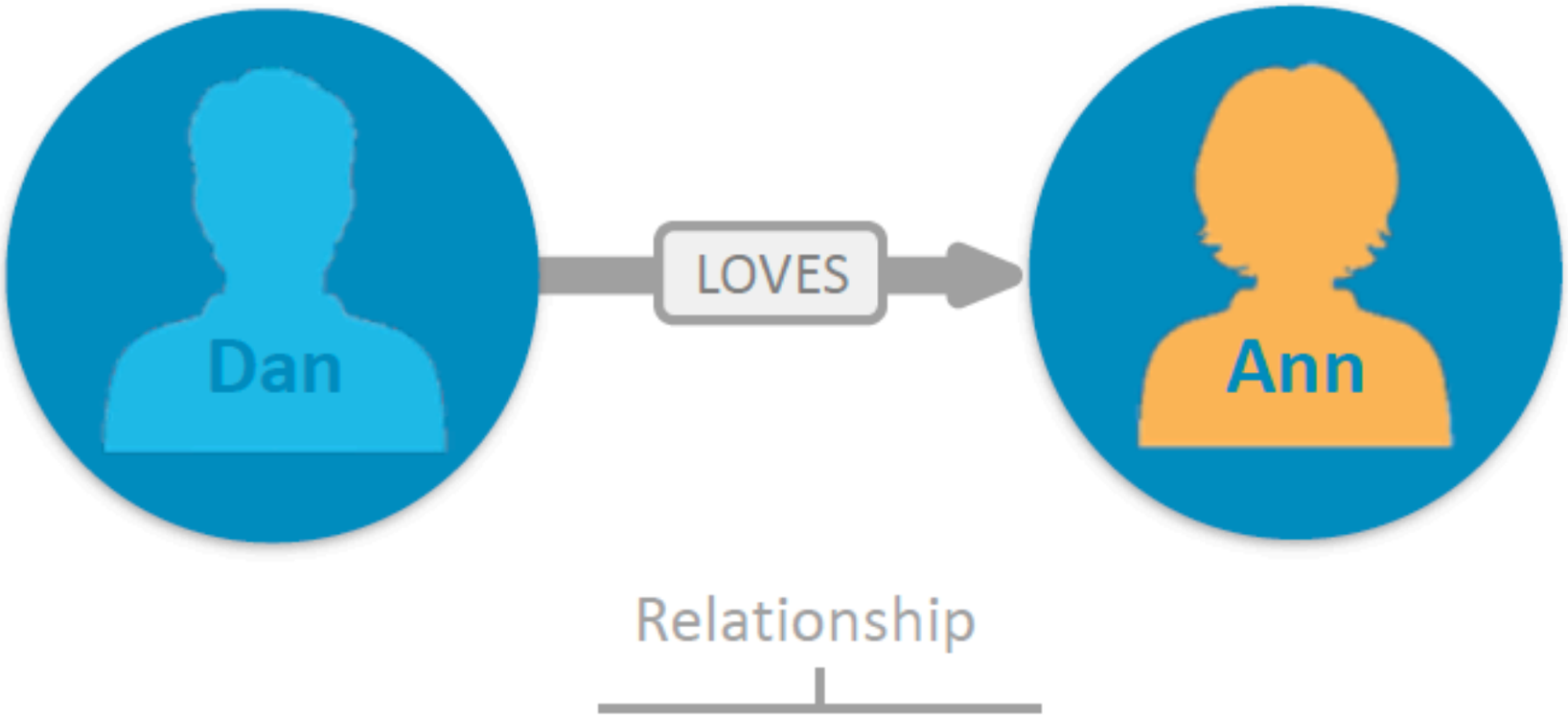
```
CREATE (:Person { name:"Dan"} ) -[:LOVES]-> (:Person { name:"Ann"} )
```

LABEL PROPERTY LABEL PROPERTY

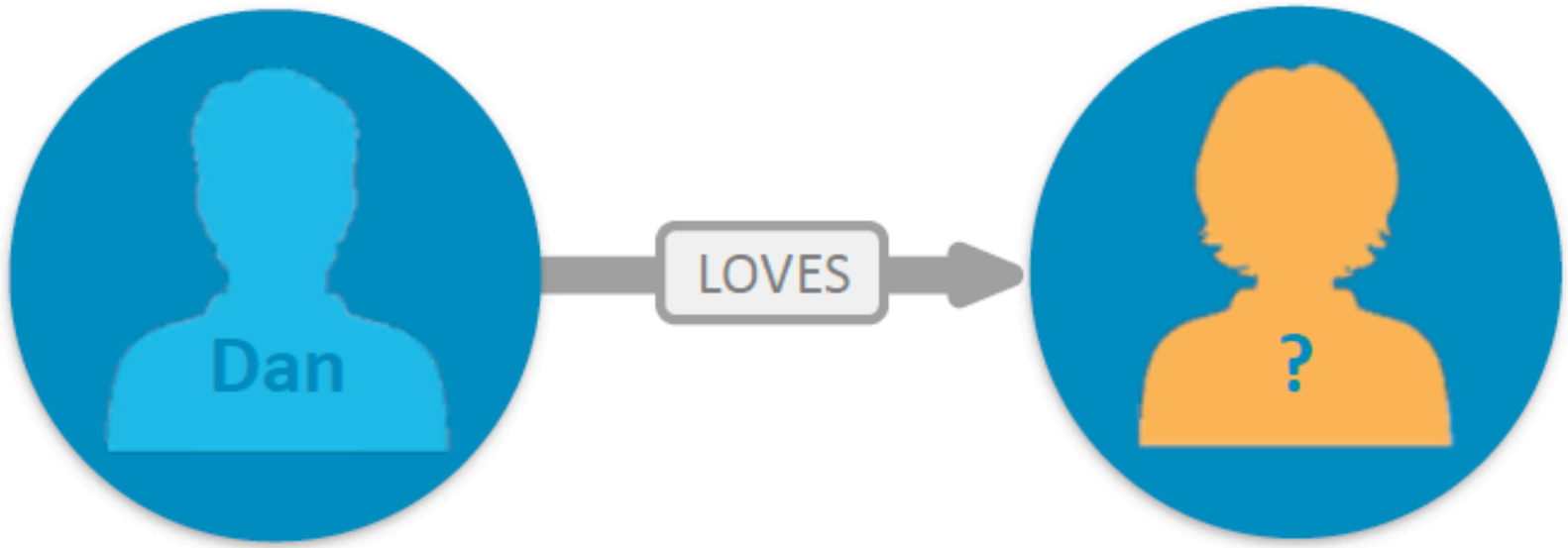
Cypher : création des patterns de graphe



