Les influenceurs sur Twitter

Rapport pour UE NFE204

PHELIZOT Yvan
Semestre 1 – 2017

le cnam

Youtube est un site d'hébergement de vidéos apparu en février 2005. En février 2017, Youtube annonçait un milliard d'heures de vidéos vues quotidiennement [youtube]. Grâce à cette plateforme, de nouveaux métiers sont apparus comme « Youtubeur », vidéaste amateur ou professionnel qui réussissent à vivre de leur production qu'ils partagent gratuitement. Comment vivre sans faire payer le visionnage de leurs vidéos, modèle que l'on retrouve sur les plateformes de VoD (Video On Demand) ?
Pour réussir à obtenir un salaire suffisant, plusieurs méthodes sont possibles : utiliser la monétisation de leurs vidéos par l'inclusion de publicités au moment du visionnage [youtube-money]ou les coopérations avec différentes marques. C'est le « placement de produits » [youtube-ads]. Cette méthode permet aux youtubeurs de gagner des salaires conséquents à l'image des sportifs de haut niveau. Pour y arriver, il est nécessaire d'avoir une influence importante.
Le pouvoir des youtubeurs se définit généralement par le nombre de personnes qui les suivent : les « followers ». Les followers sont ainsi en première ligne des publicités ciblées qui leur sont envoyés sous forme de contenus, des fois dissimulés dans une vidéo souvent anodine, se présentant comme un conseil bienveillant, profitant de leur influence sur leur communauté dans l'espoir de vendre quelques produits.
L'exemple de Youtube est symptomatique de cette tendance que l'on retrouve sur l'ensemble de réseaux sociaux, comme Instagram, Snapchat ou Twitter.
Intéressons-nous à la notion d' « influence ». Est-il possible de la mesurer ? Pour essayer de répondre à cette question, nous utiliserons Twitter. Twitter propose notamment une interface simple offrant énormément d'informations sur les messages échangés (les « tweets »). À partir des tweets échangés, nous construirons un modèle nous permettant d'identifier les influenceurs sur Twitter.
Twitter reçoit 6000 tweets par seconde. Ce volume et le problème nous imposent d'utiliser un type de base de données prévue à cet effet : les bases de données graphe. Nous nous appuierons donc sur un des leaders de ce domaine : Neo4J. L'étude du problème d'influence nous servira de support pour présenter le fonctionnement d'une base de données et d'étudier Neo4J, principalement sur sa résistance aux pannes.
Enfin, dans une dernière étape, nous tenterons d'identifier les influenceurs sur Twitter en utilisant les données récoltées.

Illustration 1: Influenceur, le nouveau métier de rêves?
https://www.youtube.com/watch?v=5MEUVyHljIs

Sommaire

I. Les influencers sur Twitter

I.1 Un réseau social

D'après [wiki-social], « un réseau social représente un groupement qui a un sens : la famille, les collègues, un groupe d'amis, une communauté, etc. Il s'agit d'un agencement de liens entre des individus et/ou des organisations. ». On retrouve les mêmes notions que dans un graphe : des liens entre différents individus.

En général, par extension, on associé « réseau social » à « média social », ce que sont en fait les plateformes telles que Youtube ou Twitter.

I.2 Twitter, un réseau social

Twitter est une grande plateforme des réseaux sociaux, à l'instar de Youtube, Facebook, Instagram ou SnapChat. Il jouit d'un usage important et est utilisé par de nombreuses personnalités politiques. Par exemple, l'actuel président des États-Unis d'Amérique, Donald Trump, en est un utilisateur assidu[1].
Un utilisateur de Twitter (un « **twitto** » ou « **twitteur** ») peut écrire un message de 280 caractères (un « **tweet** »). Ce message peut être lu par n'importe qui. Si un utilisateur est intéressée par suivre les différentes productions d'un membre donné, il peut s'abonner à son flux : c'est la fonction « **following** ». Elle verra alors apparaître dans son flux d'information les messages en temps réel.
Il est aussi possible de notifier un utilisateur en incluant le pseudonyme de l'utilisateur dans le tweet via une arobase « @ ». De même, il est possible d'associer son tweet à un thème grâce au caractère dièse « # » : le fameux «hashtag » de Twitter. Enfin, il peut aussi retransmettre le message sans le modifier afin de le rendre visible dans le flux de ces followers : c'est le « **retweet** ». Cela permet de retransmettre un message
Comme dans une messagerie, n'importe qui peut dialoguer et répondre à un message posté sur Twitter, générant ainsi des débats sans fin[2].
Twitter met à disposition des développeurs une API[3]. Celle-ci peut être accédée directement ou via des bibliothèques logicielles. On trouve des implémentations compatibles avec cette API avec la majorité des langages. Cette API offre un mode temps réel (« stream ») qui permet d'avoir accès aux messages dès qu'ils sont postés (d'autres modes existent, certains sont payants).

I.3 Le « graphe » social

I.3.a) Théorie des graphes

Dans la théorie des graphes, un graphe est un objet constitué de nœuds (« **nodes** » ou « **vertices** » en anglais) et des arêtes (« **edges** » en anglais). Dans la base de données graphe, les arêtes sont des relations entre les nœuds.

Chaque nœud est un élément de notre domaine. Les arêtes sont les relations entre les éléments.

De nombreux algorithmes sont associés au modèle des graphes. Par exemple, pour trouver le chemin le plus court, on peut utiliser l'algorithme de Bellman-Ford, l'algorithme de Djisktra ou

1 @realDonaldTrump - https://twitter.com/realDonaldTrump
2 https://twitter.com/wendys/status/847478772311834626?lang=en
3 https://developer.twitter.com/en/docs

l’algorithme A*. Dans le cas des influenceurs, cela permettrait de répondre à des questions comme : « Combien de personnes me séparent d’un influenceur ? Est-ce qu’il existe d’autres chemins ? »

I.3.b) Les graphes appliqués aux réseaux sociaux

Les relations entre les différents membres d’un réseau social, comme celui de Twitter, peuvent être modélisés sous forme d’un graphe. Ainsi, un membre d’un réseau pourra être vu comme un nœud d’un graphe.

Une relation d’amitié peut être représentée sous la forme d’un lien entre deux amis, c’est-à-dire une arête entre deux nœuds.

Une relation peut aussi avoir une direction. Par exemple, dans le cas de Twitter, une personne peut être abonnée au flux d’une autre personne, mais la réciproque n’est pas nécessaire.

De plus, il est possible de représenter une relation sans direction (comme la relation d’amitié) comme deux relations directionnelles.

I.3.c) Comment résoudre le problème avec un graphe

Nous allons voir que notre problème de détection des influenceurs peut se réduire à un problème de graphe social, moyennant quelques hypothèses et simplifications.

Tout d’abord, on peut considérer que l’influence se définit par la manière dont une personne réagit à un stimulus produit par un influenceur. Ainsi, un twitto lambda ayant des followers peut écrire un tweet, mais être ignoré de tous. Un tweet d’un influenceur sera, par contre, activement commenté et partagé. Chaque twitto est donc un nœud du graphe, relié entre eux par les messages et les interactions des autres twittos avec ces messages.

On pourra ainsi mesurer le pouvoir d’un influenceur en combinant plusieurs critères :

- Nombre de personnes touchées par un retweet
- Réactions face à un tweet. Dans ce cas, on utilisera une analyse de sentiment qui nous donnera une note pour évaluer la réaction positive ou négative.
- Mention dans un tweet. Plus un utilisateur a d’influence, plus il est probable qu’on parle de lui en bien.

II. Une base de données graphe

II.1 Une base de données graphe ?

Un système de gestion de base de données graphe (et par extension, une base de données graphe) est un système offrant des méthodes pour manipuler un modèle de données graphe. Le graphe est l'élément de modélisation centrale, de même que la table l'est pour les bases de données relationnelles ou le document pour les bases de données documents.

