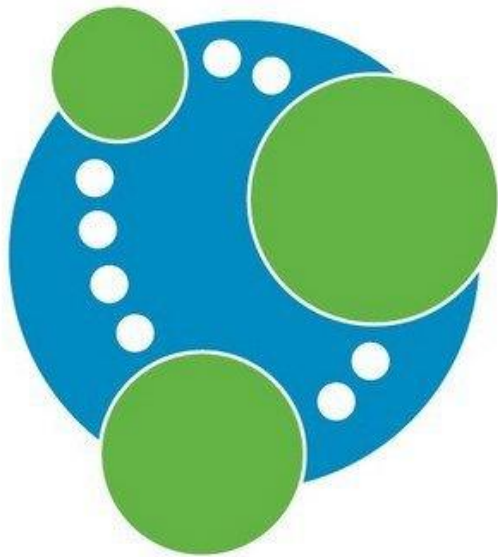


Graph Database: Neo4j



neo4j

Réalisé par:

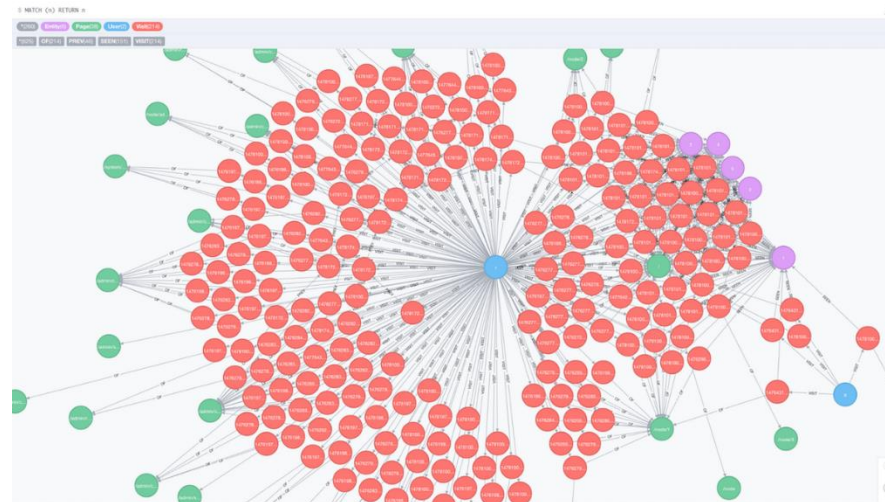
Ibtissam ESSADIK, Ayoub EL KATIBI & Abdelkarim DAKOUAN

Vue d'ensemble

- Très simplement, une base de données de graphes (graph database) est une base de données conçue pour traiter les relations entre les données comme tout aussi importantes que les données elles-mêmes.
- Il est destiné à contenir des données sans les restreindre à un modèle prédéfini. Au lieu de cela, les données sont stockées comme nous l'avons tout d'abord dessiné montrant comment chaque entité individuelle se connecte ou est liée aux autres.

Pourquoi les bases de données graphe?

- Nous vivons dans un monde connecté! Il n'y a pas d'informations isolées, mais des domaines riches et connectés tout autour de nous.
- Seule une base de données qui englobe des relations de manière native peut stocker, traiter et interroger des connexions efficacement.
- Tandis que d'autres bases de données calculent les relations au moment de l'interrogation via des opérations JOIN coûteuses, une base de données graphe stocke les connexions aux côtés des données du modèle.



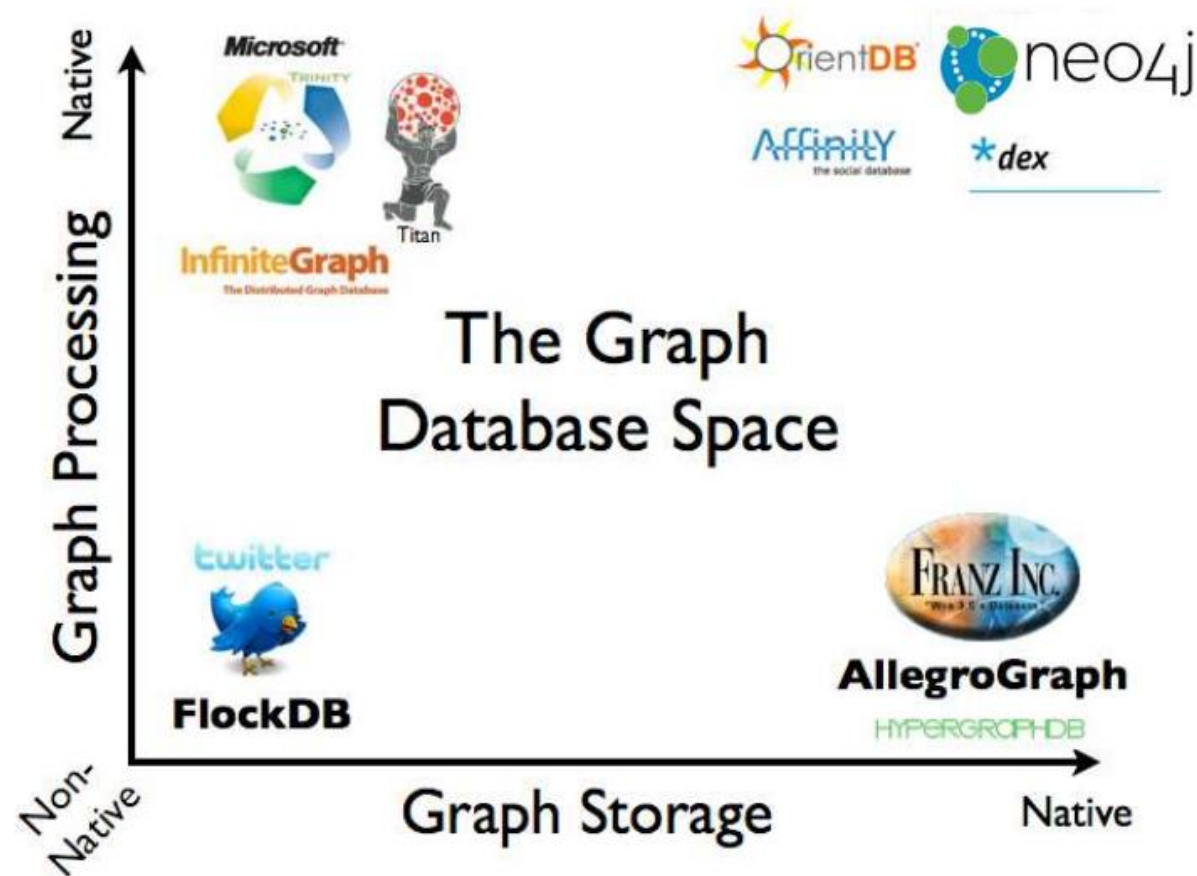
- L'accès aux nœuds et aux relations dans une base de données de graphes natif est une opération efficace à temps constant qui vous permet de parcourir rapidement des millions de connexions par seconde par cœur.
- Indépendamment de la taille totale de votre jeu de données(dataset), les bases de données graphiques excellent pour la gestion de données hautement connectées et de requêtes complexes.
- Avec seulement un modèle et un ensemble de points de départ, les bases de données graphiques explorent les données voisines autour de ces points de départ collectant et agrégeant des informations provenant de millions de nœuds et de relations et laissant intactes les données situées en dehors du périmètre de recherche.