Cypher : match des patterns de graphe



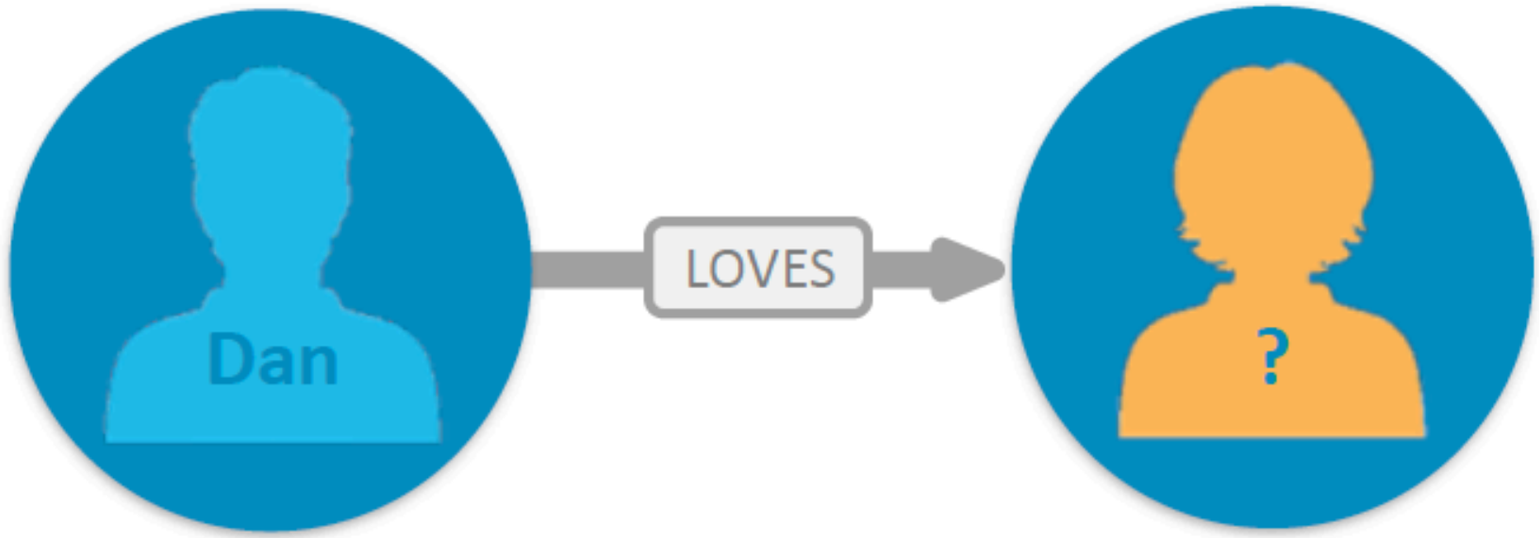
Cypher : match des patterns de graphe



Relationship

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

Cypher : match des patterns de graphe

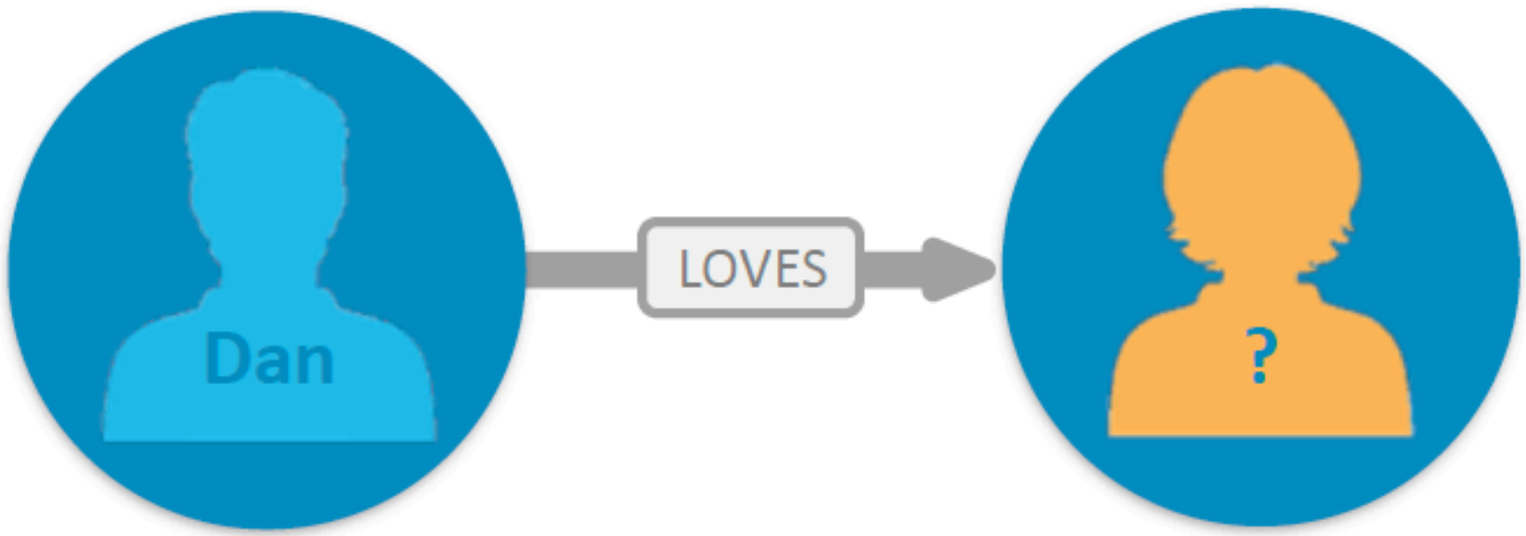


Relationship

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

LABEL PROPERTY VARIABLE

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 noeuds

- Les noeuds
  - dessinés avec des parenthèses

( )