Les bases de données graphe font partie du mouvement NoSQL.

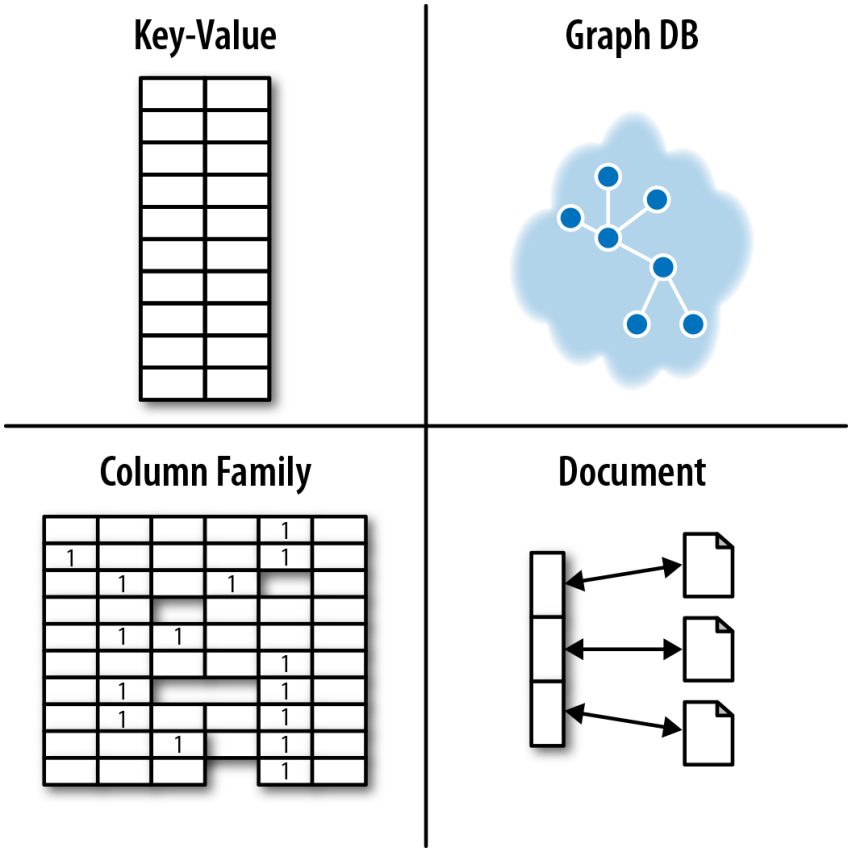


Illustration 2: Quadrant NoSQL, extrait de « Graph Databases »

De plus, il faut noter qu'il existe des différences entre les bases de données graphe, notamment sur deux points : stockage (natif et non-natif) et traitement (natif et non-natif). L'aspect natif indique que la base est optimisée pour les graphes sur le point considéré. Le schéma suivant, issu du même livre, illustre cette répartition.

Neo4J est optimisée pour la gestion des graphes sur ces deux aspects, ce qui en fait une base performante.

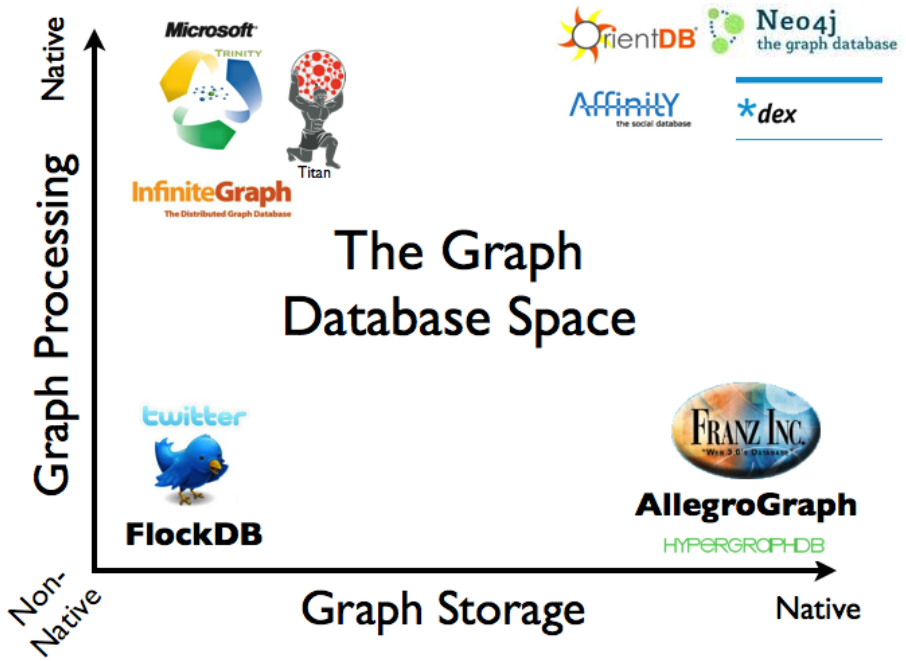


Illustration 3: Répartition des différents types de bases de données graphe

II.2 Pourquoi utiliser une base de données graphe ?

Une des questions à se poser est ce qui fait la force des bases de données graphe, comparée aux autres bases de données, qu'elles soient de type SQL ou NoSQL.

Les bases de données relationnelles modélisent la notion de « relation » entre entité sous la forme d'identifiants. Par exemple, dans une base de données relationnelle, le lien entre entité est représenté par une jointure. On utilise soit une référence vers une table (clé étrangère ou « foreign key », soit une table de jointure. Les relations ne sont pas des éléments de première de ces bases. Ainsi, des requêtes faisant intervenir ces relations sont lentes, malgré la présence d'indexes.

Les bases de données NoSQL ne mettent pas non plus en avant l'aspect « relation » entre les entités. Par exemple, une base de données document comme MongoDB préconise de regrouper le maximum d'information dans un document avec de ne pas à avoir à rechercher d'autres données dans d'autres documents.

Le tableau suivant, extrait du livre « Graph Databases », met en évidence les performances obtenues lorsque l'on essaie d'accéder à des relations profondes (par exemple, la profondeur 3 correspond à des « amis d'amis d'amis ») :

Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

L’augmentation du coût vient du fait que chaque saut oblige à effectuer de nouveaux une recherche sur un index. Un recherche dans un index coûte log(n), avec n le nombre d’éléments dans l’ensemble. Si on suppose que l’on trouve en moyenne m voisins répondant à la requête par élément, le premier niveau remonte donc m éléments. Pour atteindre le deuxième niveau, il est nécessaire d’effectuer n * m * log(n) requêtes. En supposant que le deuxième niveau remonte aussi m éléments correspondants en moyenne, on doit de nouveau effectuer un ensemble de requêtes de complexité (n * m) * m * log (n).

Pour gagner en performance, les bases de données graphe utilisent la notion d’ « ***index-free adjacency*** ». Le principe est que chaque nœud connaît ces voisins directs au sens d’une relation donnée. Pour un nœud donné, il devient facile de connaître les voisins, puisqu’il suffit de parcourir l’index local au nœud. Par récurrence, les voisins de voisins s’obtiennent en parcourant l’index des voisins.

II.3 Le graphe dans une base de données graphe

Le paragraphe suivant présente le glossaire utilisé dans le domaine des graphes. Ils existent d’autres types de modèles de graphe :

- Resource description framework (RDF)
- Hypergraphs
- Triples
- Property graph : Neo4J est basé sur ce modèle de représentation de graphe. Nous nous concentrerons sur la description de ce modèle.

