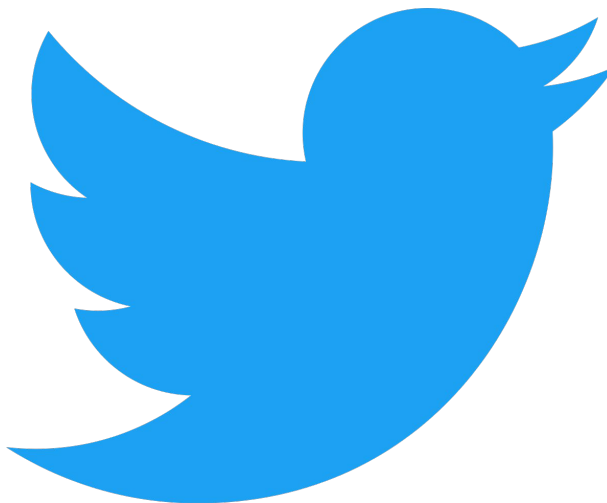
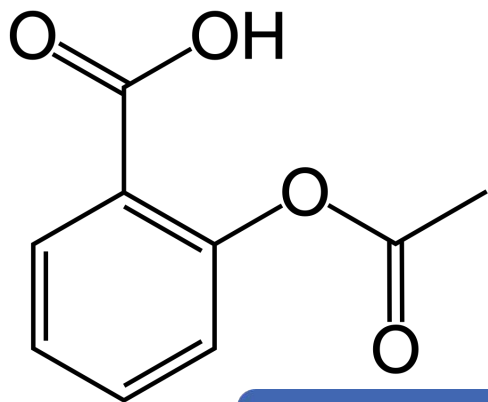


Neo4j

Base de données orientée graphe

1. Introduction

1. Introduction



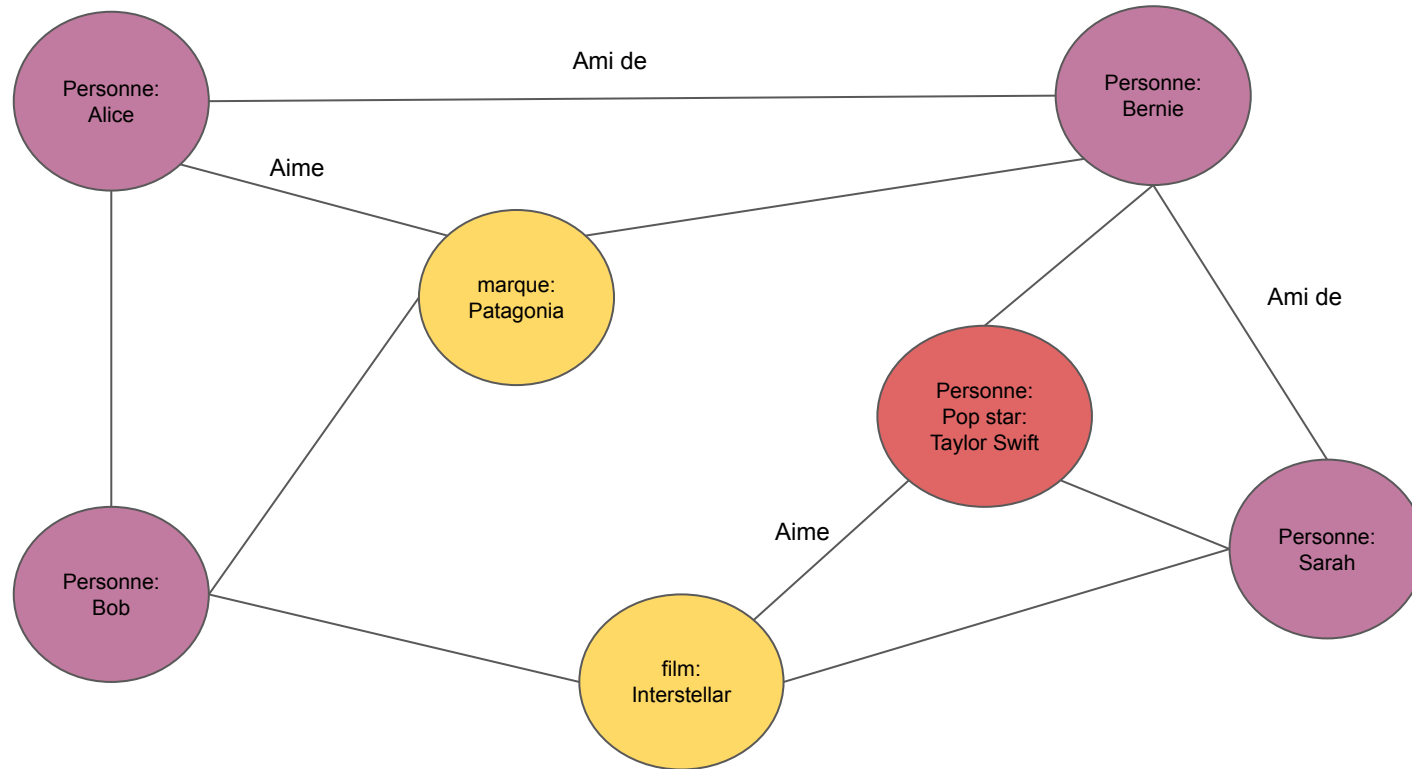
1. Introduction

La théorie des graphes est un champ des mathématiques. Les graphes servent à représenter de manière abstraite des réseaux et des objets.

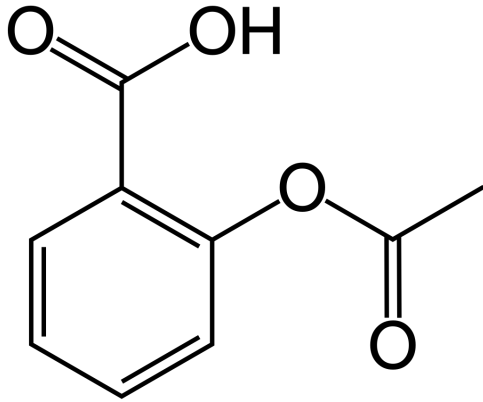
On peut représenter énormément de phénomènes sous forme de graphes:

- Réseaux sociaux et électriques, internet
- Systèmes solaires
- Molécules chimiques
- Colonies de fourmis
- Chaînes de Markov