Les noeuds

- Les noeuds
  - dessinés avec des parenthèses, peuvent avoir plusieurs types

(:type1 :type2)

Les relations

- Les relations
  - dessinés avec des flèches, détails entre crochets (un seul type)

```
- - >

- [:DIRECTED] ->
```

Les patterns

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

    la spécification de directions avec > et < est optionnelle

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

Les composants d'une requête Cypher

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

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

m est une variable

: Movie est un label de noeud

Les composants d'une requête Cypher

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

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

p, r et m sont des variables

: ACTED_IN est un type de relation

Les composants d'une requête Cypher

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

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

* retourne tous les noeuds, relations et chemins dans la requête

: ACTED_IN est un type de relation

Les composants d'une requête Cypher

```
MATCH path = (p : Person) - [r : ACTED_IN] - > (m : Movie)
RETURN path
```

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

path est une variable

: ACTED_IN est un type de relation

Graphes de résultats versus tableaux de résultats

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

Graphes de résultats versus tableaux de résultats

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

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

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
```

Partie de MATCH, OPTIONAL MATCH et WITH : ajoute des contraintes au pattern ou filtre les résultats passés à WITH

Opérateurs : NOT, OR, XOR, AND, STARTS, CONTAINS, ENDS WITH, AS

La sous clause WHERE

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

Partie de MATCH, OPTIONAL MATCH et WITH : ajoute des contraintes au pattern ou filtre les résultats passés à WITH

Opérateurs : NOT, OR, XOR, AND, STARTS, CONTAINS, ENDS WITH, AS

La sous clause WHERE

MATCH (m : Person)

WHERE m.name ENDS WITH ‘li’

RETURN m.name

Partie de MATCH, OPTIONAL MATCH et WITH : ajoute des contraintes au pattern ou filtre les résultats passés à WITH

Opérateurs : NOT, OR, XOR, AND, STARTS, CONTAINS, ENDS WITH, AS

La sous clause WHERE

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

Partie de MATCH, OPTIONAL MATCH et WITH : ajoute des contraintes au pattern ou filtre les résultats passés à WITH

Opérateurs : NOT, OR, XOR, AND, STARTS, CONTAINS, ENDS WITH, AS

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
```

Les sous clauses SKIP et LIMIT

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

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

La sous clause ORDER

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

Modificateur par défaut : ASC (ordre croissant)

DESC : ordre décroissant

Souvent utile en combinaison avec LIMIT

Chaîner plusieurs types

```
MATCH (m : Person { name: 'Clint Eastwood' } )-
[:ACTED_IN | DIRECTED]->(n)

RETURN n
```

On peut chaîner plusieurs types si on veut matcher sur plus d'un

Ne s'utilise qu'avec MATCH

Pas de sens avec CREATE ou MERGE

Pattern matching de longueur variable

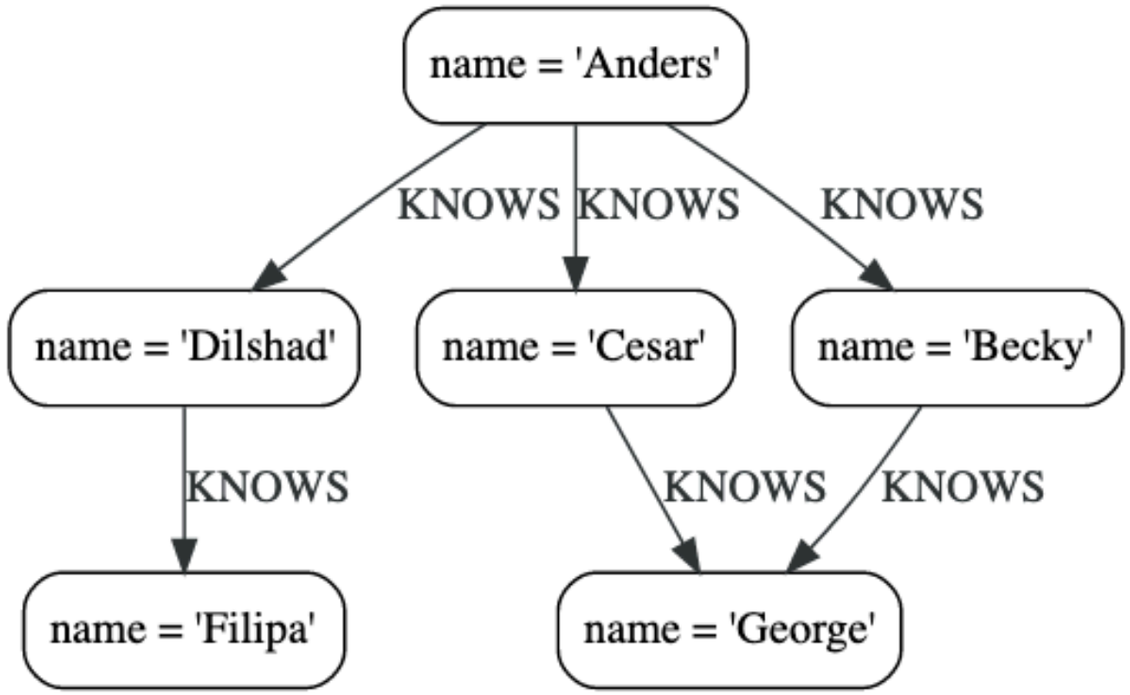
Au lieu de :