La liste suivante présente les définitions des termes que nous utiliserons par la suite :

- Nœud (EN : node) : il s’agit d’un enregistrement dans un graphe. Des nœuds d’un même ensemble peuvent avoir des propriétés différentes. De même que pour certaines bases de données, comme MongoDB, il n’y a pas de schéma imposé pour les nœuds. Il revient à l’application de gérer l’absence des données.
- Label : un nœud peut avoir de zéro à n labels. Ils permettent de former des groupes parmi les nœuds.
- Propriété (EN : properties) : valeur associée à un nœud. Une propriété est un élément de type « clé/valeur », de type string, nombre ou booléen.
- Relation : il s’agit des arêtes dirigées au sens des graphes. Elles connectent les nœuds. Elles ont un nom.

II.4 Modélisation pour une base de données graphe

Les relations sont l’élément central dans une base de données, contrairement aux autres systèmes. Chaque nœud contient directement et physiquement la liste de ces voisins par type de relation. La modélisation d’une base de données SQL s’appuie sur la modélisation selon les différentes formes

normales afin de réduire la duplication et d'améliorer la consistance. Il est nécessaire de définir les différents formats pour les données (le schéma). Les relations entre entités sont souvent implémentées via des tables de jointure. Dans les bases de données graphes, le principe de relation est un élément central. Les entités sont prévues pour être reliées et avoir des relations entre elles. L'un des changements fondamentaux est le replacement des jointures entre tables dans un modèle NoSQL par une relation. Prenons l'exemple suivant : « Alice vit à Paris ». En SQL, nous aurions le schéma suivant :

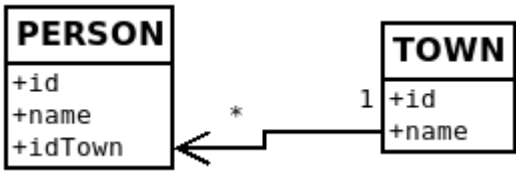


Illustration 4: Modélisation entité-association

De même, avec la phrase suivante : « Bob est ami avec Alice », il est nécessaire d'avoir une table de jointure pour lier deux personnes.

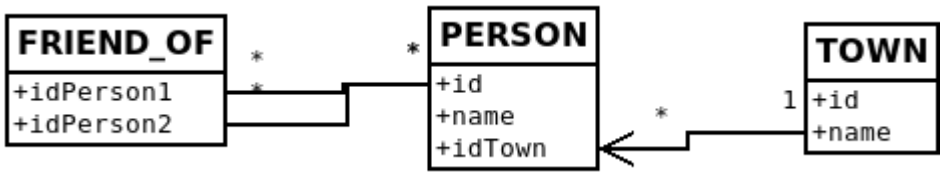


Illustration 5: Modélisation SQL

Sous Neo4J, nous obtiendrions la représentation suivante :

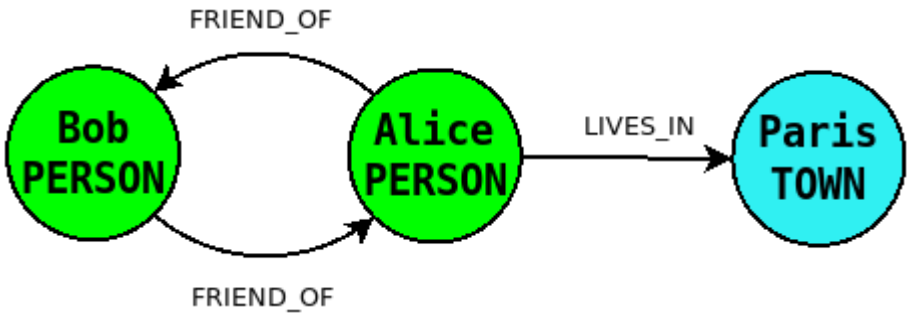
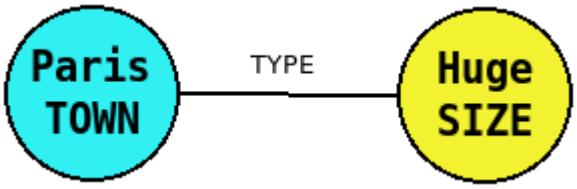


Illustration 6: Modélisation graphe

Ainsi, les tables deviennent des nœuds de type PERSON, les jointures deviennent des arcs entre les nœuds. Il est possible de pousser plus loin la dénormalisation. Supposons que TOWN est un attribut taille avec plusieurs valeurs (petite, moyenne, grande, très grande…). Pour accélérer les recherches, on peut introduire un type de nœud supplémentaire pour faciliter la recherche des villes :



Les requêtes au format Cypher sont présentées en annexe du document.

III. Neo4J, une base de données graphe

Comme évoqué dans l'introduction, nous nous intéressons au système Neo4J. Ce paragraphe introduit Neo4J et présente comment installer et interagir avec la base.

III.1 Présentation de Neo4J

Neo4J est une base de données graphe. Elle est développée en sur Java (tout comme Cassandra par exemple) depuis 2007 et est aujourd'hui en version 3.

Neo4J existe en deux versions : **Community** et **Entreprise**. La version Community est mise à disposition avec une licence Open Source GPL v3. Le code de cette version est d'ailleurs téléchargeable ici : https://github.com/neo4j. La version Entreprise donne accès aux fonctionnalités avancées :

- Haute-disponibilité via des clusters
- Sécurité (compte, rôle …)
- Optimisations (requêtes compilées …)
- Propriétés ACID pour les transactions

Ces fonctionnalités sont vues plus en détail au chapitre IV – Aspects avancés de Neo4J. La version Entreprise est gratuite pour le développement d'applications « Open Source ».

Neo4J est reconnu pour ces qualités en tant que base de données graphe. De plus, c'est un outil open source. La documentation est claire et offre de nombreux exemples, facilitant son utilisation. Ces raisons ont motivé le choix de cette base pour réaliser ce projet.

III.2 Installation de Neo4J

La base Neo4J peut être installée de différentes manières :

- Via un utilitaire (Neo4J Desktop)
- En ligne de commande
- Via Docker

III.2.a) Avec Neo4J Desktop

Neo4J met à disposition un outil appelé « Neo4J Desktop ». Il facilite la création d'une nouvelle base de données en quelques clics. Il donne notamment accès à la version Entreprise. C'est une solution simple pour tester et prendre en main rapidement Neo4J.

Il est disponible sous Windows, Mac OS et Linux et est téléchargeable à l'adresse suivante : https://neo4j.com/download/?ref=product

Pour créer une nouvelle base, il suffit de cliquer sur le bouton « New » et de choisir la version qui nous intéresse. L'instance est alors disponible. En cliquant sur le bouton « Start », la base démarre.

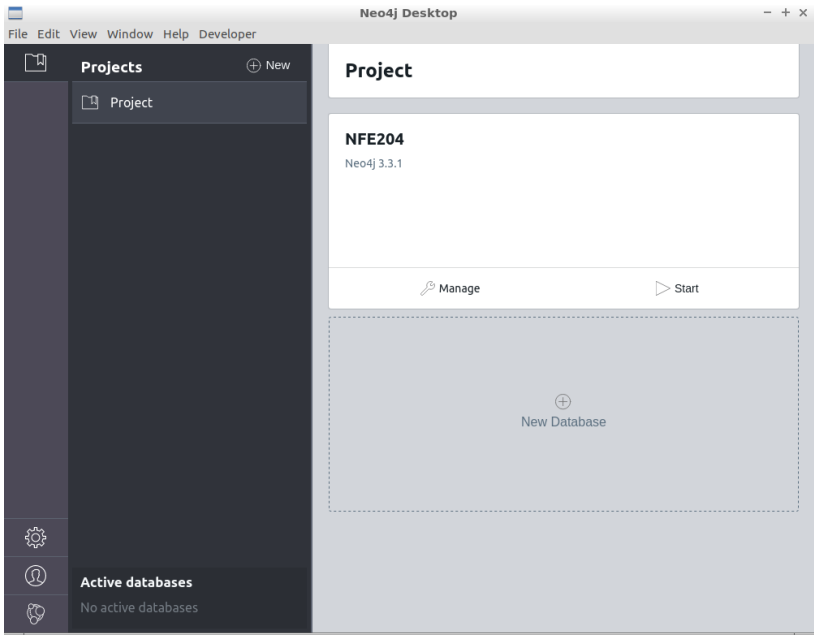


Illustration 7: Capture d'écran de neo4j Desktop

III.2.b) En ligne de commande

Il est aussi possible de lancer une instance de Neo4J depuis la ligne de commande. Les distributions sont accessibles ici : https://neo4j.com/download/other-releases/#releases.