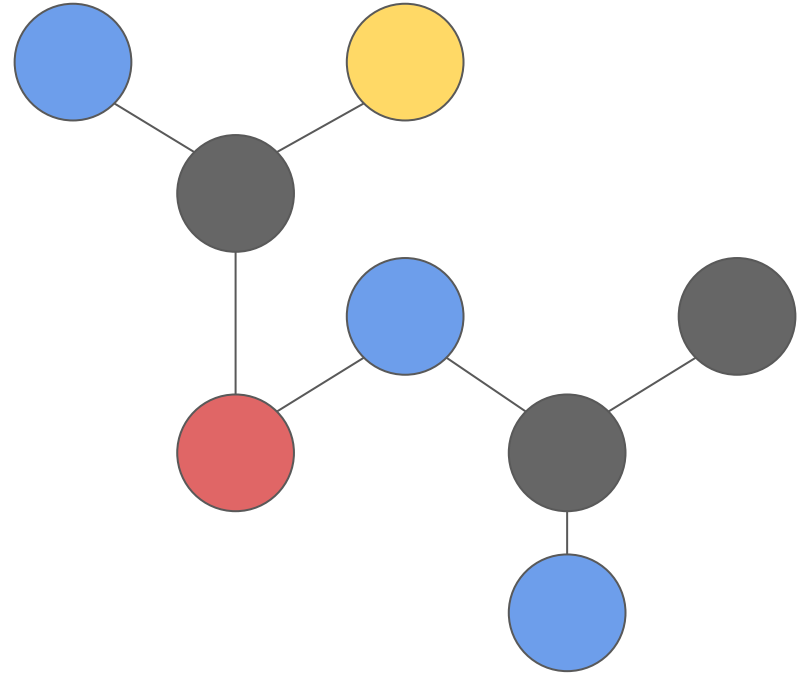
1. Introduction



1. Introduction



source: wikipedia



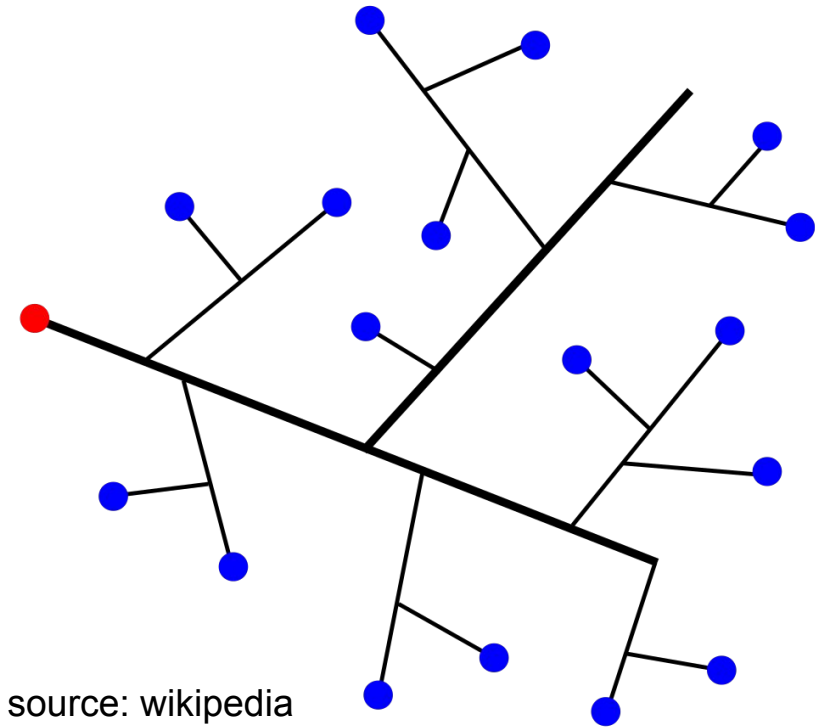
1. Introduction

Modélisation d'un réseau électrique

Le point rouge pourrait représenter une centrale

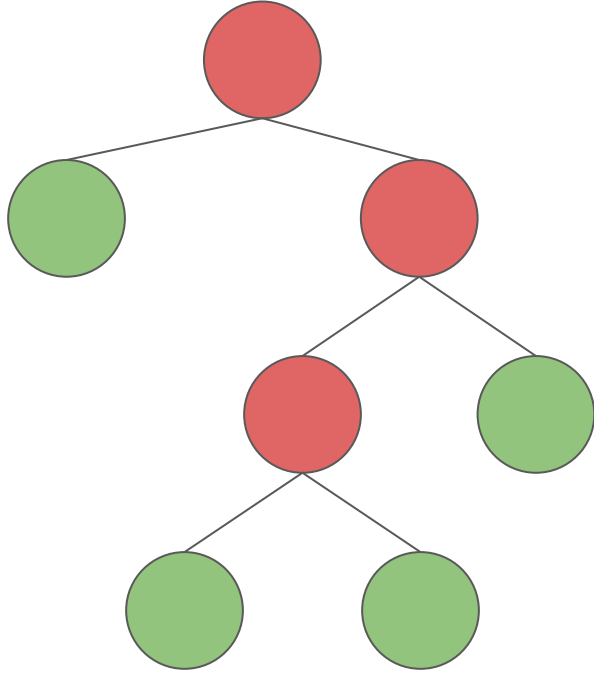
Les points bleus des transformateurs

les lignes noires, les lignes à haute tension



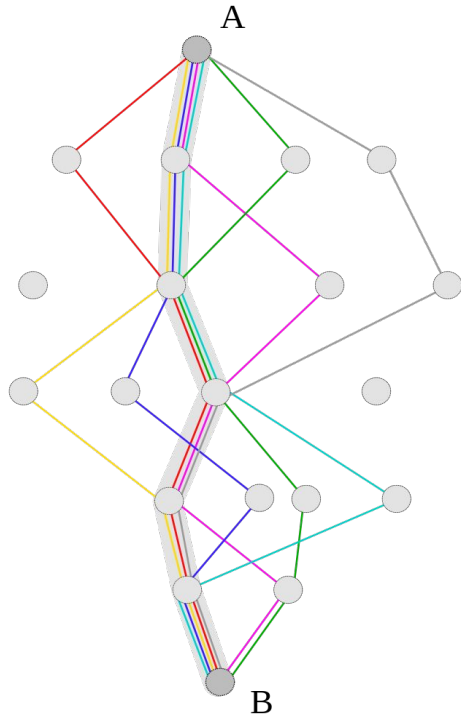
source: wikipedia

1. Introduction



Les arbres de décision sont eux aussi de parfait exemples

1. Introduction



L'Algorithme de colonies de fourmis est un algorithme d'optimisation qui permet de trouver le plus court chemin entre deux points.