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

On peut écrire :

```
MATCH (me) - [: knows*2] - > (remote_friend)
WHERE me.name = `Filipa’
RETURN remote_friend
```

Pattern matching de longueur variable



Query **cypher** Copy to Clipboard Run in Browser

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

Table 1. Result

remote_friend.name
"Dilshad"
"Anders"

2 rows

Pattern matching de longueur variable

De façon générale :

MATCH (me) - [: knows*min...max] - > (remote_friend)

WHERE me.name = `Filipa'

RETURN remote_friend

Et pour les chemins de longueur arbitraire :

MATCH (me) - [: knows*] - > (remote_friend)

WHERE me.name = `Filipa'

RETURN remote_friend

(par défaut min = 1 et max = +∞)

Plus court chemin

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

Retourne un plus court chemin

Implémentation limitée aux chemins de longueur <= à 15

Plus court chemin avec prédicats

```
MATCH (m: Person{name: 'Martin Sheen'}),(o: Person{name : 'Oliver Stone'}),
p = shortestPath((m) -[*] - (o))
WHERE none (r IN relationships(p) WHERE type(r) = 'FATHER')
RETURN p
```

none() s'applique aux relations du chemin

Plus court chemin avec prédicats

```
MATCH (m: Person{name: 'Martin Sheen'}),(o: Person{name : 'Oliver Stone'}),
p = shortestPath((m) -[*] - (o))
WHERE all (r IN relationships(p) WHERE type(r) = 'ACTED_IN')
RETURN p
```

all() s'applique aux relations du chemin

Plus courts chemins

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

Retourne tous les plus courts chemins

Implémentation limitée aux chemins de longueur <= à 15

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

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

D'autres sémantiques existent. (c.f., manuel Cypher p7)

La clause CREATE

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

La clause SET

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

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

RETURN m
```

La clause CREATE

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

La clause MERGE

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

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
```

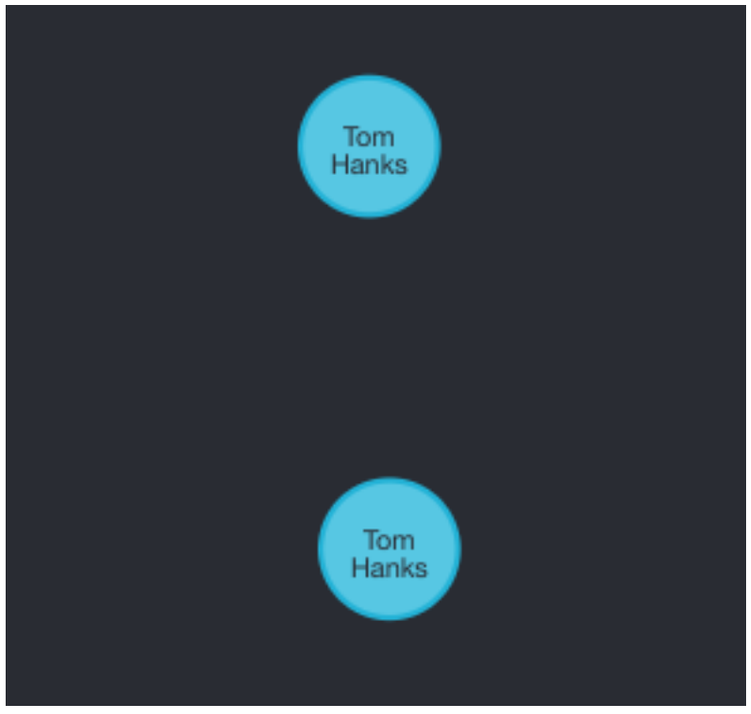
Il n'y a pas de noeud a: Person avec à la fois name: 'Tom Hank' et oscar: true dans le graphe.

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

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'})
SET p.oscar = true
RETURN p
```

Ajoute une propriété au noeud existant

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
```

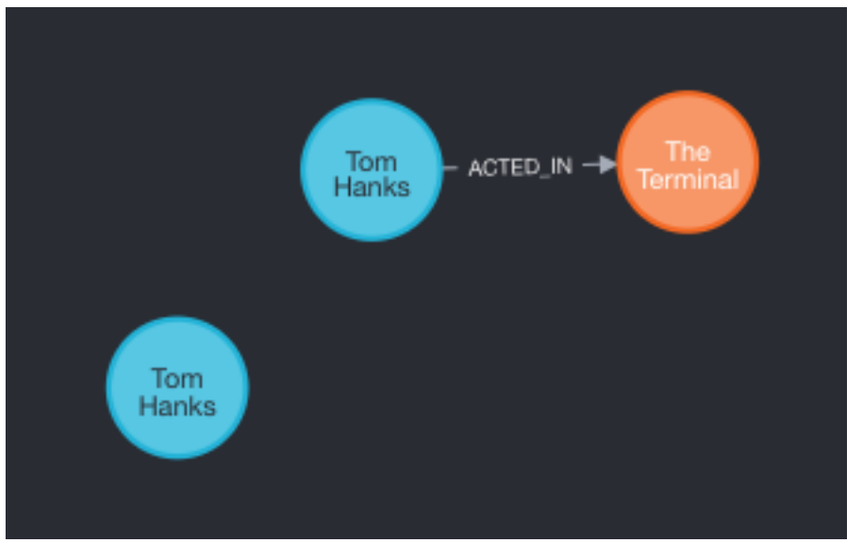
Il n'y a pas de noeud a: Movie avec pour titre : 'The Terminal' dans le graphe, mais il y a un noeud :Person avec name: 'Tom Hanks'

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
```

Un nouveau noeud est créé…



La clause MERGE

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

Pour fusionner plutôt le nouveau noeud avec le noeud existant

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
```

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 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

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

Pour supprimer toutes les propriétés, la valeur retournée pour p.age sera alors nulle.