III.2.c) Avec Docker

Une des manières les plus simples d'installer un serveur Neo4J est de créer un container Docker à partir de l'image officielle :

```
$ docker run --publish=7474:7474 –publish=7687:7687 \
    --volume=$HOME/neo4j/data:/data neo4j
```

III.3 Accès à Neo4J

On peut ensuite à Neo4J de différentes façons:

- Neo4J Browser
- HTTP API
- Bolt

On peut accéder à l'interface appelée « Neo4J Browser » en se connectant à l'adresse http://localhost:7474/browser/.

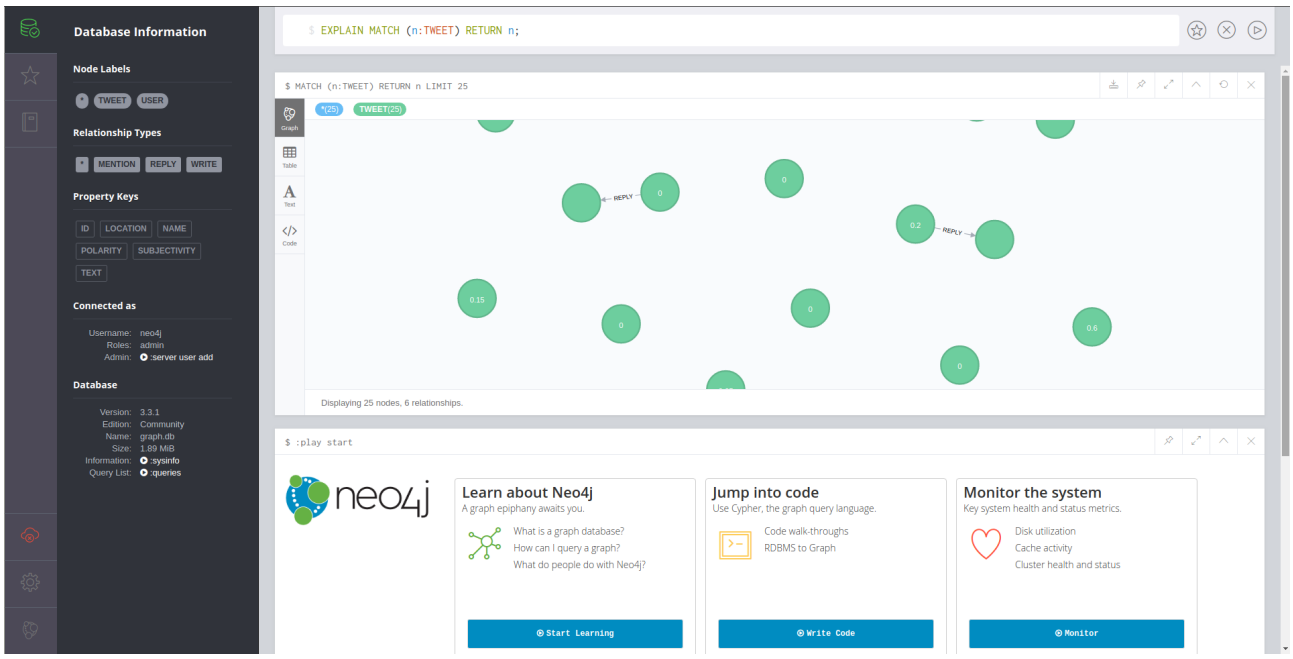


Illustration 8: Capture d'écran de Neo4J Browser

« Neo4J Browser » se décompose de la façon suivante :

- La barre du haut est l'invité de commande. Il sert à envoyer des requêtes Cypher (voir le paragraphe VII.1 - Le langage Cypher)
- Le menu donne accès aux différents types de nœuds et relations
- Enfin, la partie centrale permet de visualiser les données.

Neo4j offre l'accès via une interface REST.

```
$ curl --user neo4j:root -H "Accept: application/jon; charset=UTF-8"
http://localhost:7474/db/data
```

Des requêtes peuvent être exécutées depuis l'interface.

```
$ curl -X POST --user neo4j:root -H "Accept: application/json; charset=UTF-8" -H
'Content-Type: application/json' -d '{"query": "MATCH (n:TWEET) RETURN n LIMIT
2"}' http://localhost:7474/db/data/cypher
```

L'API donne accès aux ressources enregistrées dans la base :

```
$ curl --user neo4j:root -H "Accept: application/json; charset=UTF-8"
http://localhost:7474/db/data/node/2
```

On obtient une réponse au format JSON :

```
{
  "metadata":{
    "id":2,
    "labels":[
      "TWEET"
    ]
  },
  ...
```

```
  "labels":"http://localhost:7474/db/data/node/2/labels",
  "all_relationships":"http://localhost:7474/db/data/node/2/relationships/all",
  "self":"http://localhost:7474/db/data/node/2",
  "properties":"http://localhost:7474/db/data/node/2/properties",
  "data":{
    "SUBJECTIVITY":0.0,
    "POLARITY":0.0,
    "TEXT":"Quelquefois, le monde est beau!😁🏝 https://t.co/StqGLqVkfd",
    "ID":945955362720665600
  }
}
```


Enfin, le protocole BOLT est un protocole binaire optimisée pour l’interroger de la base Neo4J. Il s’utilise principalement dans les programmes (Python, Java …). Son port par défaut est le 7867.

IV. Aspects avancés de Neo4J

Le paragraphe précédent offre une vue haut-niveau du système Neo4J. Nous nous intéresserons en particulier à l'aspect haute-disponibilité de Neo4J.

IV.1 Indexes

La gestion des relations entre entités est ce qui fait la particularité et la force de Neo4J aussi bien par rapport aux solutions SQL qu'aux solutions NoSQL. Elles cherchent à résoudre des cas où les relations entre entités ont plus de valeur. Comme vu précédemment, le principe d'« index-free adjacency » est mis en œuvre. Les nœuds ont une connaissance physique des autres nœuds avec qui ils partagent des relations.