source: wikipedia

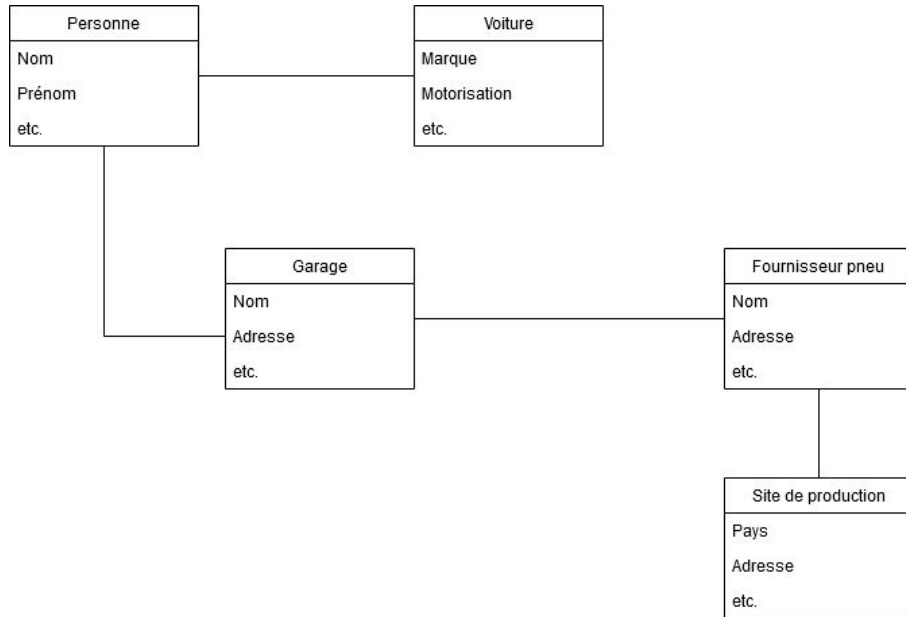
1. Introduction



Quels est rapport avec les bases de données ?

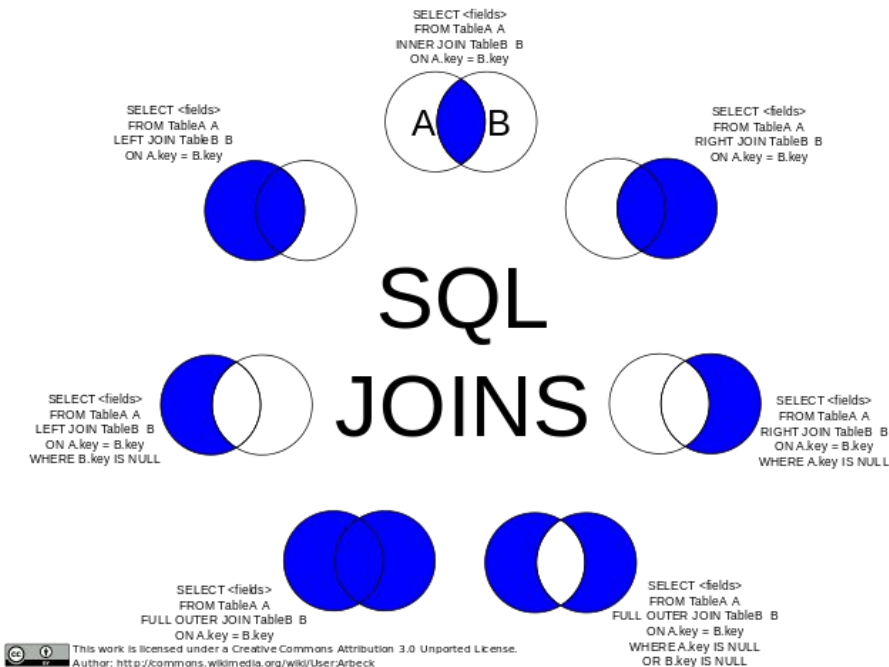
Pour bien comprendre les bases de données orientées graphe et leur intérêt, il faut d'abord comprendre les limites des autres bases de données (SQL ou NoSQL).

1. Introduction



Dans ce cas fictif, peut-on savoir d'où proviennent les pneus d'une voiture récemment achetée par une personne à un garage?

1. Introduction




Oui c'est tout à fait faisable mais cela a un coût.
Surtout avec un volume de données imposant.

Le système de jointure repose sur des **pointeurs logiques** (clefs primaires/étrangères).

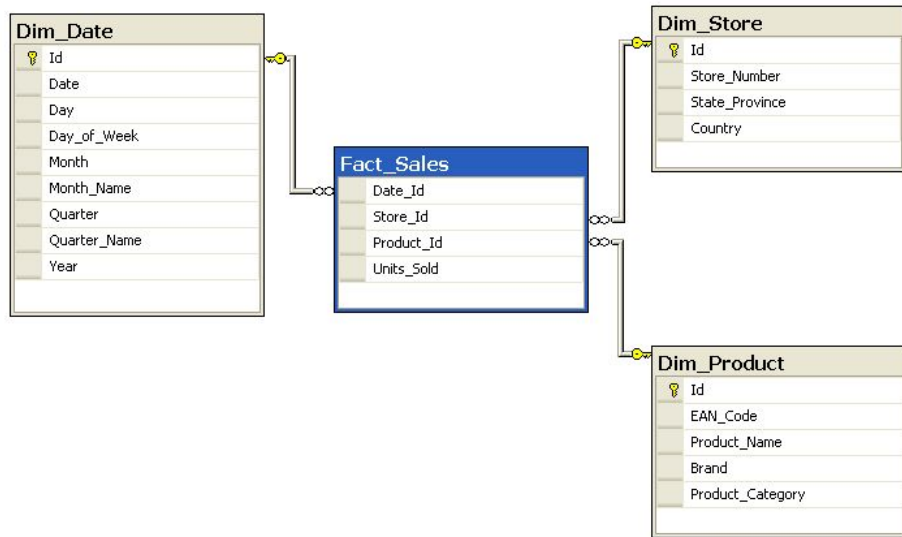
=> Cela demande du temps, beaucoup de temps

=> Les requêtes SQL peuvent potentiellement devenir longues et complexes

 This work is licensed under a Creative Commons Attribution 3.0 Unported License.
Author: <http://commons.wikimedia.org/wiki/User:Arbeck>

Source: Par Arbeck — Travail personnel, CC BY 3.0,
<https://commons.wikimedia.org/w/index.php?curid=25555796>

1. Introduction



On peut en partie régler le problème en **dénormalisant** notre base de données relationnelle au profit d'un datawarehouse par exemple avec un schéma plus simple.

Source: By SqlPac at English Wikipedia, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=76901169>

1. Introduction

Prenons un autre exemple, un réseau social.

Certains utilisateurs indiquent un numéro de téléphone, un diplôme, un parrain, une marraine, etc.

=> Ceci implique dans une base de données relationnelles d'avoir une multitude de **valeur nulle (sparsity)** .