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(n)
```

Pour supprimer des labels sur un noeud.

L'ensemble de labels retourné sera vide [ ]

La clause WITH

WITH sert à enchaîner les clauses (pipe) : les variables doivent être incluses dans le WITH pour être visibles dans la clause suivante

```
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

La clause WITH

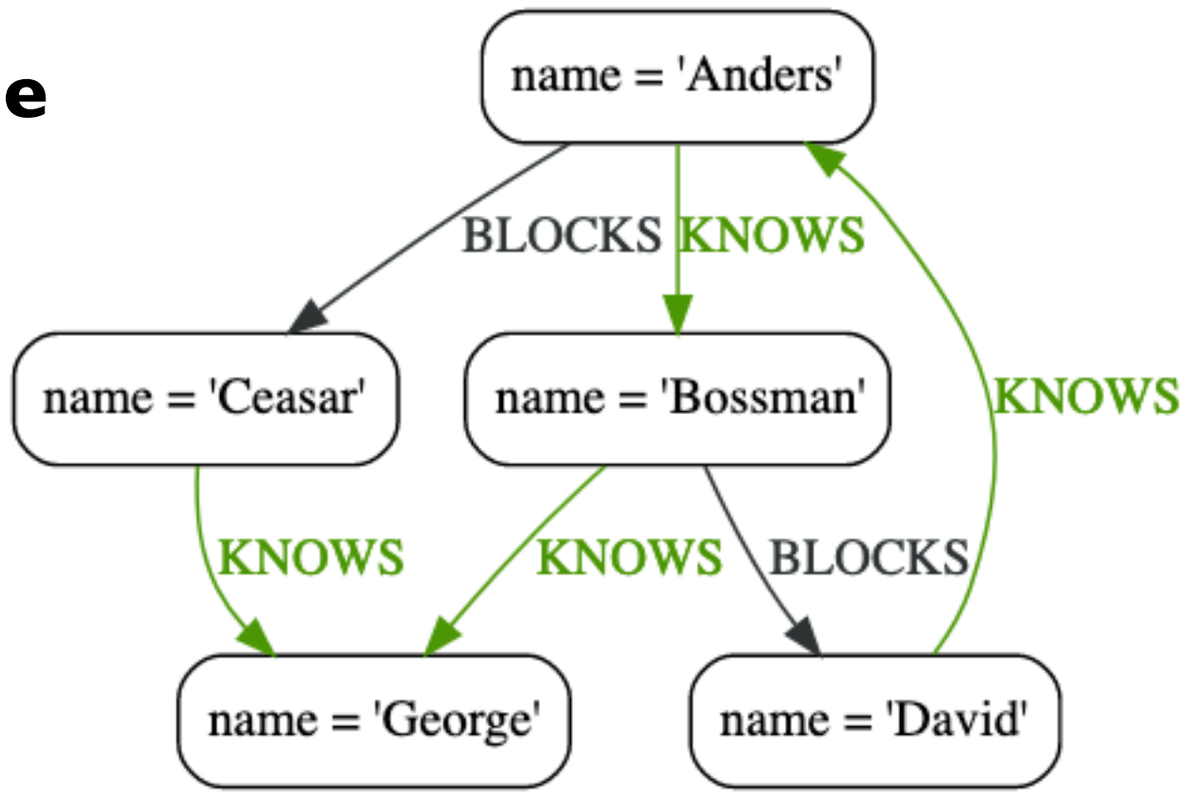
Les résultats agrégés doivent passer par un WITH pour être filtrés par le WHERE :