Qu'est-ce que Neo4j ?

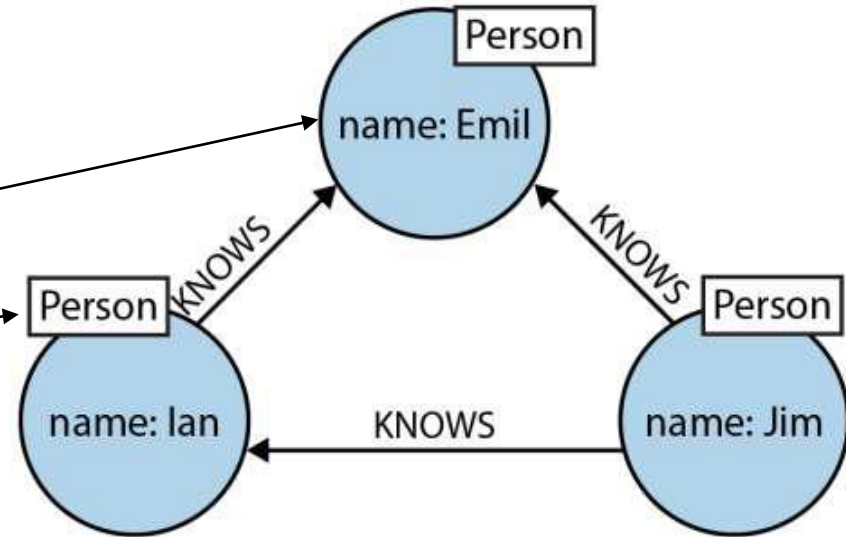
- Neo4j est développé par Neo Technology, Inc. Il est utilisé par des milliers d'organisations, dont plus de 50 entreprises du Global 2000, dans des applications de production critiques.
- Neo4j a à la fois une édition Community Edition et une édition Enterprise.
- Neo4j stocke et présente des données sous forme de graphe. les données sont représentées par des nœuds et des relations entre ces nœuds.
- Le code source écrit en Java et en Scala.
- Neo4j fournit toutes les caractéristiques de la base de données, notamment la conformité des transactions ACID, la prise en charge des clusters et le basculement au moment de l'exécution.

Stockage et traitement graphe



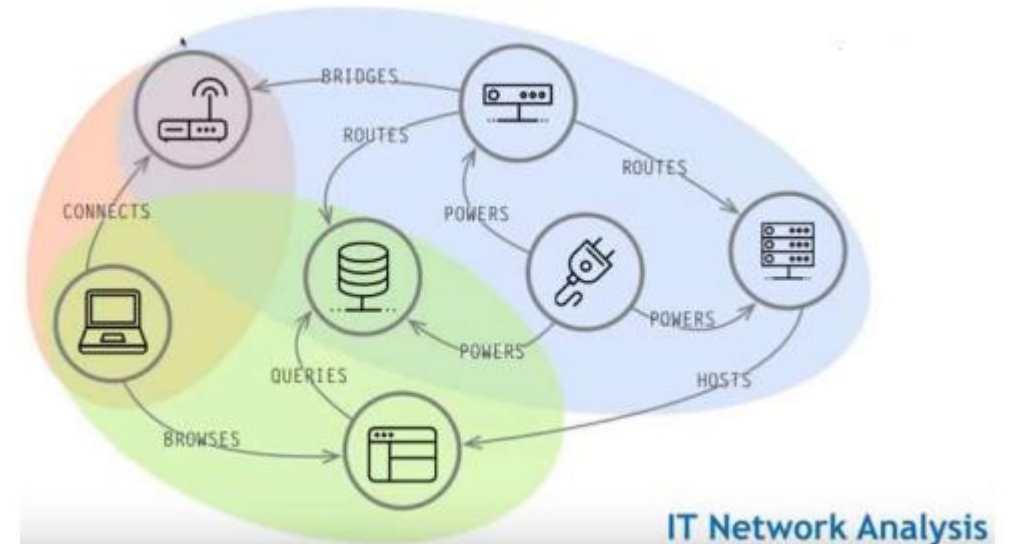
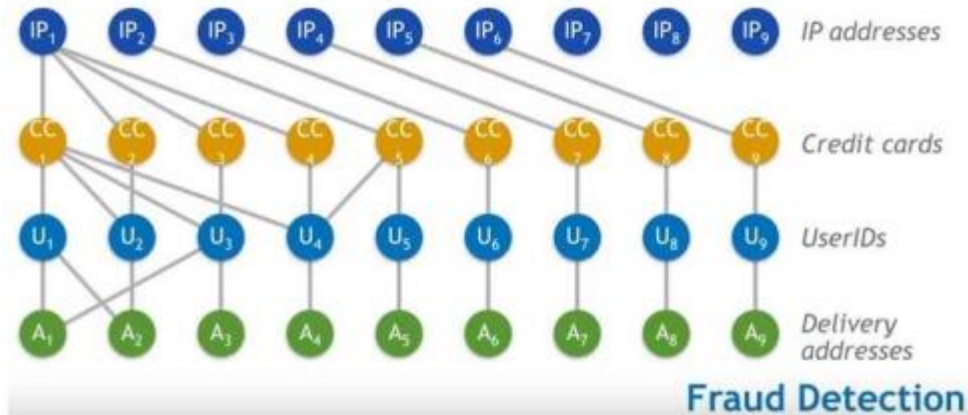
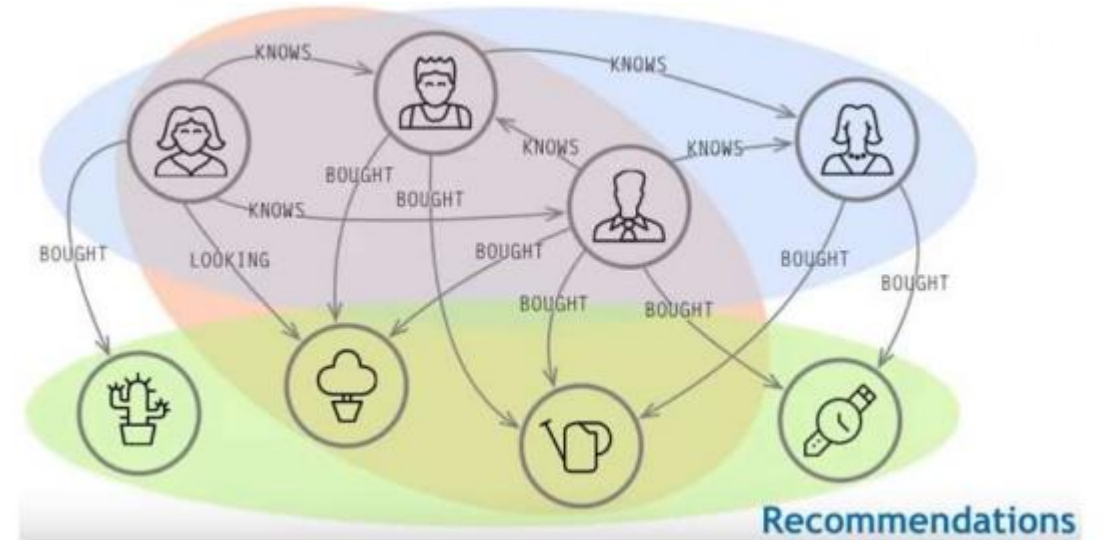
Modèle de graphe par propriétés