1. Introduction

```
{
  id: 123
  nom: 'Hazard'
  prenom: 'Alice'
  telephone: 0258 54 52 56
  diplome: {
    université: 'UCL'
    cycle: 'Master'
    domaine: 'Informatique'
    année: 2015
  }
}

{
  id: 124
  nom: 'Dupont'
  prenom: 'Bernard'
  hobby : 'Foot'
}
```

Des bases de données NoSQL comme MongoDB permettent de résoudre ce problème.

Si une personne indique son diplôme, on l'enregistre sinon on n'enregistre rien.

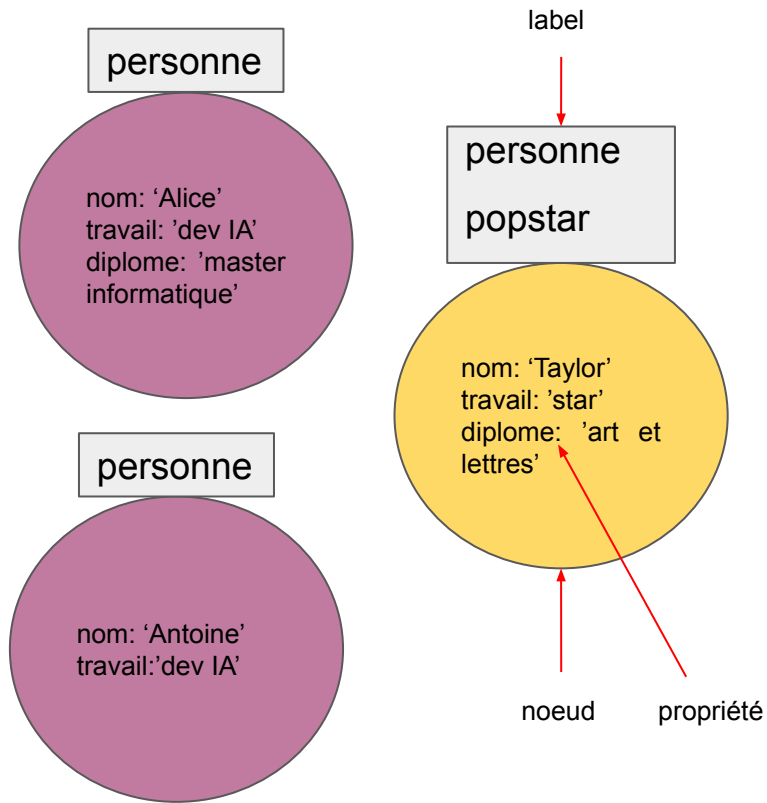
1. Introduction

Par contre, on souhaite dans le cadre de réseaux sociaux pouvoir connaître les différentes relations qui existent entre nos différentes entités.

E.g. Bob est ami avec Alice.

MongoDB ne permet pas de capter ces relations ou alors de manière peu efficace

1. Introduction

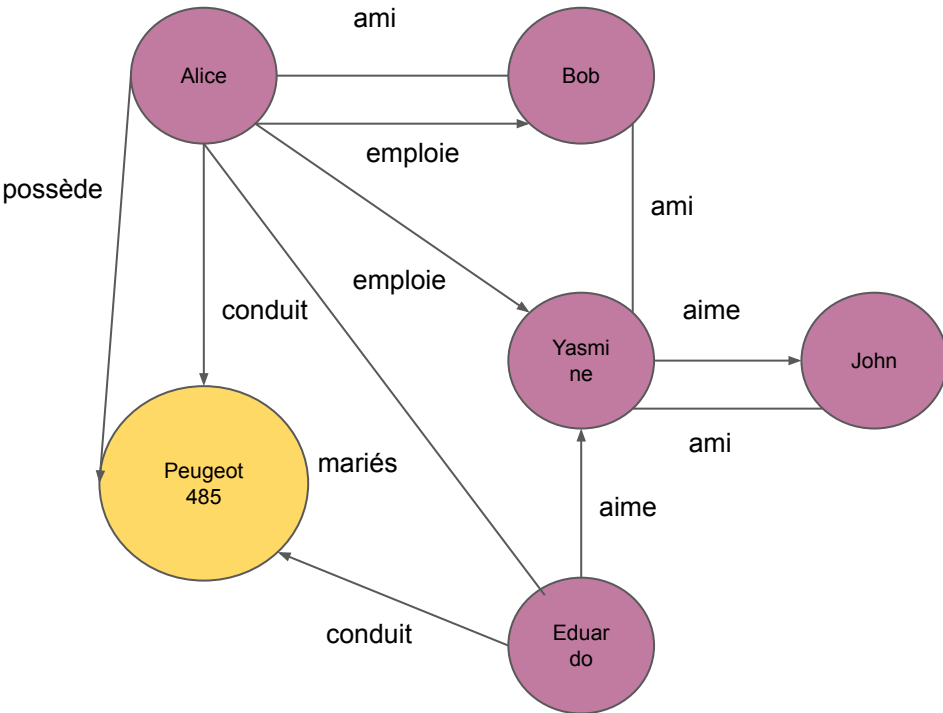


Les bases de données orientées graphe permettent de résoudre tous les problèmes rencontrés.

En premier lieu celui de la **'sparsity'**.

Chaque noeud contient des propriétés et un ou plusieurs labels.

1. Introduction

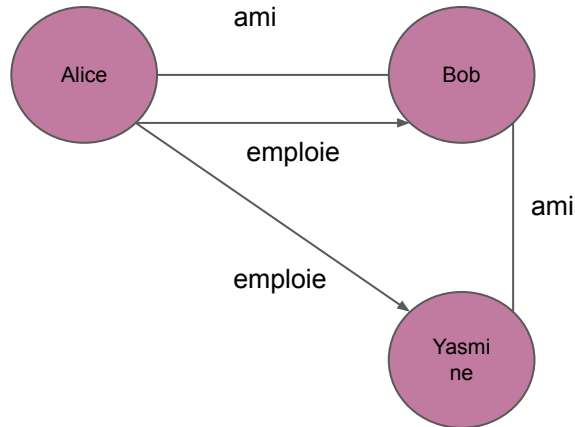


Ensuite, on peut garder une trace des relations. Relations qui peuvent ne pas être réciproques.

Tout ceci en utilisant non pas des pointeurs logiques mais des **pointeurs physiques**.