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

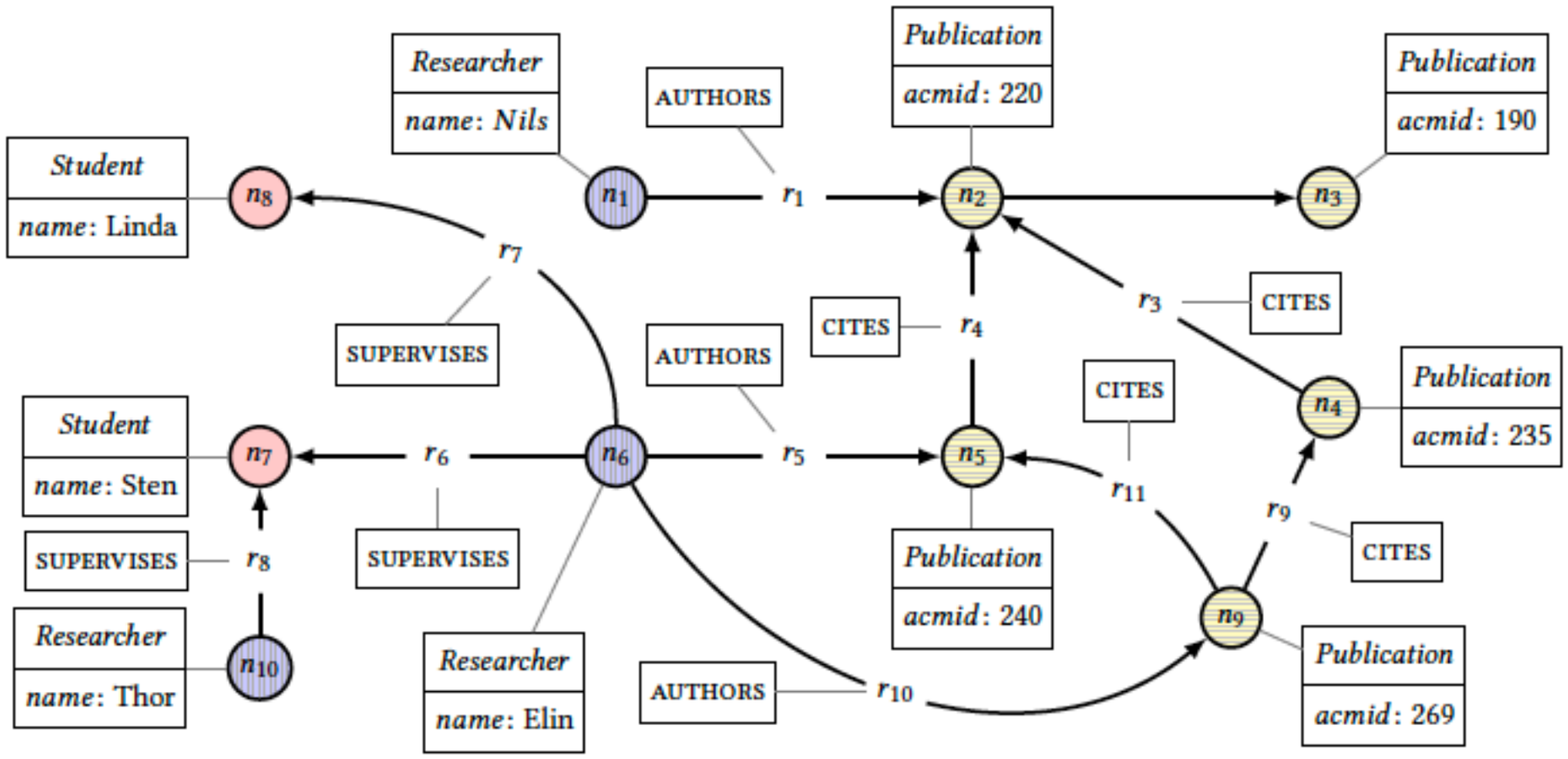
Table 1. Result

otherPerson.name
"Anders"

1 row



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

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:

Identifiants de noeuds

```
MATCH (n : city)
WHERE id(n)=23
DETACH DELETE n
```

On peut y accéder directement mais le sgbd réutilise ses identifiants internes quand les noeuds et relations sont supprimés, donc risqué..

On verra au prochain cours comment attribuer des contraintes d'unicité

Plan du troisième cours

- Syntaxe Cypher miscellaneous (listes, fonctions, etc)

- Contraintes et indexes

- Optimisation manuelle de requête et plans d'exécution

Syntaxe Cypher miscellaneous (listes, fonctions, etc.)

Opérateurs de liste

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

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

```
WITH [1,2,3]+[4,5,6] as nblist
UNWIND nblist as nb
WITH nb
WHERE nb IN [2,3,8]
RETURN nb
```

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

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

```
RETURN [x IN range(0,10) WHERE x%2 = 0 | x^3] AS
result
```

```
MATCH (a: Person { name :  'Keanu Reeves' })

RETURN [(a)- - >(b) WHERE b: Movie  | b.released] AS
years
```

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

Fonction collect() sur les listes

LA même requête sans doublons :

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

Variante (pas très naturelle) avec CALL :

```
CALL {
  RETURN [(a: Person {name: ‘Keanu Reeves'})- - >(b)
  WHERE b: Movie  | b.released] AS years
}
UNWIND years as y
RETURN collect(distinct y)
```

Clause de sous requête avec CALL

CALL { … } évalue une sous requête, typiquement post union ou agrégation

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

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

Sous requête avec CALL post UNION

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

```
CALL {
MATCH (p: Person) WHERE p.born IS NOT NULL
RETURN p ORDER BY p.born ASC LIMIT 1
  UNION
MATCH (p: Person) WHERE p.born IS NOT NULL
RETURN p ORDER BY p.born DESC LIMIT 1
}
RETURN p.name, p.born ORDER BY p.name
```

Attention aux nulls ! Une date de naissance non renseignée est classée à la fin par ORDER BY (considérée nulle)

Sous requête avec CALL post UNION

```
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

Attention, 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). Tester :

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

Sous requête avec CALL post UNION

```
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

Pour chaque personne, nombre de personnes plus jeunes

```
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
```

Call et procédures

Clause d'écriture / lecture (au même titre que merge) :

CALL [... YIELD] : invoque une procédure déployée dans la BD et retourne les résultats

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

CALL db.labels

CALL db.propertyKeys

CALL db.labels

CALL db.schema.visualization

CALL db.indexes

Quelques fonctions

Multitude de fonctions (chapitre 4 du manuel de Cypher) :
scalaires, mathématiques, temporelles, spatiales, user defined, etc…

Fonctions de prédicat

Teste

all() : si le prédicat vaut pour tous les éléments d'une liste

any() : si le prédicat vaut pour au moins un élément d'une liste

exists() : s'il y a un match pour le pattern ou si la propriété vaut pour le noeud ou la relation

none() : si le prédicat ne vaut pour aucun élément de la liste

single() : si le prédicat vaut pour exactement un élément de la liste

```
MATCH p=(a)-[:FOLLOWS*1..3]->(b)
WHERE ALL(x IN nodes(p) WHERE x.name CONTAINS 's')
RETURN p
```

Fonctions de prédicat

Teste

all() : si le prédicat vaut pour tous les éléments d'une liste

any() : si le prédicat vaut pour au moins un élément d'une liste

exists() : s'il y a un match pour le pattern ou si la propriété vaut pour le noeud ou la relation

none() : si le prédicat ne vaut pour aucun élément de la liste

single() : si le prédicat vaut pour exactement un élément de la liste

```
MATCH (n: Person)
WHERE NOT EXISTS ((n)-[:DIRECTED]-())
RETURN count(*)
```

Fonctions d'agrégation

Usuelles : avg(), count(), max(), min(), sum(), stDev(), collect()…

```
MATCH (p: Person)
RETURN collect(DISTINCT p.born)
```

Agrégation par clef de groupage :

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

```
CALL{
MATCH (n: Person)
RETURN DISTINCT n.born as date ORDER BY n.born
}
RETURN collect(date)
```