- Neo4J utilise un graphe de type propriété qui est un type de graphe particulier dans lequel à la fois les nœuds et les relations peuvent avoir des propriétés au format **clé-valeur**, ce qui offre donc un modèle de données entièrement dynamique.
- Il est Composés par:
 - des nœuds
 - des relations
 - des **propriétés**
 - des **libellés**



Cas d'utilisations

- Réseaux sociaux.
- Recommandations de produits en temps réel
- Diagrammes de réseau
- Détection de fraude
- Gestion des accès
- Recherche basée sur les graphiques des actifs numériques
- Analyse des réseaux IT



Les clients de Neo4j



V O L V O



- Certaines des caractéristiques suivantes rendent Neo4j très populaire parmi les développeurs, les architectes et les administrateurs de base de données:

Cypher	Constant time traversals	Flexible property graph schema	Drivers
- Cypher, langage de requête déclaratif similaire à SQL, mais optimisé pour les graphes.	Traversées temporelles constantes dans de grands graphiques pour la profondeur et la largeur grâce à une représentation efficace des nœuds et des relations. Permet la mise à l'échelle de milliards de nœuds sur un matériel modéré.	Schéma de propriété graphique flexible pouvant s'adapter au fil du temps, permettant de matérialiser et d'ajouter de nouvelles relations ultérieurement afin de raccourcir et d'accélérer les données du domaine lorsque les besoins de l'entreprise changent.	Pilotes(drivers) pour les langages de programmation courants, notamment Java, JavaScript, .NET, Python et bien d'autres.

Neo4j vs bases de données relationnelles traditionnelles (SGBDR)

	 Base de données relationnelle	 Neo4j, base de données de graphes natifs
Stockage de données	<ul style="list-style-type: none">- Le stockage dans des tables fixes et prédéfinies avec des lignes et des colonnes avec des données connectées est souvent disjoint entre tables, ce qui nuit à l'efficacité des requêtes.	<ul style="list-style-type: none">- La structure de stockage graphique avec une contiguïté sans index permet des transactions et un traitement plus rapides des relations de données
La modélisation des données	<ul style="list-style-type: none">- Le modèle de base de données doit être développé avec les modélisateurs et converti d'un modèle logique en un modèle physique.- Etant donné que les types et les sources de données doivent être connus à l'avance, toute modification nécessite une mise en œuvre longue de plusieurs semaines.	<ul style="list-style-type: none">- Modèle de données flexible, compatible avec les tableaux blancs, sans décalage entre les modèles logique et physique.- Les types de données et les sources peuvent être ajoutés ou modifiés à tout moment, entraînant des temps de développement considérablement plus courts et une véritable itération agile.

Performance de la requête	<ul style="list-style-type: none"> - Les performances de traitement des données souffrent du nombre et de la profondeur des JOIN (ou des relations interrogées). 	<ul style="list-style-type: none"> - Le traitement graphique garantit une latence nulle et une performance en temps réel, quel que soit le nombre ou la profondeur des relations.
Support de transaction	<ul style="list-style-type: none"> - Prise en charge des transactions ACID requise par les applications d'entreprise pour des données cohérentes et fiables. 	<ul style="list-style-type: none"> - Conserve les transactions ACID pour des données totalement cohérentes et fiables 24 heures sur 24 - parfait pour les applications d'entreprise globales toujours actives.
Langage de requête	SQL : langage de requête dont la complexité augmente avec le nombre de jointures requises pour les requêtes de données connectées.	Cypher : langage de requête de graphe natif qui fournit le moyen le plus efficace et le plus expressif de décrire des requêtes de relation.
Efficacité du datacenter	La consolidation de serveurs est possible mais coûteuse pour une architecture évolutive. L'architecture scale-out est coûteuse en termes d'achat, de consommation d'énergie et de temps de gestion.	Les données et les relations sont stockées de manière native et les performances s'améliorent à mesure que la complexité et la taille grandissent. Cela conduit à la consolidation des serveurs et à une utilisation incroyablement efficace du matériel.