P.s.: Relation est synonyme d'arc ou 'edge' en Anglais

1. Introduction

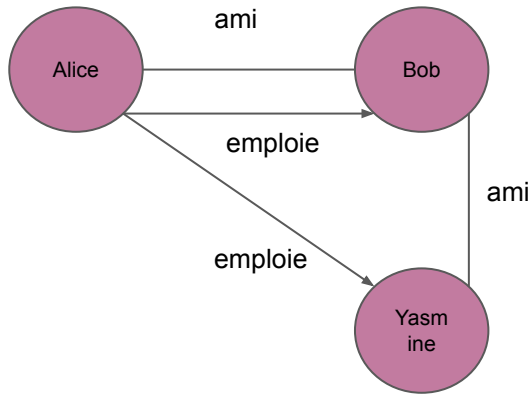


Comment peut-on représenter des graphes pour les ordinateurs ?

Deux possibilités:

- Adjacency matrix
- Adjacency list

1. Introduction



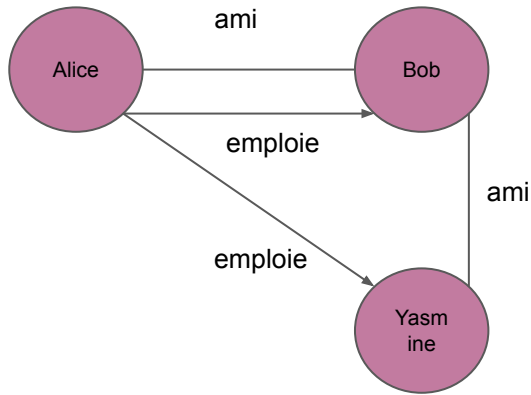
	Alice	Bob	Yasmine
Alice	0	1	1
Bob	1	0	1
Yasmine	0	1	0

Existe-t-il une relation entre des lignes (i) et des colonnes (j)?

Ici nous avons utilisé un booléen mais nous aurions pu mettre un poids à la place. (Cfr. partie DDL).

=> Se lit très facilement mais prend **beaucoup de place**.

1. Introduction



Alice -> [Bob, Yasmine]

Bob -> [Alice, Yasmine]

Yasmine -> [Bob]

Il s'agit ici d'associer à chaque noeud une liste des noeuds vers lesquels ils ont une relation.

A nouveau on peut associer un poids correspondant à ces relations.

2. Cypher

2. Cypher

Cypher est le langage utilisé par Neo4j pour la création, l'interrogation et la mise à jour des bases de données.

Il existe cependant d'autres langages:

- SPARQL
- GraphQL
- GremlinQL

2. Cypher

Cypher se veut très proche de SQL et son apprentissage est d'ailleurs simplissime pour ceux le maîtrisant déjà.

De plus, les requêtes Cypher sont généralement nettement plus intuitives et plus courtes que celles de SQL.

C'est pourquoi la suite de la présentation présuppose que vous maîtrisiez déjà SQL.

2. Cypher

DDL

2. Cypher

La clause 'CREATE' permet de créer des noeuds et d'établir des relations.

```
CREATE (:Person {first_name:'Donald', last_name:'Biden', job:'US President'})
```

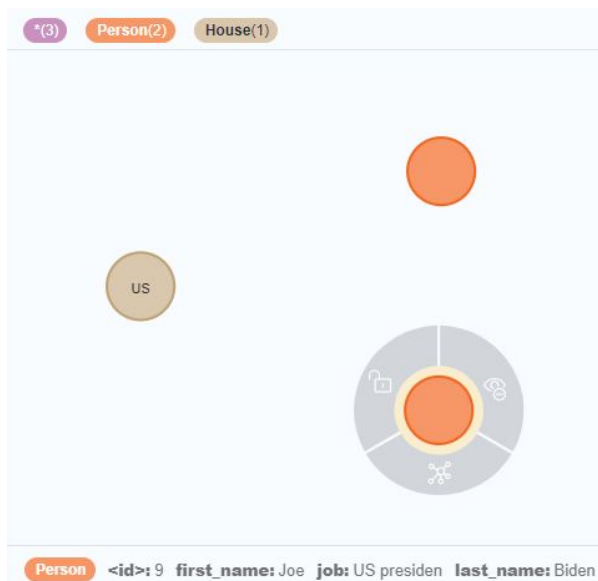
```
{  
  "identity": 9,  
  "labels": [  
    "Person"  
  ],  
  "properties": {  
    "last_name": "Biden",  
    "job": "US President",  
    "first_name": "Donald"  
  }  
}
```

Ici nous avons créé un noeud avec le label 'Person' et 3 propriétés qui lui sont associées.

2. Cypher

Create

```
(:Person {first_name:'Joe', last_name:'Biden', job:'US president'}),  
(:Person {first_name:'Donald', last_name:'Trump', job:'US president', hobby:'golf'}),  
(:House {country:'US', latitude:38.897321, longitude:-77.036571})
```



Nous avons créé 3 noeuds ainsi que deux labels.

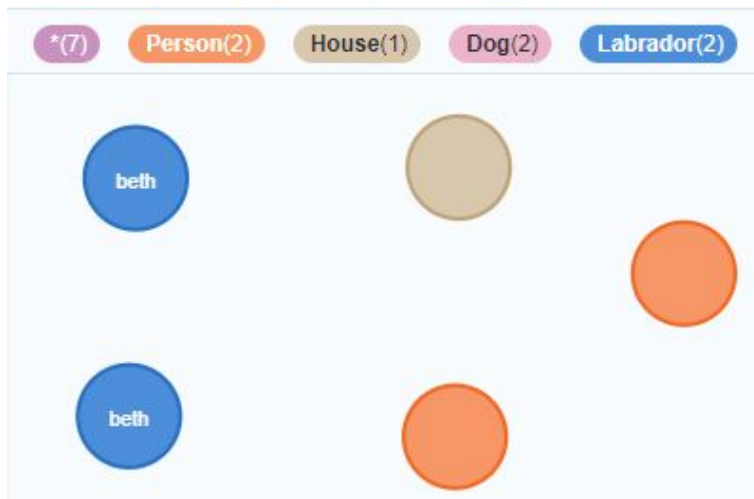
Il n'y a aucune obligation à avoir des noeuds d'un même label possédés les mêmes propriétés.

Il est également possible de créer des noeuds sans propriété ou sans label.

2. Cypher

Create

```
(:Person {first_name:'Joe', last_name:'Biden', job:'US presiden'}),  
(:Person {first_name:'Donald', last_name:'Trump', job:'US president', hobby:'golf'}),  
(:House {Country:'US', latitude:38.897321, longitude:-77.036571}),  
(:Dog :Labrador {name:'beth', birth_date:date('1999-02-12')}),  
(:Dog :Labrador {name:'beth', birth_date:date({year:1999, month:0o2, day:12})})
```



Voici un graphe avec plus de noeuds. On peut constater:

- la gestion des dates
- la gestion des doublons
- de multiples labels

2. Cypher

Jusqu'ici nous n'avons pas encore établie de relation entre nos noeuds. Ceci s'effectue toujours avec la clause 'CREATE'

CREATE

```
(:Person {name:'Alice'}) -[:Sister]→ (:Person {name:'Romain'})
```



Il n'est pas possible de créer des relations bidirectionnelles en une seule ligne.