Neo4J fournit aussi la possibilité de créer des index sur les propriétés. Par exemple, on peut créer un index sur la propriété « **born** ».

```
: create index on :Person(born)
```

On peut voir l'impact de cette optimisation sur la requête donnant le nombre d'acteurs nés la même année :

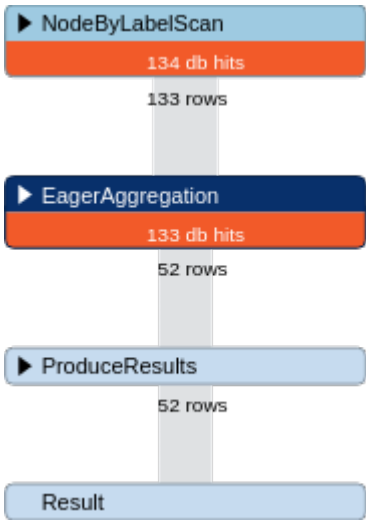


Illustration 9: Plan d'exécution de la requête

Le plan d'exécution de la requête s'obtient avec la commande suivante :

```
: PROFILE MATCH (m:Person) RETURN m.born, count(*)
```

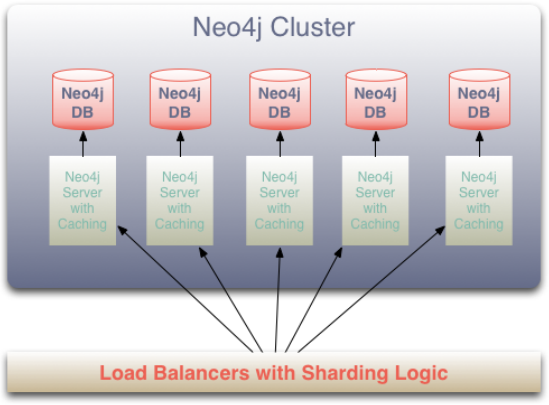
IV.2 Haute-disponibilité via le partitionnement

Pour un graphe, le partitionnement efficace d'un graphe est quasiment impossible. Il s'agit d'un problème NP-complet[4] auquel n'échappe pas Neo4J. Neo4J ne propose pas de solution pour séparer physiquement les nœuds sur les différents serveurs selon un critère de partitionnement. Une copie complète est répliquée sur chacun des nœuds. C'est une limitation importante qui réduit les performances en écriture de Neo4J qui supporte les propriétés ACID. En reprenant le théorème

4 https://en.wikipedia.org/wiki/Graph_partition#Problem_complexity

CAP, Neo4J ne pouvant pas être partitionné, elle assurera malgré tout les propriétés de consistances et disponibilités.

Cependant, Neo4J propose, malgré tout, une solution permettant d'améliorer les performances : le « cache sharding ». Le principe est de mettre un répartiteur de charge devant le cluster Neo4J et répartir les requêtes selon un critère (par exemple, un identifiant). Ainsi, un nœud n'a à garder en mémoire qu'un certain ensemble de nœuds permettant d'améliorer la scalabilité d'un cluster Neo4J.



Neo4j Architecture

Cache Sharding For High Scalability (fromdev.com)

IV.3 Haute-disponibilité via la réplication

Neo4J propose deux modes de fonctionnement pour obtenir une haute disponibilité : High Availability Cluster & Causal Cluster. Nous nous intéresserons principalement au premier type de cluster que nous utiliserons par la suite.

IV.3.a) High Availability Cluster

Principe

Un « High Availability Cluster » implémente un modèle d'architecture maître-esclave à un maître. Un tutoriel est disponible à l'adresse suivante : https://neo4j.com/docs/operations-manual/current/tutorial/highly-available-cluster/

Ce cluster applique le principe du quorum : « Un cluster qui doit être capable de supporter jusqu'à n défaillance d'un nœud maître doit être constitué de 2n + 1 nœuds ». Dis autrement, un nœud maître doit régner sur une majorité des nœuds.

Ainsi, en cas de défaillance d'un nœud maître, un nouveau maître est élu. Le maître est choisi par une élection avec un *quorum*.

La partie suivante présente comment mettre en œuvre un cluster haute-disponibilité constitué de trois nœuds en local. Il est important de noter que la version entreprise de Neo4J est nécessaire, ce qui implique l'acceptation de la licence.

Architecture

Le schéma suivant présente l'architecture d'un cluster HA à trois nœuds.

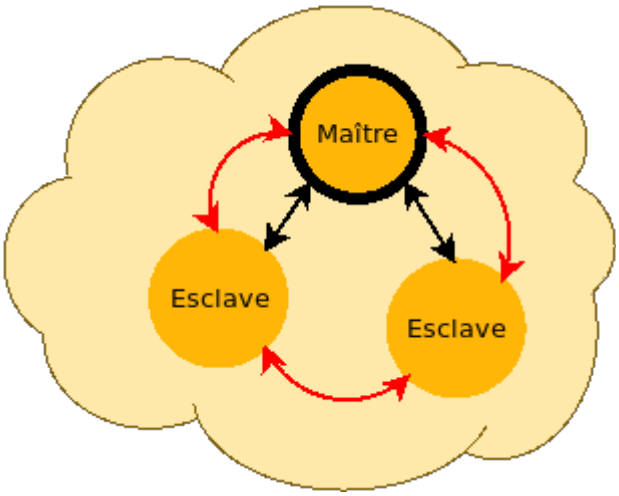


Illustration 10: Architecture d'un cluster HA

Les échanges entre nœuds sont de deux types :

- Échanges liés aux transactions (en noir) : ajout/suppression/modification de données
- Échanges liés à la gestion du cluster (en rouge)

Ces échanges seront analysés dans la partie suivante. Le cluster met en œuvre un protocole de gestion où les nœuds discutent entre eux sur les dernières nouvelles du cluster : est-ce qu'il y a de nouveaux arrivants ? Est-ce qu'il y a des nœuds qui ne répondent plus ? … Il s'agit d'un protocole d'échange de type « gossip » (ragot).

Le système d'élection est géré par le protocole PAXOS. Ce protocole est assez répandu[5] mais il est réputé complexe. Nous testerons dans le paragraphe « Tolérance à la panne » le fonctionnement de PAXOS pour l'élection d'un maître.

Configuration

Une fois la version entreprise téléchargée (https://neo4j.com/download/), les commandes suivantes permettent de récupérer le binaire sous Linux :

```
$ tar -xf neo4j-enterprise-3.3.2-unix.tar.gz –directory node1
$ cd node1/neo4j-enterprise-3.3.2
```

Il est nécessaire de modifier les lignes suivantes dans le fichier conf/neo4j :

```
dbms.connector.bolt.listen_address=:7688 # A modifier pour éviter les colisions
dbms.connector.http.listen_address=:7474 # A modifier pour éviter les colisions
dbms.connector.https.listen_address=:7473 # A modifier pour éviter les colisions
dbms.mode=HA # Pour activer le mode High Availability
```

5 https://en.wikipedia.org/wiki/Paxos_(computer_science)#Production_use_of_Paxos

```
ha.server_id=1 # Identifiant du nœud dans le cluster, à changer pour chaque nœud
ha.initial_hosts=127.0.0.1:5001,127.0.0.1:5002,127.0.0.1:5003
ha.host.coordination=127.0.0.1:5001 # Le port est à adapter pour chaque nœud (5002 pour le deuxième, …), utiliser pour la gestion du cluster (élection)
ha.host.data=127.0.0.1:6001 # idem, utiliser pour la partie transaction
```

Il faut ensuite répéter cette opération pour les différents nœuds. Nous allons créer trois nœuds.

Il faut ensuite démarrer les différents nœuds :

```
$ bin/neo4j start
...
This HA instance will be operational once it has joined the cluster.
```

Tolérance à la panne

On peut regarder les logs pour observer ce qui se passe. Sur le premier nœud, nous obtenons :

```
2018-01-27 16:43:52.890+0000 INFO  Instance 1 (this server)  entered the cluster
2018-01-27 16:43:52.897+0000 INFO  Instance 1 (this server)  was elected as coordinator
2018-01-27 16:43:52.933+0000 INFO  I am 1, moving to master
2018-01-27 16:43:52.985+0000 INFO  Instance 1 (this server)  was elected as coordinator
2018-01-27 16:43:52.998+0000 INFO  I am 1, successfully moved to master
```

Il est devenu le maître. En regardant les logs des autres nœuds, nous trouvons les informations suivantes (pour plus de lisibilité, les informations non essentielles seront supprimées) :

```
Instance 2 (this server)  entered the cluster
Instance 1  was elected as coordinator
Instance 1  is available as master at ha://127.0.0.1:6001?serverId=1
ServerId 2, moving to slave for master ha://127.0.0.1:6001?serverId=1
Checking store consistency with master
Store is consistent
Catching up with master. I'm at RequestContext[machineId=2]
Now caught up with master
ServerId 2, successfully moved to slave for master ha://127.0.0.1:6001?serverId=1
Instance 2 (this server)  is available as slave at ha://127.0.0.1:6002?serverId=2
```

Les nouveaux nœuds rejoignent le cluster en tant qu'esclave. En arrêtant le maître actuel, on peut voir une nouvelle élection se produire :

```
Instance 1  has left the cluster
Instance 1  is unavailable as master
Instance 2 (this server)  was elected as coordinator
Write transactions to database disabled
I am 2, moving to master
Instance 2 (this server)  is unavailable as slave
I am 2, successfully moved to master
Instance 2 (this server)  is available as master
Database available for write transactions
```

Les logs montrent qu'en cas de disparition du maître, la base devient inaccessible en écriture jusqu'à l'élection. On montre ainsi que la règle du quorum est bien respectée en tuant encore une fois le nouveau maître. Nous nous retrouverons donc avec un seul maître ayant autorité que sur un nœud (lui-même). La majorité n'étant pas présente, le cluster se verrouille et se retrouve disponible uniquement en lecture seule.