Fonctions de liste

Renvoient des listes : keys(), labels(), nodes(), relationships()…

```
MATCH (p: Person)
WHERE p.name = 'Clint Eastwood'
RETURN keys(p)
```

```
MATCH path= (p: Person)- -()
WHERE p.name = 'Clint Eastwood'
RETURN relationships(path)
```

```
MATCH (p: Person)
WHERE p.name = 'Clint Eastwood'
RETURN labels(p)
```

```
MATCH path= (p: Person)- -()
WHERE p.name = 'Clint Eastwood'
RETURN nodes(path)
```

Attention : relationships()≠type() ! (type(r) renvoie le type d'une relation r, donc son label, relationships(r) renvoie ses propriétés)

Import de données

LOAD CSV import de données à partir de fichiers CSV

p152 manuel Cypher

USING periodic commit : query hint pour prévenir les dépassements de mémoire quand on importe de grosses quantités de données avec LOAD CSV

Contraintes et Indexes

Contraintes d'intégrité

- Flexibilité des BD graphes : notion de schéma limitée
- Avantage à contrebalancer avec certains inconvénients
- Risque de redondance et d'incohérence dans les données
- E.g. deux noeuds (:Person {name: `Tom Hanks'}), noeud (: Movie) avec un arc sortant -[:FOLLOWS]->, etc.
- Pour pallier certains de ces problèmes (mais pas - encore ? - tous), Neo4j permet de définir des contraintes d'intégrité
- Vérifiées à chaque insertion et mise à jour

Contrainte d'unicité de noeud

CREATE CONSTRAINT constraint_name

ON (n :LAbelName)

ASSERT n.propertyName **IS UNIQUE**

Exemple :

CREATE CONSTRAINT unique_isbn

ON (book :Book)

ASSERT book.isbn **IS UNIQUE**

Erreur si :

CREATE (:Book {isbn : '42', title :'The hitchhiker guide to galaxy'), (:Book {isbn : '42', title :'Graph databases')

Contraintes d'unicité avec MERGE

MERGE (book : Book { isbn: '42' })

RETURN book.isbn, book.name

Si le noeud existe déjà, MERGE match simplement avec.

En revanche, échec en cas de match partiel :

MERGE (book : Book { isbn: '42', year : 1979)

RETURN book.isbn, book.name

Préférer :

MERGE (book : Book { isbn: '42', year : 1979)

SET book.year = 1979

Contrainte d'existence de propriété de noeud

```
CREATE CONSTRAINT constraint_name
ON (n :LAbelName)
ASSERT EXISTS(n.propertyName)
```

Exemple :

```
CREATE CONSTRAINT exist_isbn
ON (book :Book)
ASSERT EXISTS (book.isbn)
```

Erreur si :

```
MATCH (book: Book {title : 'The hitchhiker guide to galaxy'})
REMOVE (book_isbn)
```

Contrainte d'existence de propriété de relation

CREATE CONSTRAINT constraint_name

ON ()-[r :Relationship_Type]-()

ASSERT EXISTS(r.propertyName)

Exemple :

CREATE CONSTRAINT exist_day

ON ()-[like :LIKED]-()

ASSERT EXISTS(like.day)

Erreur si :

CREATE (user :USER)-[like:LIKED]->(book:Book)

ou

MATCH (user :USER)-[like:LIKED]->(book:Book)

REMOVE like.day

Contrainte de clef de noeud

CREATE CONSTRAINT constraint_name

ON (n :LAbelName)

ASSERT n.propertyName_1,

...

n.propertyName_k)

IS NODE KEY

Exemple :

CREATE CONSTRAINT key_name

ON (p :Person) **ASSERT** (p.firstname, p.surname) **IS NODE KEY**

Erreur si :

CREATE (p :Person {first name : 'John', age : 34})

ou

CREATE (p :Person {first name : 'John', Surname : 'Doe', age : 34}),

(p :Person {first name : 'John', Surname : 'Doe', age : 40})

Supprimer une contrainte

DROP CONSTRAINT **constraint_name**

Lister les contraintes

```
CALL db.constraints
```

Clauses d'administration

CREATE | DROP | START database

CREATE | DROP INDEX

CREATE | DROP CONSTRAINT

Users, roles, privileges

Indexes

- Index = copie redondante de certaines données
  - recherche plus efficace
  - écriture ralentie
- Deux types d'indexes dans Neo4j :
  - b-tree (adapté à : exact look ups sur tout type de valeur, range scan, full scan, recherche de préfixe), le query planer de Cypher décide de quel index utiliser dans quelle situation
  - full-text (indexation et recherche de texte, seulement string, basé sur un analyseur syntaxique, e.g. stop words etc), interrogés et créés par l'utilisateur via des procédures

e

Indexes

- Pour chaque noeud d'un label donné, on peut créer un index sur une ou plusieurs propriétés :
  - Single property index (sur une seule propriété)

```
CREATE INDEX index_name
FOR (n :labelName)
ON (n.propertyName)
```

le nom doit être unique !

(parmi les noms d'indexes et de contraintes)

  - Composite index (sur plusieurs propriétés)

```
CREATE INDEX [ index_name ]
FOR (n :labelName)
ON (n.propertyName_1,
        …
        n.propertyName_k)
```

Exemples

- Single property index:

CREATE INDEX index_name FOR (n:Person)ON (n.surname)

- Composite index :

CREATE INDEX index_name FOR (n:Person)ON (n.age, n.country)
Attention, ne fonctionnera que sur cette combinaison précise de propriétés…

- Suppression d'un index : DROP INDEX index_name
- Liste des indexes : CALL db.indexes

Indexes de textes (lucene)

Utilisables pour le projet, c.f., manuel p 300

Optimisation manuelle de requête et plans d'exécution

Optimisation de requêtes

- Requêtes Cypher déclaratives, optimisées par le sgbd
- Mais optimisation manuelle : s'assurer que seul le nécessaire est extrait du graphe
- Filtrer les données le plus tôt possible
- Retourner uniquement ce dont on a besoin (e.g., telle propriété plutôt que le noeud entier)
- Limiter autant que possible la taille des patterns de longueur variable
- Utilisez des paramètres autant que possible (à ce sujet : lisez la doc)
- Efficacité mesurée en nombre d'accès disques (db hits)