2. Cypher

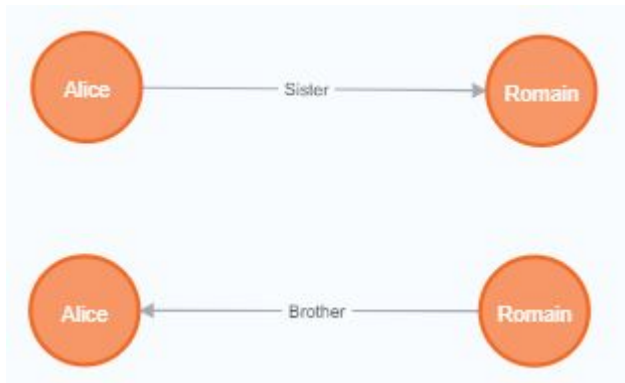
La requête suivante est-elle correcte ?

CREATE

```
(:Person {name:'Alice'}) -[:Sister]→ (:Person {name:'Romain'}),  
(:Person {name:'Alice'}) ←[:Brother]- (:Person {name:'Romain'})
```

2. Cypher

Non, nous créons surtout des doublons mais toujours pas de relation bidirectionnelle.

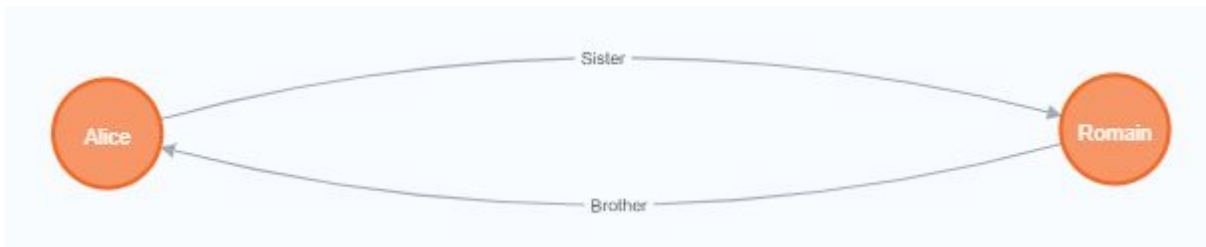


2. Cypher

Il faut stocker nos noeuds dans des variables.

CREATE

```
(a:Person {name:'Alice'}), //stockage du noeud alicé dans la variable a  
(r:Person {name:'Romain'}),  
(a) -[:Sister]→ (r),  
(r) -[:Brother]→ (a)
```



Est-ce utile dans ce cas précis ?

2. Cypher

Ceci dit, attention à ne pas créer des relations dont on peut déduire la réciproque.

Si Alice est la soeur de Romain, il n'y a pas besoin de spécifier que Romain est le frère d'Alice.

2. Cypher

Comment gérer les doublons, s'assurer que certains doublons ne puissent pas être entrés en base de données ?

Avec des contraintes.

```
CREATE CONSTRAINT name_unique IF NOT EXISTS  
ON (l:Person)  
ASSERT l.last_name IS UNIQUE
```

Ou avec la clause 'Merge', *cfr.* "importe des données".

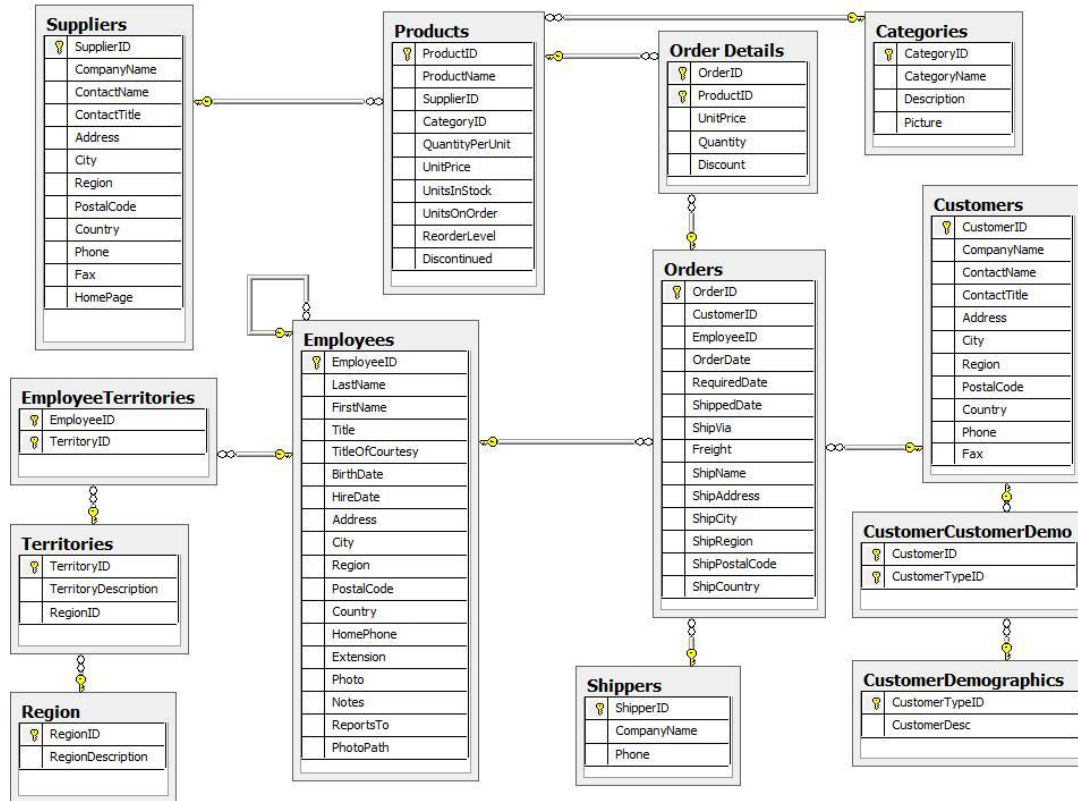
2. Cypher

```
1 CREATE
2 (:Person {first_name:'Joe', last_name:'Biden'}),
3 [(:Person {first_name:'Hunter', last_name:'Biden'})]
```