Nous avons vu comment le cluster se comportait lors de l'ajout et de la suppression de nœuds de manière « propre », c'est-à-dire en utilisant les outils de gestion de neo4j. Comment le cluster se comporte face à une situation de défaillance logicielle ? Redémarrons le cluster et simulons une panne du système. Pour ce faire, il est de tuer deux instances avec la commande « **kill -9** » au lieu de « **neo4j stop** ». Voici un extrait du fichier « logs/debug.log » présentant la réaction du nœud survivant :

```
Got memberIsFailed(1) and cluster lost quorum to continue, moved to PENDING from SLAVE, while maintaining read only capability.

Context says election is not OK to proceed. Failed instances are: [1, 3]
```

Si on essaie d'insérer des données, l'erreur suivante est obtenue (logs/neo4j.log) :

```
Cannot allocate new entity ids from the cluster master. The master instance is either down, or we have network connectivity problems
```

Ces tests nous montrent que la règle du quorum est bien respectée par Neo4j.

Ragots entres noeuds

Pour vérifier qu'ils sont vivants, les nœuds vérifient le « heartbeat » des autres nœuds du cluster. Ils se servent aussi des autres nœuds avec lesquels ils échangent des informations sur l'état de santé des autres nœuds du cluster. La détection des défaillances est ainsi plus rapide et précise, puisque le choix se fait de manière en recoupant les informations avec les autres. Les extraits suivants présentent les échanges extraits des logs du cluster :

- Un nœud disponible :

```
[o.n.c.p.h.HeartbeatState] Received i_am_alive[3] after missing
```

- Un nœud absent

```
[o.n.c.p.h.HeartbeatState] Received timed out for server 3
[o.n.c.p.h.HeartbeatContext] 2(me) is now suspecting 3
[o.n.c.p.h.HeartbeatState] Received suspicions as Suspicions:[3] from 1
[o.n.c.p.h.HeartbeatContext] 1 is now suspecting 3
```

Le protocole de gestion du cluster est PAXOS. Les messages « suspicions » sont une extension de ce protocole[6]. PAXOS est un protocole complexe et ne sera pas présenté en détail.

Modification de données

Examinons maintenant la manière dont neo4j gère les modifcations dans son cluster, notamment si nous essayons de manipuler des données via un nœud esclave.

Il est possible de vérifier l'état des nœuds en utilisant l'API REST mise à disposition :

- Est-ce que le nœud est un esclave : http://localhost:7474/db/manage/server/ha/slave
- Ou un maître: http://localhost:7474/db/manage/server/ha/master

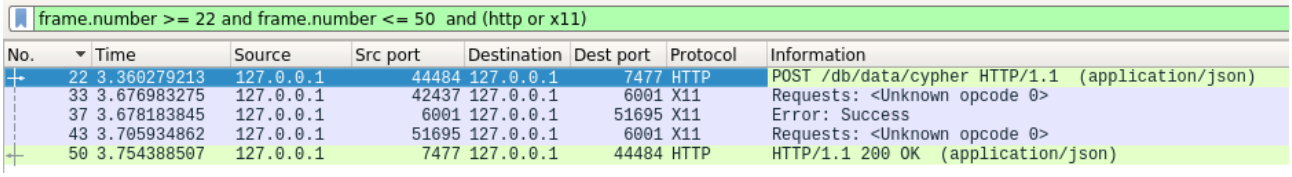
Après vérification, le nœud 3 (port 7477) est un esclave. Insérons une donnée quelconque en base :

```
$ curl -v -X POST --user neo4j:root -H "Accept: application/json; charset=UTF-8" -H "Content-Type: application/json" -d '{
```

6 Generalized Paxos Made Byzantine (and Less Complex) https://arxiv.org/pdf/1708.07575.pdf

```
  "query" : "CREATE (n:Person { name : {name} }) RETURN n",
  "params" : {
    "name" : "Andres"
  }
}' http://localhost:7477/db/data/cypher
...
< HTTP/1.1 200 OK
...
```

L'insertion s'est faite avec succès. On aurait pu s'attendre à une erreur du fait que ce nœud n'est pas sensé autoriser les écritures. On se retrouve dans le même cas que certains systèmes étudiés en cours : le serveur renvoie l'insertion de manière transparente vers le serveur maître. Il est possible de le confirmer en analysant les paquets réseaux transitant lors de l'échange :



Suite à l'insertion, le nœud 3 transmet vers le nœud 1 la requête. Il s'agit de la connexion existante entre le port 42437 (nœud 3 – esclave) et le port 6001 (nœud 1 – maître). Par la suite, le nœud maître propage l'information vers les autres nœuds.

La démonstration précédente montre que Neo4J est une solution offrant une solution de réplication efficace. Pour se protéger des défaillances, des techniques de « failover » sont mises en œuvre comme l'élection d'un nouveau maître et la réplication des données. Le paragraphe suivant présente un autre type de cluster.

IV.3.b) Causal Cluster

De même qu'un cluster « High-Availability », « Causal Cluster » propose une architecture maître-esclave. Dans ce mode de fonctionnement, on distingue des nœuds servant uniquement à la lecture (read replica) des nœuds « cœur » (core server) en charge du maintien de la cohésion des écritures.

Une écriture est acceptée si la majorité des nœuds cœur a validé la transaction. On retrouve la règle du quorum, qui implique aussi ici d'avoir au moins trois nœuds dans le cœur. La gestion du des nœuds cœur se fait en s'appuyant sur l'algorithme RAFT[7]. Ce protocole est notamment utilisé par le service registry Consul. Les données sont ensuite propagées sur les nœuds de réplication.