Neo4j vs les autres bases de données NOSQL

	Cassandra	MongoDB	Neo4j
Supported programming languages	C#,C++,Clojure,Erlang,Go Haskell, Java, JavaScript Perl, PHP, Python, Ruby Scala	Actionscript , C ,C#,C++ Clojure ,ColdFusion ,D Dart ,Delphi ,Erlang Go ,Groovy ,Haskell ,Java JavaScript,Lisp ,Lua MatLab ,Perl, PHP PowerShell, Prolog, Python ,R ,Ruby , Scala Smalltalk	.Net Clojure Elixir Go Groovy Haskell Java JavaScript Perl PHP Python Ruby Scala
Triggers	yes	No	Yes
Foreign keys	No	No	Yes
Transaction concepts	No	Multi-document ACID Transactions with snapshot isolation	ACID
MapReduce	yes	Yes	Yes
Replication methods	selectable replication factor	Master-slave replication	Causal Clustering using Raft protocol

Cypher

- Neo4j a son propre langage de requête appelé Cypher(Cypher utilise une syntaxe similaire à SQL).
- Désormais utilisé par d'autres bases de données telles que SAP HANA Graph et Redis graph via le projet openCypher.
- Cypher utilise ASCII-Art pour représenter des motifs.
- Les principales choses à retenir:
 - Les nœuds sont représentés par des parenthèses: (noeud)
 - Les relations sont représentées par des flèches: ->
 - Les informations sur une relation peuvent être insérées entre crochets:
[:Relationship]

Créer des nœuds

- Créer un seul nœud:

```
1 CREATE (p:Person {name:"Ibtissam",born:1996})  
2 RETURN p
```

- Vous pouvez créer plusieurs nœuds à la fois en séparant chaque nœud par une virgule:

```
$ create (n: Actor { name: ' Kristen Stewart' }),(m:Movie { title: 'snow white and the huntsman',  
distributed: 2012}) return m,n;
```

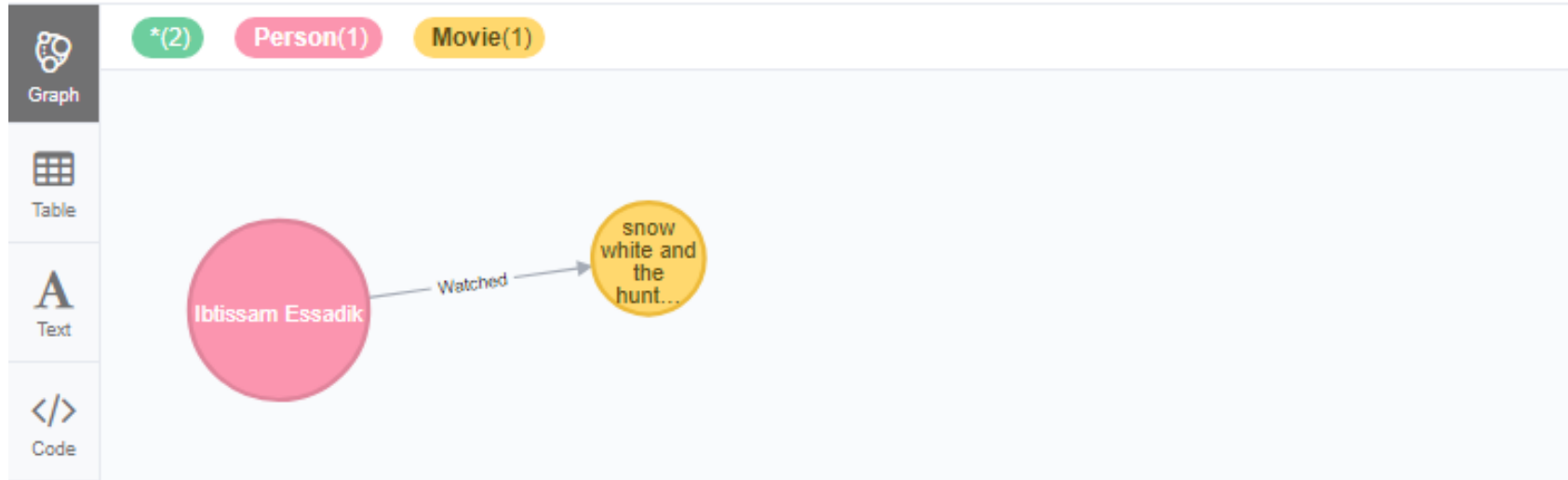
- Ou vous pouvez utiliser plusieurs instructions CREATE:

```
1 create (n: Actor { name: ' Kristen Stewart' })  
2 create(m:Movie { title: 'snow white and the huntsman', distributed: 2012})  
3 return m,n;
```

Création des relations

```
⚠ 1 MATCH (a:Person),(b:Movie)
   2 WHERE a.name = 'Ibtissam Essadik' AND b.title = 'snow white and the huntsman'
   3 CREATE (a)-[r:Watched]->(b)
   4 RETURN a,b;
```

```
$ MATCH (a:Person),(b:Movie) WHERE a.name = 'Ibtissam Essadik' AND b.title = 'snow white and the
```



- On peut également créer des nœuds et des relations avec des autres nœuds déjà créés

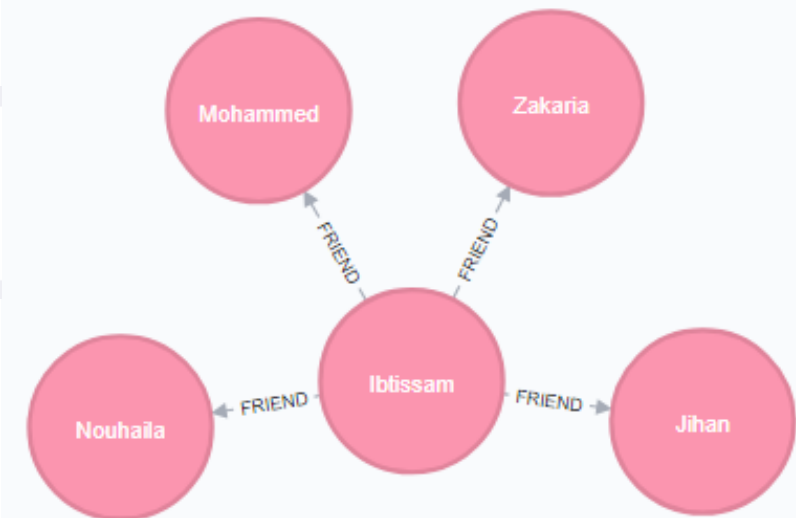
```
1 MATCH (p:Person {name:"Ibtissam"})
2 CREATE (p)-[like:LIKE{since:2018}]->(neo:Database {name:"Neo4j",type:"NoSql Database" })
3 RETURN p,like,neo
```

- FOREACH vous permet d'exécuter des opérations de mise à jour pour chaque élément d'une liste.

```
1 MATCH (p:Person {name:"Ibtissam",born:1996})
2 FOREACH (name in ["Jihan","Zakaria","Nouhaila","Sara","Mohammed"] )
3 CREATE (p)-[:FRIEND]->(:Person {name:name})
```