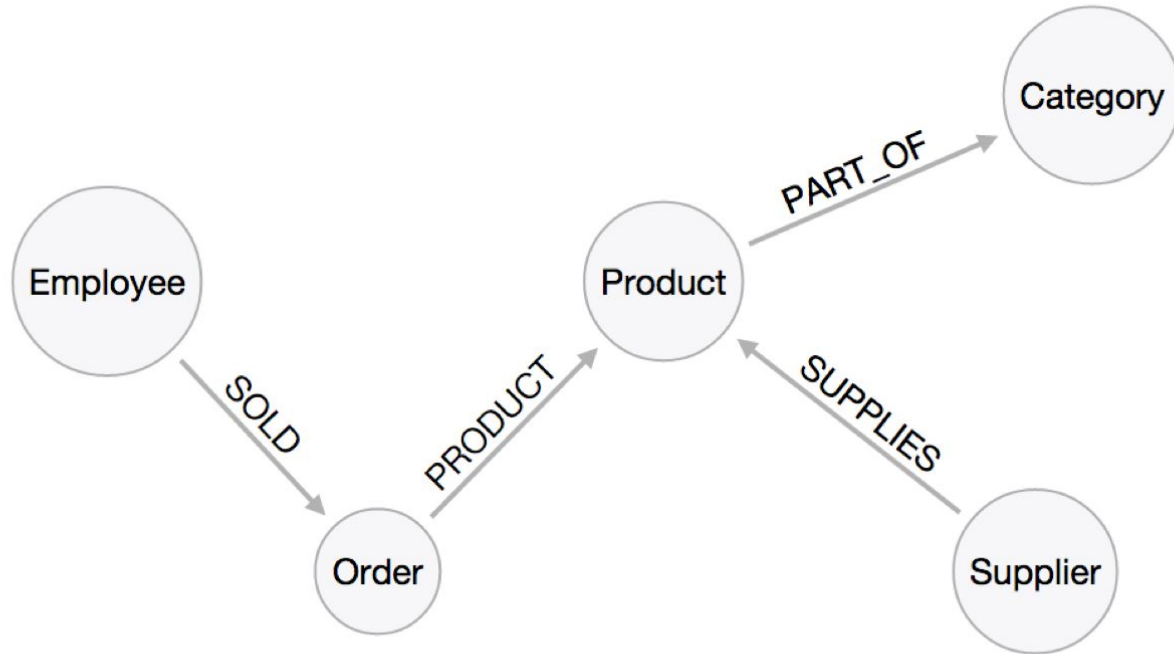
ERROR Neo.ClientError.Schema.ConstraintValidationFailed

Node(17) already exists with label `Person` and property `last_name` = 'Biden'

2. Cypher



2. Cypher



2. Cypher

DRL

2. Cypher

La clause 'MATCH' permet de trouver des patterns au sein des graphes. C'est le 'SELECT' de Cypher.

Pour la suite de la présentation nous nous baserons sur le dataset suivant:

<https://github.com/mathbeveridge/gameofthrones>

Il s'agit d'un dataset qui répertorie toutes les interactions de tous les personnages de Game Of Thrones.

2. Cypher

‘MATCH’ fonctionne avec une ou plusieurs variables ainsi qu’avec une clause ‘RETURN’ qui renvoie les résultats de la recherche.

```
MATCH(n)  
RETURN n
```

Ceci renvoie l’entièreté du graphe.

```
MATCH(p:Person)  
RETURN p.name
```

Ceci ne renvoie que la propriété ‘name’ des noeuds avec le label ‘Person’.

2. Cypher

```
MATCH(n:Person {name:'ned'})--(s)  
RETURN n,s
```

Tous les noeuds qui ont une relation avec 'ned'.
Qu'importe le label ou le type de relation.

On peut également spécifier le nom de l'interaction et le type de noeud recherché. Dans la requête suivante, on cherche seulement les noeuds 'Person' ayant eu une relation avec 'ned' durant la première saison.

```
MATCH(n:Person {name:'ned'})-[r:Interacts_1]-(p:Person)  
RETURN n,p  
LIMIT 10
```

2. Cypher

Tout comme pour SQL, il est possible de filtrer nos résultats avec la clause 'WHERE'.

```
MATCH (p:Person)
WHERE p.name IN ['arya', 'ned', 'robb']
RETURN p
```

```
MATCH (p:Person)-[r]-()
WHERE p.name IN ['arya', 'ned', 'robb'] OR r.weight > 150
RETURN p
```

```
MATCH (p:Person)-[r]-()
WHERE p.name CONTAINS 'rob'
RETURN p
```

2. Cypher

```
MATCH (p:Person)-[r]-()  
WHERE size(p.name) > 15  
RETURN p
```

```
MATCH  
  (n:Person {name:'ned'}),  
  (a:Person {name:'arya'}),  
  (other:Person)  
WHERE (other)--(a) AND NOT (other)--(n)  
RETURN n,a,other  
LIMIT 10
```

La fonction 'size' permet de calculer la taille d'une liste.

On peut également utiliser 'WHERE' pour trouver des patterns basés sur des relations.

Cette requête permet de trouver toutes les personnes en lien avec 'arya' mais pas avec 'ned'.

2. Cypher

La clause 'WITH' permet de stocker temporairement un résultat dans une variable.

Dans ce cas on place le nom des personnes connectée à 'ned' dans la variable 'person_ned_linked'. 'collect' permet de créer une liste.

Alors on peut retrouver les personnes connectées à 'arya' mais qui ne sont pas dans notre liste.

```
MATCH (n:Person {name:'ned'}) -- (p:Person)
WITH collect(p.name) as person_ned_linked

MATCH (a:Person {name:'arya'}) -- (p:Person)
WHERE NOT p.name IN person_ned_linked or p.name='ned'

RETURN p,a
```

2. Cypher

Tout comme dans SQL nous retrouvons toute une série de fonctions mathématiques et d'agrégation.

COUNT, AVG, SUM, COS, SIN, etc.

Vous pouvez les retrouver ici:

<https://neo4j.com/docs/cypher-manual/current/functions/>

=> procédé identique à SQL.

2. Cypher

DML

2. Cypher

Cette partie traite:

- la mise à jour d'un noeud préexistant
- l'ajout de nouvelles relations
- la suppression de certains noeuds ou relations

2. Cypher

La clause 'DELETE' sert à supprimer soit un noeud soit une relation bien spécifique.

Il faut donc l'utiliser avec un MATCH pour trouver des pattern à modifier.

Cette requête ne fonctionne pas parce que pour pouvoir supprimer un noeud, il faut d'abord supprimer les relations associées à ce noeud.

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

2. Cypher

```
MATCH (p:Person {name : 'ned'})-[r]-()  
DELETE r,p
```

La requête équivalente au 'TRUNCATE' dans SQL est la suivante:

```
MATCH (n)  
DETACH DELETE n
```

Pour retirer une propriété d'un noeud ou d'une relation ou pour retirer le label d'un noeud, il faut utiliser la clause 'REMOVE' et non pas 'DELETE'.