7 On trouvera une démonstration claire et visuelle de l'algorithme à l'adresse suivante : http://thesecretlivesofdata.com/raft/

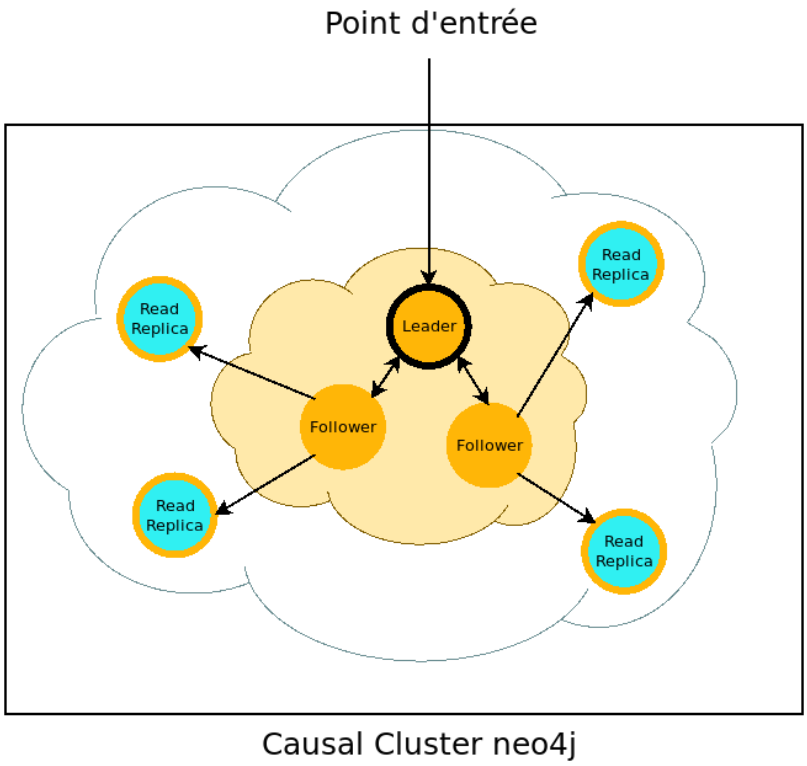


Illustration 11: Schéma d'architecture d'un cluster causal neo4j

V. Mise en œuvre du projet

Cette partie détaille le principe de l'algorithme mis en place pour détecter les influenceurs sur Twitter ainsi que l'utilisation faite du cluster HA neo4j.

V.1 Récupération des données depuis Twitter

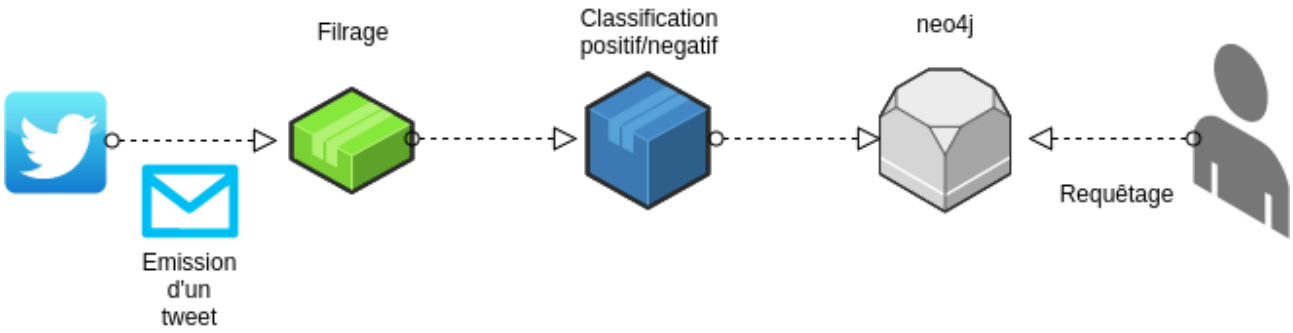
La récupération des tweets se fait en Python via l'API twitter. Elle a l'avantage d'être simple à utiliser. Les tweets seront récupérés en temps réel via l'API « Stream ». Ils vont être enrichis avec les informations sur la notion de sentiment et conservés dans la base neo4j.

Dans la mesure où nous nous intéressons aux influenceurs français, nous devons détecter les tweets français. Twitter ne semble pas permettre de récupérer les tweets uniquement sur la langue. Nous allons donc définir une zone géographique (**location=FRANCE**) et exclure les tweets qui ne sont pas de langue française (**lang != 'fr'**).

Pour chaque tweet récupéré, nous détectons le niveau de sentiment pour chaque tweet, c'est-à-dire si le texte du tweet peut être vu comme positif ou négatif. Afin de simplifier la mise en œuvre, la librairie TextBlob[8] en version française a été utilisée. À partir du texte, deux valeurs sont retournées : la polarité et la subjectivité. Une polarité positive indique un texte positif. La subjectivité est le niveau de confiance dans la polarité retournée. Une subjectivité élevée montre une confiance importante dans la réponse retournée.

La récupération des tweets s'est faite par intermittence sur plusieurs jours, du samedi 17 au mardi 20 février. Le code source du script est disponible à l'adresse suivante : https://github.com/cotonne/cnam/blob/master/nfe204/project/impl/tweet-reader.py

Le schéma suivant présente l'architecture mise en œuvre :



Avant d'insérer en base, il est nécessaire de définir le schéma utilisé.

V.2 Modélisation des tweets

Il est possible de s'appuyer sur des modélisations déjà effectuées [oscon-twitter-graph]. Ces modèles sont néanmoins plus compliqués par rapport à la solution à laquelle nous souhaitons arriver.

8 http://textblob.readthedocs.io/en/dev/

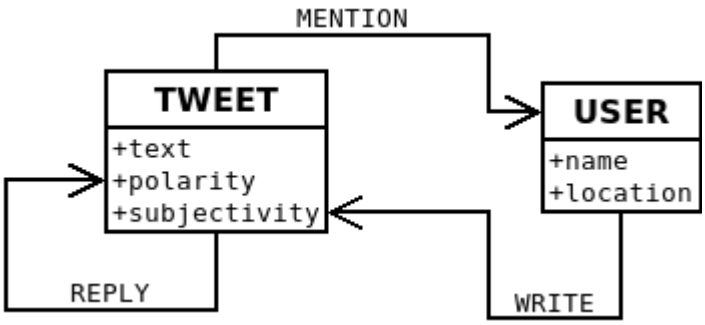


Illustration 12: Modèle de données

On retrouve ici les notions évoquées précédemment :

- Un utilisateur (**USER**) écrit (**WRITE**) un tweet (**TWEET**)
- Un tweet (**TWEET**) fait mention (MENTION) un autre utilisateur (**USER**)
- Un tweet (**TWEET**) peut être une réponse à un autre tweet.

La base contiendra donc deux types de nœuds (TWEET et USER) et deux relations (WRITE et MENTION)

V.3 Interaction avec Neo4J

Dans les représentations graphiques suivantes, générées à partir de Neo4J Browser, les « twittos » sont schématisés sous la forme de nœuds bleus, les tweets, sous la forme de nœuds rouges.

V.3.a) Twittos mentionné dans un tweet

Dans un premier temps, nous pouvons déterminer le nombre de tweets et d'utilisateurs capturés lors des campagnes de récupération des tweets :

```
MATCH (u:USER)
WITH count(u) as c
RETURN c
```

On trouve **34269** tweets et **21966** twittos.

Dans le paragraphe « I.3.c - Comment résoudre le problème avec un graphe », nous avons émis l'hypothèse que si un twitto était mentionné, cela signifiait qu'il intéressait une personne. Récupérons la liste des utilisateurs dont les tweets font mention de la personne de manière élogieuse :

```
MATCH (u:USER)<-[:MENTION]-(t:TWEET)<-[:WRITE]-(v:USER)
WHERE t.POLARITY > 0.2 and t.SUBJECTIVITY > 0.2
WITH u, count(t) as c, count(v) as d
WHERE c >=10 and d >= 5
RETURN u
```

Nous conservons les utilisateurs qui ont au moins 10 tweets positifs par au moins 5 personnes. Nous obtenons trois identifiants. Ces identifiants sont ceux de :

- 133663801 ⇒ @BFMTV
- 19438626 ⇒ @laurentwauquiez. Il faut se rappeler qu'une polémique est présente autour de Laurent Wauquiez depuis le samedi 17 février