- Créer des Nœuds avec plusieurs type et labels

```
1 MATCH (neo:Database {name:"Neo4j"})
2 MATCH (p:Person {name:"Zakaria"})
3 CREATE (p)-[:FRIEND]->(:Person:Expert {name:"Youssef"})-[:WORKED_WITH]->(neo)
```



Modifications des propriétés

- Ajout d'une nouvelle propriété:

```
1 MATCH (n { name: 'Ibtissam' })
2 SET n.age = 22
3 RETURN n
```

- Ajout des propriétés spécifiques à l'aide du { } et +:

```
1 MATCH (p { name: 'Ibtissam' })
2 SET p += { age: 22, student: TRUE , city: 'Marrakech' }
3 RETURN p.name, p.age, p.student, p.city
```

- Copier les propriétés entre les nœuds et les relations

```
⚠ 1 MATCH (at { name: 'Ibtissam' }),(pn { name: 'Jihan' })
2 SET at = pn
3 RETURN at.name, at.age, pn.name, pn.age
```

- Modification du type d'une propriété:

```
1 MATCH (n { name: 'Ibtissam' })
2 SET n.age = toString(n.age)
3 RETURN n.name, n.age
```

- Suppression de toutes les propriétés :

```
1 MATCH (p { name: 'Ibtissam' })
2 SET p = { }
3 RETURN p.name, p.age
```

- Ajout d'une étiquette(label) sur un nœud :

```
1 match (n { name: 'Ibtissam'})
2 set n :Student
3 return n.name, labels(n) AS labels
```

Suppression des nœuds et des relations

- Supprimer un seul nœud :

```
1 MATCH (n:Person { name: 'Widad' })  
2 DELETE n
```

- Supprimer tous les nœuds et relations :

```
1 MATCH (n)  
2 DETACH DELETE n
```

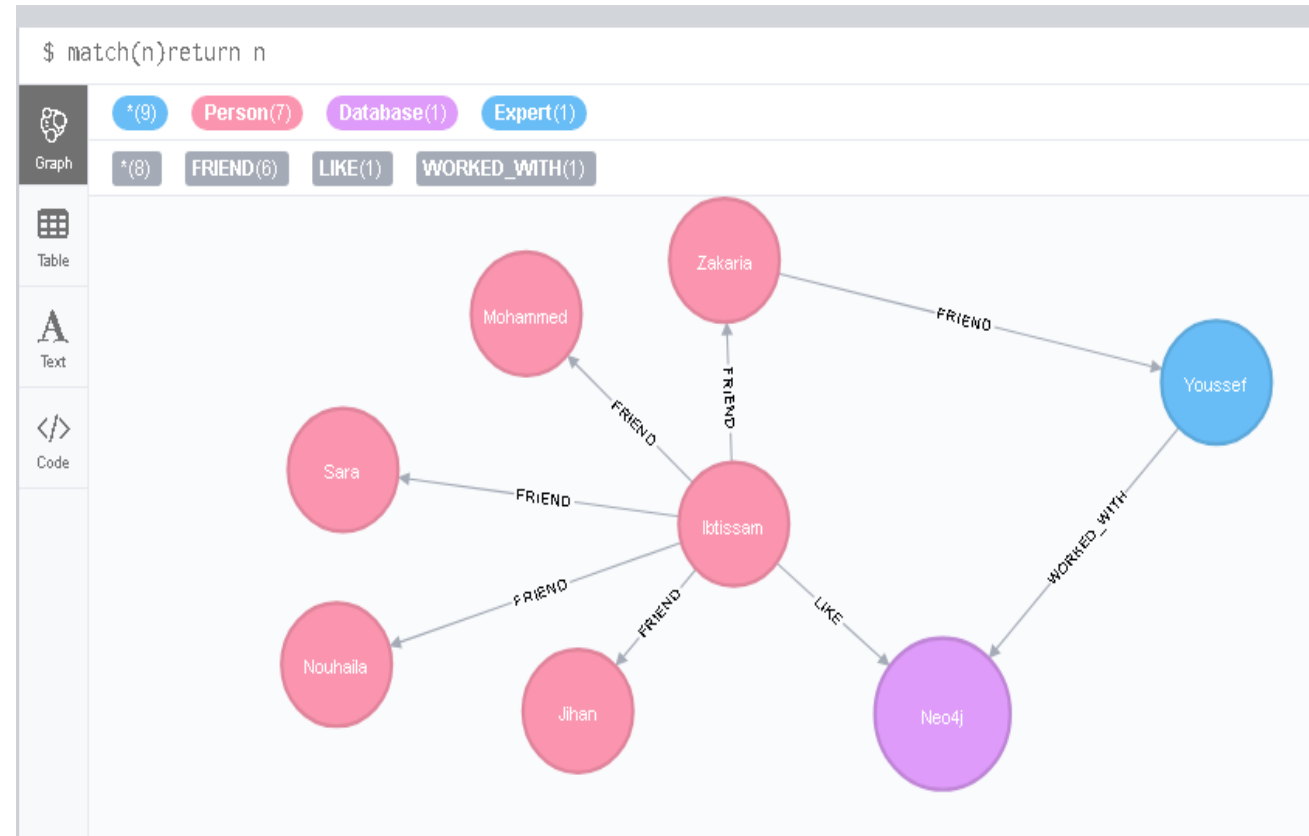
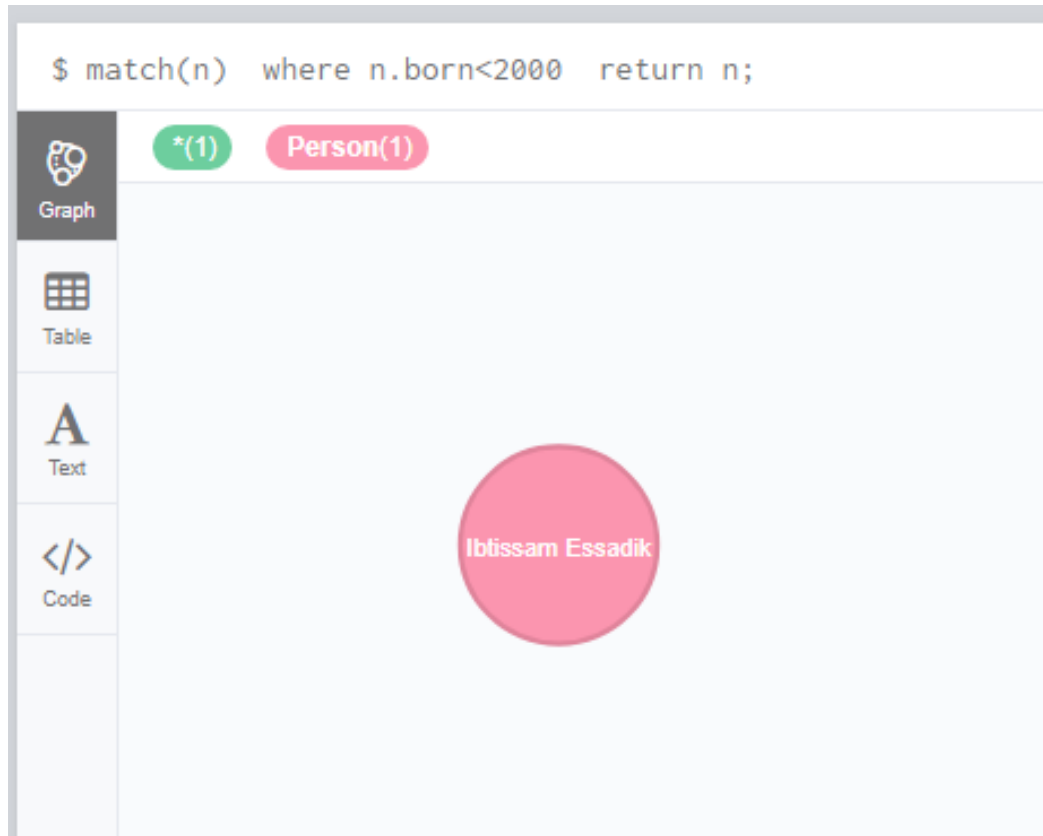
- Supprimer un nœud avec toutes ses relations :

```
1 MATCH (n { name: 'Ibtissam' })  
2 DETACH DELETE n
```