2. Cypher

La clause 'SET' permet d'ajouter une ou plusieurs propriétés à des noeuds ou relations à condition de respecter les contraintes.

```
MATCH (stark:Person)
WHERE stark.name IN ['ned', 'catelyn', 'robb', 'arya', 'sansa', 'bran', 'jon']
SET stark.last_name = 'stark'
```

2. Cypher

Importer des données

2. Cypher

Intéressons nous maintenant à l'import de données avec Neo4J.

'LOAD CSV' permet de charger des fichiers CSV.

Les données de Game Of Thrones proviennent d'ici:

<https://github.com/mathbeveridge/gameofthrones>

2. Cypher

got-s1-edges.csv	fixing EGAN typo and renaming got-s5-node.csv to got-s5-nodes.csv
got-s1-nodes.csv	consistent names between seasons
got-s2-edges.csv	consistent names between seasons
got-s2-nodes.csv	consistent names between seasons
got-s3-edges.csv	consistent names between seasons
got-s3-nodes.csv	consistent names between seasons
got-s4-edges.csv	consistent names between seasons
got-s4-nodes.csv	consistent names between seasons
got-s5-edges.csv	consistent names between seasons
got-s5-nodes.csv	fixing EGAN typo and renaming got-s5-node.csv to got-s5-nodes.csv
got-s6-edges.csv	consistent names between seasons
got-s6-nodes.csv	consistent names between seasons
got-s7-edges.csv	consistent names between seasons
got-s7-nodes.csv	consistent names between seasons
got-s8-edges.csv	consistent names between seasons
got-s8-nodes.csv	consistent names between seasons

Chaque saison a deux fichiers CSV, les noeuds et les relations.

En réalité seule les relations nous intéressent.

2. Cypher

Source	Target	Weight	Season
NED	ROBERT	192	1
DAENERYS	JORAH	154	1
JON	SAM	121	1
LITTLEFINGER	NED	107	1
NED	VARYS	96	1
DAENERYS	DROGO	91	1
ARYA	NED	90	1
CATELYN	ROBB	90	1
BRONN	TYRION	86	1
CERSEI	NED	86	1
CERSEI	ROBERT	80	1
LITTLEFINGER	VARYS	73	1
SHAE	TYRION	73	1
CATELYN	NED	69	1

Chaque saison a deux fichiers CSV, les noeuds et les relations.

En réalité seules les relations nous intéressent.

2. Cypher

```
LOAD CSV WITH HEADERS FROM  
'https://raw.githubusercontent.com/mathbeveridge/gameofthrones/master/data/got-s1-edges.csv' AS line  
MERGE (source:Person {name:toLower(line.Source)})  
MERGE (target:Person {name:toLower(line.Target)})  
MERGE (target)-[:INTERACT {weight:line.Weight}]- (source)
```

On peut directement passer des liens. Voir la doc si ce sont des fichiers en local.

La clause 'MERGE' sert à créer des relations ou noeuds que s'ils n'existent pas.

2. Cypher

```
UNWIND range(1,8) AS season
LOAD CSV WITH HEADERS FROM
'https://raw.githubusercontent.com/mathbeveridge/gameofthrones/master/data/got-s'+season+'-edges.csv' AS line
MERGE (source:Person {name:toLower(line.Source)})
MERGE (target:Person {name:toLower(line.Target)})
WITH season, source, target, line
CALL apoc.merge.relationship(source,'INTERACT'+season, {}, {weight:toFloat(line.Weight)}, target) YIELD rel
RETURN DISTINCT 'done'
```

Voici la requête pour aller chercher toutes les saisons.

La clause 'UNWIND' permet de traiter notre liste renvoyée par 'range' de manière à reconstruire l'URL.

La clause 'WITH' est ici nécessaire pour l'utilisation d'une fonction de la librairie apoc.

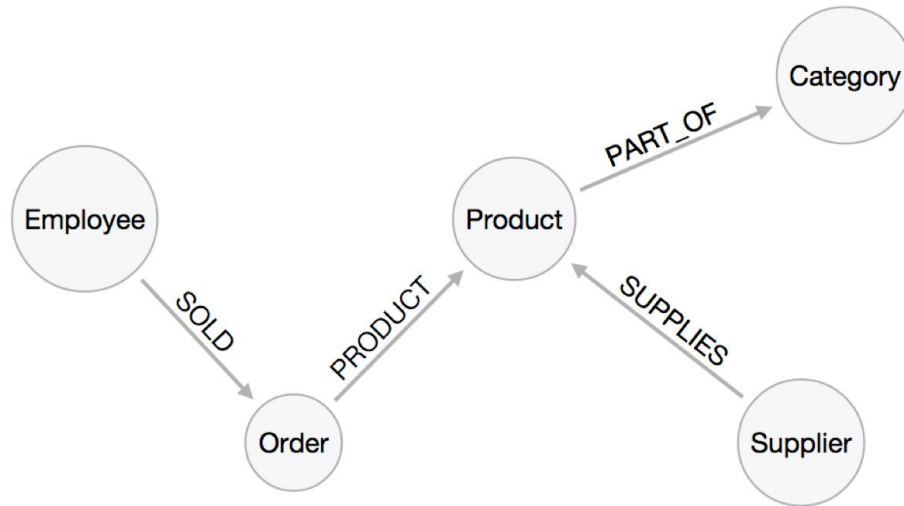
2. Cypher

apoc est une librairie qui contient une série de fonctions et de procédures écrites en Java qu'il est possible d'utiliser directement dans neo4j.

Ici nous en avons besoin pour construire de manière dynamique la relation reliant les personnes. Un 'INTERACT' par saison.

2. Cypher

Comment faire quand on a plusieurs csv/tables et que l'on souhaite relier ?



2. Cypher

D'abord créer les produits:

```
LOAD CSV WITH HEADERS FROM 'file:/product.csv' AS line FIELDTERMINATOR ','  
MERGE (p:Product {id:toInteger(line.ProductId), category_id:toInteger(line.CategoryId),  
|      |      name:line.name, unit_price:toFloat(line.UnitPrice)})
```

Ensuite les catégories que l'on relie aux produits:

```
LOAD CSV WITH HEADERS FROM 'file:/category.csv' AS line FIELDTERMINATOR ','  
MATCH (p:Product {category_id:toInteger(line.CategoryId)})  
MERGE (c:Category {id:line.CategoryId, name:line.CategoryName})  
MERGE (p)-[:BELONGS_TO]-(c)
```

3. Neo4j 4 data science