Profilage de requête

PROFILE : affiche le plan d'execution sans exécuter la requête

EXPLAIN : affiche le plan d'execution et exécute la requête

Règle générale : expliciter les labels et propriétés attendus

Préférer :

MATCH (a :Person {name : 'Clint Eastwood'}) NodeByLabelScan

plutôt que :

MATCH (a {name = 'Clint Eastwood'}) AllNodesScan

Utilisation des indexes

Création d'un index sur une propriété unique :

CREATE INDEX index_person for (n:Person) on (n.name)

Comparer avant et après :

PROFILE MATCH (a :Person {name : 'Clint Eastwood'})

RETURN a

NodeIndexSeek

PROFILE MATCH (a :Person)

WHERE a.name = 'Clint Eastwood'

RETURN a

tests d'égalité dans le WHERE

Utilisation des indexes

Comparaisons multiples dans le WHERE :

```
MATCH (person:Person)
WHERE 1979 < person.born < 1980
RETURN person
```

index simple sur Person(born)

ou

```
MATCH (person:Person)
WHERE 1979 < person.born < 1980
AND EXISTS(person.name)
RETURN person
```

index composite sur Person(born, name)

- Un seul scan d'index

Utilisation des indexes

Test d'appartenance à une liste :

```
MATCH (person:Person)
WHERE person.firstname IN ['Andy', 'John']
RETURN person
```

- Utilisation de l'index sur Person(firstname) - s'il existe -

```
MATCH (person:Person)
WHERE person.age IN [10,20,35]
AND person.country IN ['Sweden', 'USA', 'UK']
RETURN person
```

- Utilisation de l'index sur Person(age, country) - s'il existe -

Utilisation des indexes

Utilisation du prédicat STARTS WITH :

```
MATCH (person:Person)
WHERE person.firstname STARTS WITH 'And',
RETURN person
```

- Utilisation de l'index sur Person(firstname) - s'il existe -

```
MATCH (person:Person)
WHERE person.firstname STARTS WITH 'And'
AND EXISTS (person.surname)
RETURN person
```

- Utilisation de l'index sur Person(firstname, surname) - s'il existe -

Utilisation des indexes

Utilisation du prédicat ENDS WITH :

```
MATCH (person:Person)
WHERE person.firstname ENDS WITH ‘dy’,
RETURN person
```

- Utilisation de l’index sur Person(firstname) - s’il existe -
- Pas optimisé pour =, IN, >, <, STARTS WITH
- Quand même plus rapide que sans index

non supporté pour les indexes composites (réécrit comme test d’existence + filtre)

Idem pour CONTAINS et EXISTS

Recherche de propriété via index

Exemple :

```
CREATE INDEX FOR (person:Person)
ON (person.name)

MATCH (p:Person)-[:ACTED_IN]->(m:Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN p.name, count(m)
```

Tester sans le where

Projection db hits ≠0

Versions précédentes de Neo4j : RETURN p.name induisait un look up du noeud n
Neo4j 4.1.3 : exploite le fait que le stockage de la propriété par l'indexe, pas de look up du noeud

cache[p.name]

Projection : db hits=0

Recherche de propriété via index

La même optimisation fonctionne avec les prédicats suivants :

- Existence WHERE EXISTS(n.name)
- Egalité WHERE n.name = 'Tom Hanks'
- Intervalle WHERE n.uid > 1000 AND n.uid < 2000
- Préfixe. WHERE n.name STARTS WITH 'Tom'
- Suffixe WHERE n.name ENDS WITH 'Hanks'
- Sous-chaîne WHERE n.name CONTAINS 'a'
- Combinaison avec OR de ces prédicats sur la même propriété
  WHERE n.prop < 10 OR n.prop='infinity'
- Automatique en cas de contrainte d'existence sur la propriété

Agrégation et optimisation

La même optimisation fonctionne en présence d'agrégats :

```
PROFILE
MATCH  (p:  Person)
RETURN count(DISTINCT p.name) AS nbOfnames
```

cache[p.name]

NodeIndexScan

Order by et optimisation

La même optimisation fonctionne en présence d'agrégats :

```
PROFILE
MATCH  (p:  Person)-[:ACTED_IN]->(m: Movie)
WHERE p.name STARTS WITH 'Tom'
RETURN p.name, count(m)
ORDER BY p.name
```

Tester sans le where

Sort db hits ≠0

OrderedAggregation versus

EagerAggregation

Tombe bien : l'index stocke les noms en ordre croissant !
Pas besoin de sort (≠Neo4J 3.4)

Un peu moins performant avec ORDER BY DESC

min(), max() et optimisation

Comparer :

```
PROFILE
MATCH  (p:  Person)-[:ACTED_IN]->(m: Movie)
WHERE p.name STARTS WITH ‘’
RETURN min(p.name) AS name
```

NodeIndexSeekByRange

et (bien plus couteux)

```
PROFILE
MATCH  (p:  Person)-[:ACTED_IN]->(m: Movie)
USING INDEX p:Person(name)
WHERE EXISTS(p.name)
RETURN min(p.name) AS name
```

EagerAggregation

(pire sans USING)

min(), max() et optimisation

Optimisation déclenchée avec les prédicats :

- Egalité WHERE n.name = 'Tom Hanks'
- Intervalle WHERE n.uid > 1000 AND n.uid < 2000
- Préfixe. WHERE n.name STARTS WITH 'Tom'
- Suffixe WHERE n.name ENDS WITH 'Hanks'
- Sous-chaîne WHERE n.name CONTAINS 'a'

Ne marche pas avec :

- Combinaison avec OR de ces prédicats
- Existence WHERE EXISTS(n.name)

existence : pas d'information de type

OR : propriétés de type des prédicats peuvent différer

Prochain et dernier cours

- Procédures
- Applications
- Bibliothèques
- La graph data science library (gestion de la mémoire incluse)