- Supprimer uniquement les relations :

```
1 MATCH (n { name: 'Ibtissam' })-[r:FRIEND]->()  
2 DELETE r
```

Sélection de données avec MATCH



interroger les nœuds avec l'étiquette (Label)spécifiée

\$ match(n:Movie) return n;

Graph

Table

Text

Code

*(1)

Movie(1)

snow white and the hunt...

interroger les nœuds avec un attribut spécifiée

\$ match(n{name:'Ibtissam Essadik'}) return n;

Graph

Table

Text

Code

*(1)

Person(1)

Ibtissam Essadik

```
$ MATCH (p{name:"Ibtissam"})-[:FRIEND]->(n{name:"Sara"}) RETURN p,n
```

Graph

Table

Text

Code

*(2)

Person(2)



```
$ MATCH (n) RETURN n LIMIT 5
```

Graph

Table

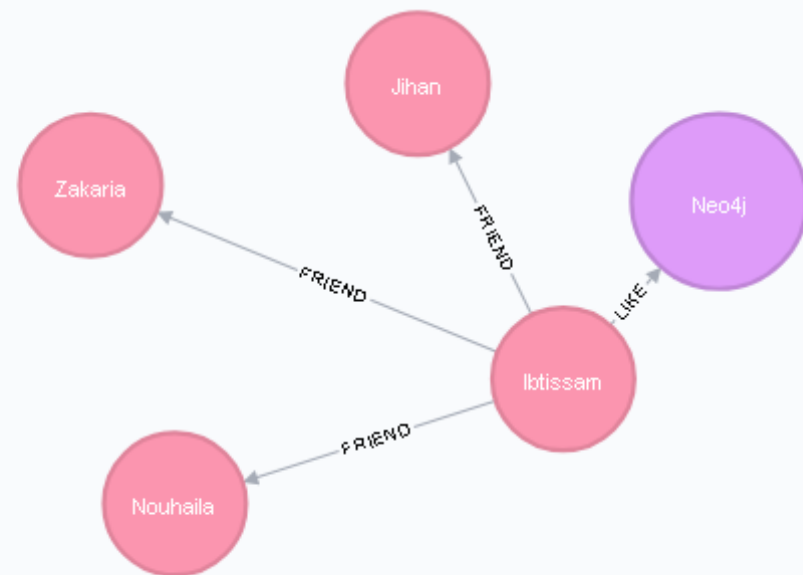
Text

Code

*(5)

Person(4)

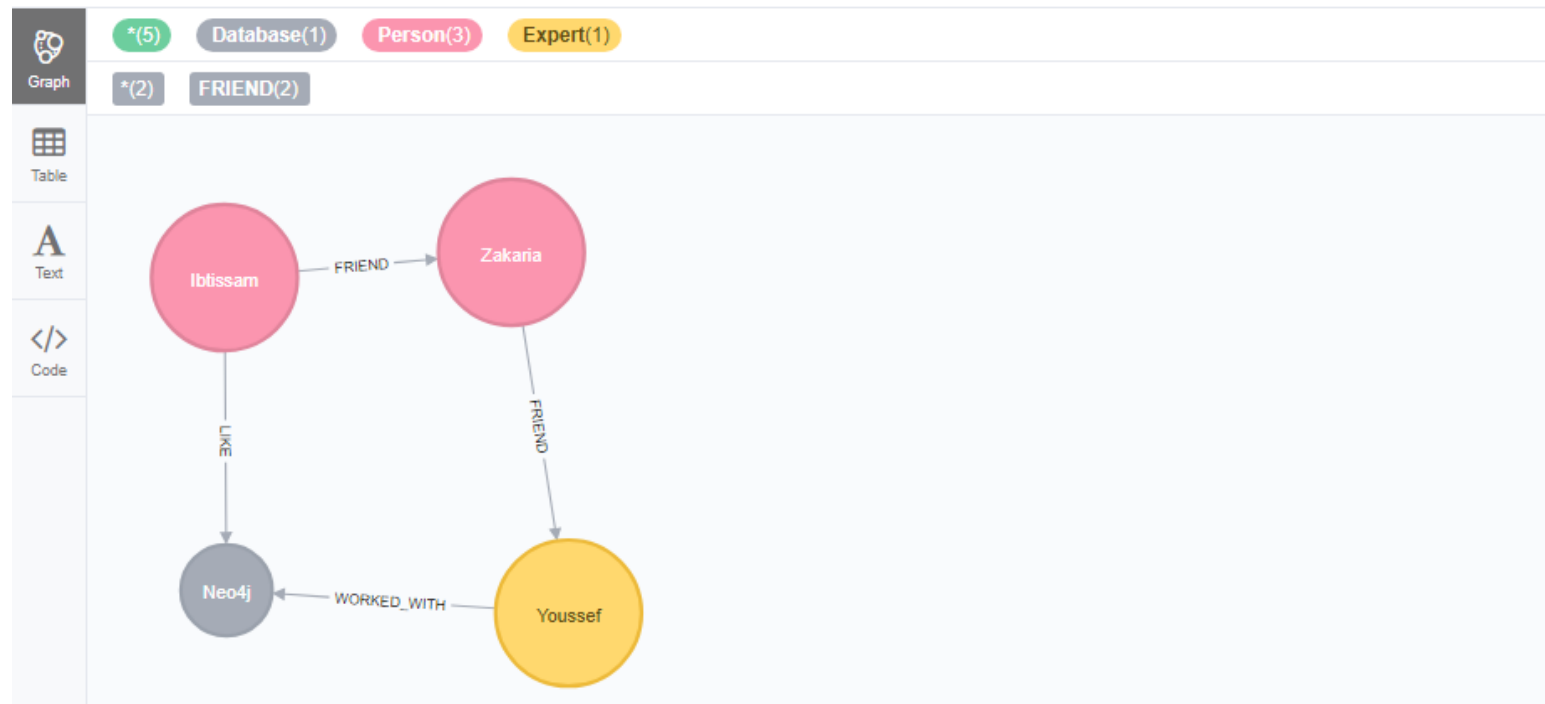
Database(1)



- Trouvez dans votre réseau quelqu'un qui peut vous aider à apprendre Neo4j (shortest path)

```
1 MATCH (you {name:"Ibtissam"})
2 MATCH (expert)-[:WORKED_WITH]->(db:Database {name:"Neo4j"})
3 MATCH path = shortestPath( (you)-[:FRIEND*..5]-(expert) )
4 RETURN db,expert,path
```

```
$ MATCH (you {name:"Ibtissam"}) MATCH (expert)-[:WORKED_WITH]->(db:Database {name:"Neo4j"}) MATCH path = shortestPath
```



Créer et supprimer un index

- Dans Neo4j, vous pouvez créer un index sur une propriété sur tout nœud ayant reçu une étiquette. Une fois que vous avez créé un index, Neo4j le gère et le tient à jour chaque fois que la base de données est modifiée.

```
$ CREATE INDEX ON :Person(name)
```

- Dans l'exemple ci-dessus, nous créons un index sur la propriété Name de tous les nœuds portant l'étiquette Person.
- Dans le navigateur Neo4j, vous pouvez passer en revue tous les index et toutes les contraintes à l'aide de la commande: schema.

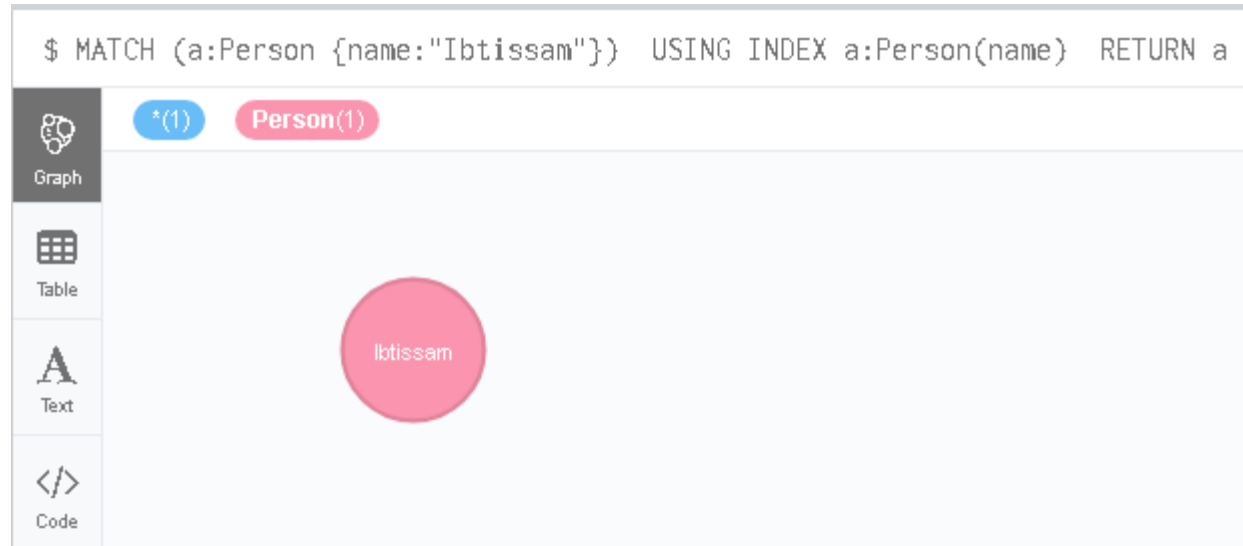
```
$ :schema
```

```
Indexes
```

```
ON :Person(name) ONLINE
```

```
No constraints
```

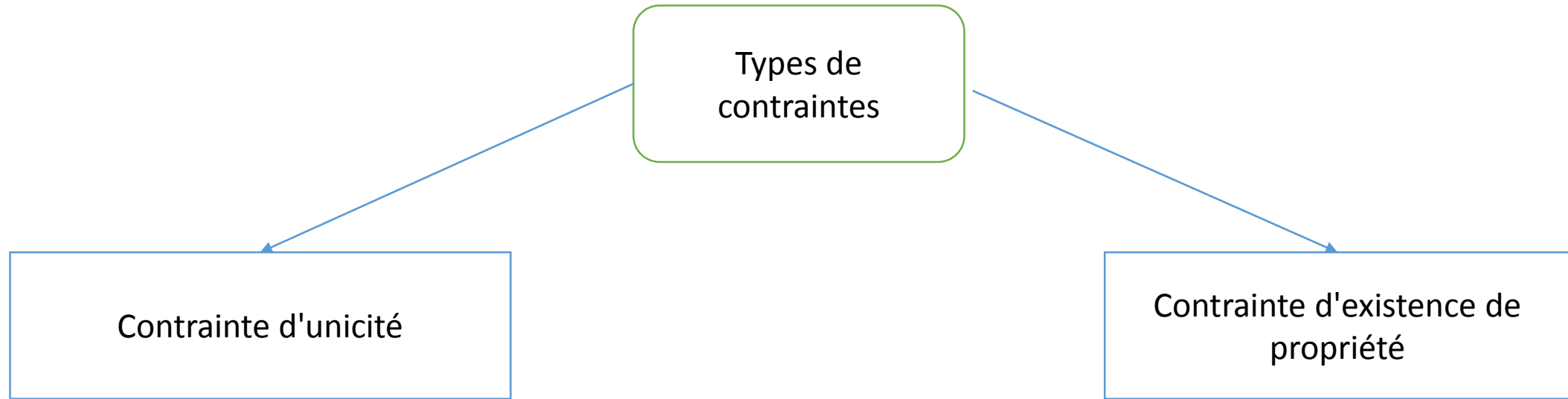
- Cependant, Neo4j vous permet également d'appliquer un ou plusieurs index avec un indice(index Hint). Vous pouvez créer un indice en incluant USING INDEX ... dans votre requête.



- pour supprimer notre index créé précédemment, nous pouvons utiliser l'instruction suivante:

```
$ DROP INDEX ON :Person(name)
```

Créer et supprimer des contraintes



- Spécifie que la propriété doit contenir une valeur unique (c'est-à-dire que deux nœuds avec une étiquette **Database** ne peuvent partager une valeur pour la propriété **Name**.)

- Garantit qu'une propriété existe pour tous les nœuds avec une étiquette spécifique ou pour toutes les relations avec un type spécifique.
- Les contraintes d'existence de propriété ne sont disponibles que dans Neo4j Enterprise Edition.

- Pour créer une contrainte d'unicité dans Neo4j, utilisez l'instruction CREATE CONSTRAINT ON. Comme ça:

```
$ CREATE CONSTRAINT ON (a:Database) ASSERT a.name IS UNIQUE
```

- Nous pouvons afficher la contrainte que nous venons de créer à l'aide de la commande: schema :

```
$ :schema

Indexes
  ON :Database(name) ONLINE  (for uniqueness constraint)

Constraints
  ON ( database:Database ) ASSERT database.name IS UNIQUE
```

- Vous pouvez vérifier que la contrainte fonctionne réellement en tentant de créer deux fois la même base de données.

```
$ CREATE (a:Database{name: "Neo4j"}) RETURN a
```



Error

ERROR

Neo.ClientError.Schema.ConstraintValidationFailed

```
Neo.ClientError.Schema.ConstraintValidationFailed: Node(45) already exists with label `Database` and property `name` = 'Neo4j'
```

- **Les contraintes d'existence de propriété** peuvent être utilisées pour garantir que tous les nœuds ayant une étiquette donnée ont une propriété donnée.
- Par exemple, vous pouvez spécifier que tous les nœuds étiquetés avec **Database** doivent contenir une propriété **Name**.
- Pour créer une contrainte d'existence de propriété, utilisez la syntaxe suivante:

```
$ CREATE CONSTRAINT ON (a.Database) ASSERT exists(a.name)
```

- Donc, pour supprimer notre contrainte précédemment créée (et son index associé), nous pouvons utiliser l'instruction suivante:

```
$ DROP CONSTRAINT ON (a:Database) ASSERT a.name IS UNIQUE
```

- Vous pouvez maintenant utiliser la commande: schema pour vérifier que la contrainte applicable (et son index associé) a été supprimée du schéma:

```
$ :schema
```

```
No indexes
```

```
No constraints
```

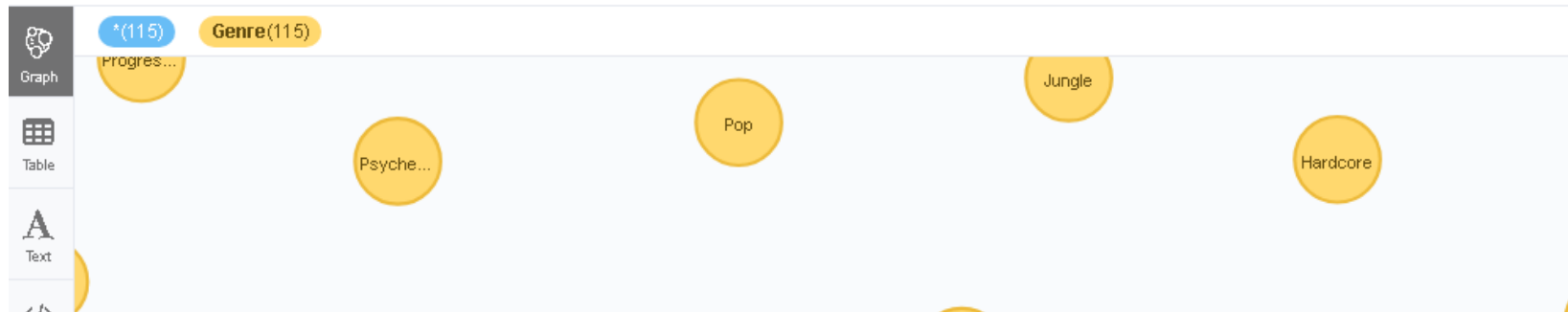
Importer des données depuis un fichier CSV

- Chargeons un fichier CSV appelé genres.csv en utilisant le protocole HTTP. Ce n'est pas un fichier volumineux - il contient une liste de 115 genres musicaux, ainsi il créera 115 nœuds (et 230 propriétés).

```
1 LOAD CSV FROM 'https://www.quackit.com/neo4j/tutorial/genres.csv' AS line
2 CREATE (:Genre { GenreId: line[0], Name: line[1]})
```

```
$ match (n:Genre) return n
```

```
$ match (n:Genre) return n
```



```
1 LOAD CSV WITH HEADERS FROM 'https://www.quackit.com/neo4j/tutorial/tracks.csv' AS line FIELDTERMINATOR ';'
2 CREATE (:Track { TrackId: line.Id, Name: line.Track, Length: line.Length})
```

Fonctions

- **Fonctions de prédicats** : exists(), all()...
- **Fonctions d'agrégation** : count(), max(), min(), avg(), collect(), sum()...
- **Fonctions scalaires**: id(), length(), size(), properties()...
- **Fonctions de liste**: keys(), labels(), nodes(), relationships(), reverse()...
- **Fonctions mathématiques – numériques** :abs(), ceil(), round(), rand()...
- **Fonctions mathématiques – logarithmique** :exp(), sqrt(), log(), e()...
- **Fonctions mathématiques – trigonométriques** :cos(),acos(), acos(),pi(), radians()...
- **Fonctions temporelles** :date(),datetime(), localtime()...

Un peu de pratique !



Question time!