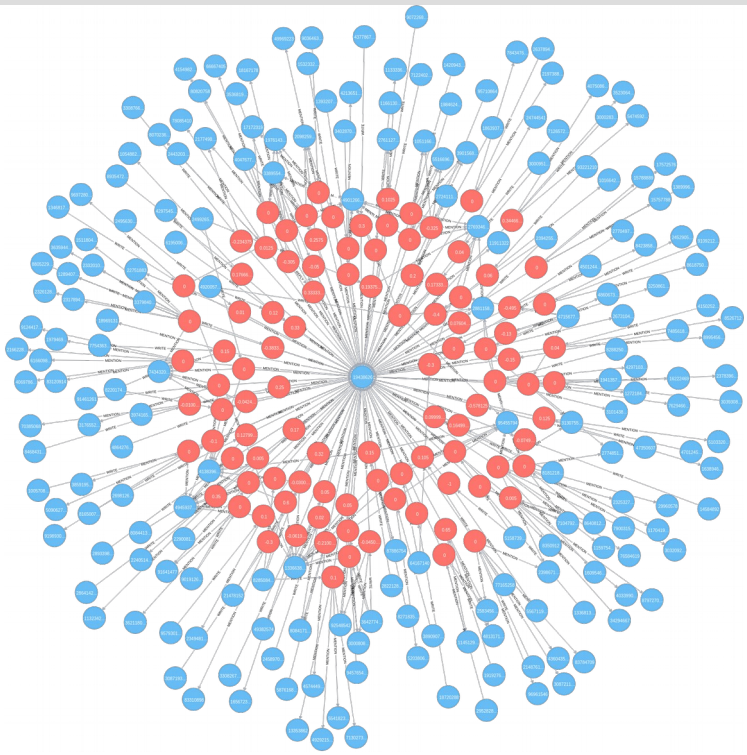
- 1172986574 ⇒ @martinfkde. C’est Martin Fourcade, le nouveau champion du monde.

Ainsi, on retrouve une chaîne de télévision connue. Le contenu des tweets montre que les gens s’intéressent aux propos qui y sont tenus. Elle a donc bien le rôle d’influenceur tel que nous l’avons défini. Les deux autres utilisateurs sont des personnes qui défraient la chronique ces derniers jours. Les utilisateurs de Twitter les félicitent … ou pas. De même, on peut aussi s’intéresser aux personnes détestées sur Internet. Il suffit pour cela d’inverser la polarité :

```
MATCH (u:USER)<-[:MENTION]-(t:TWEET)<-[:WRITE]-(v:USER)
WHERE t.POLARITY < -0.2 and t.SUBJECTIVITY > 0.2
WITH u, count(t) as c, count(v) as d
WHERE c >=10 and d >= 5
RETURN u
```

Un seul résultat remonte, c’est l’identifiant de Laurent Wauquiez. Affichons les différents tweets et twittos s’intéressant à Laurent Wauquiez pour bien comprendre ce paradoxe :

```
MATCH  (u:USER {ID:19438626})<-[r]-(t)-[s]-(v:USER)
RETURN *
```



Le nœud représentant Laurent Wauquiez est en bleu, au centre. Certaines personnes soutiennent Laurent Wauquiez Mais, d’autres font preuve d’ironie. On obtient ainsi un « faux positif » : il est classé comme un supporteur alors qu’il est critique. Par exemple, le tweet suivant (https://t.co/vNO4HBpqKc) a une polarité de 0,34 et une subjectivité de 0,24 :

C’est une des limites du système. Il a du mal à détecter l’ironie ou les sarcasmes. Une solution est de s’intéresser au nombre de détracteurs/promoteurs (une métrique inspirée de la notion de « Net Promoter Score ») :

```
MATCH (u:USER)<-[:MENTION]-(t1:TWEET)<-[:WRITE]-(v1:USER)
WHERE u.ID in [133663801,19438626,1172986574] and t1.POLARITY > 0.2 and
t1.SUBJECTIVITY > 0.2
WITH u, count(t1) as c1, count(v1) as d1
RETURN u.ID, d1 as nb, "Promoteurs" as type
UNION ALL MATCH (u:USER)<-[:MENTION]-(t2:TWEET)<-[:WRITE]-(v2:USER)
WHERE u.ID in [133663801,19438626,1172986574] and t2.POLARITY < -0.2 and
t2.SUBJECTIVITY > 0.2
WITH u, count(t2) as c2, count(v2) as d2
RETURN u.ID, d2 as nb, "Détracteurs" as type
```

Le résultat est le suivant :

u.ID	nb	type
133663801	15	"Promoteurs"
19438626	12	"Promoteurs"
1172986574	10	"Promoteurs"
133663801	6	"Détracteurs"
19438626	14	"Détracteurs"
1172986574	1	"Détracteurs"

En comparant la différence, il est possible de classer les individus du plus apprécié au moins apprécié : Martin Fourcade (9), BFMTV (9) puis Laurent Wauquiez (-2).

V.3.b) Réponse aux tweets

Intéressons-nous maintenant aux personnes répondant à des tweets. On peut se demander combien de réponses nous avons :

```
MATCH (t:TWEET)<-[:REPLY]-(o:TWEET)
RETURN count(*)
```

Nous avons 9289 en tout. Cependant, ce résultat cache un problème inhérent au fonctionnement de l'API de streaming. Seul le message publié est ajouté. Si le message n'était pas connu au préalable, il n'est pas remis à jour. Si on regarde les messages réellement bien constitués (avec une polarité et une subjectivité) :

```
MATCH (t:TWEET)<-[:REPLY]-(o:TWEET)
WHERE t.POLARITY is not null
RETURN count(*)
```

Nous obtenons 498 tweets, trop peu pour en tirer des conclusions. De plus, en analysant les tweets corrects, c'est-à-dire que nous voulons la liste des twittos pour lesquels les personnes ont répondu à un tweet de manière positive :

```
MATCH (i:USER)-[:WRITE]->(t:TWEET)<-[:REPLY]-(o:TWEET) <-[:WRITE]-(u:USER)
WHERE o.POLARITY > 0.2 and o.SUBJECTIVITY > 0.2
RETURN *
```

On constate que certaines personnes se répondent à elle-même pour créer des conversations plus longues. Voici un extrait du résultat obtenu (https://t.co/O2iew6aQjJ) :

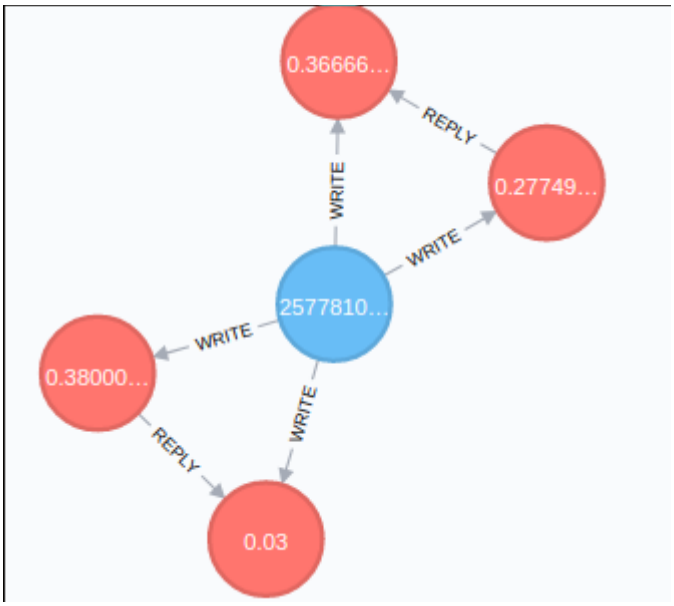


Illustration 13: Un twitto se répondant à lui-même

Il suffit donc d'éliminer ce type de schéma pour retrouver les « vrais » influenceurs. De plus, de fait que tous les messages n'ont pas été capturés, nous supprimons la polarité. La requête suivante résout ce problème et renvoie les groupes de personnes avec beaucoup de réponses positives :

```
MATCH (i:USER)-[w:WRITE]->(t:TWEET)<-[r:REPLY]-(o:TWEET) <-[:WRITE]-(u:USER)
WHERE i.ID <> u.ID
WITH t, count(r) as nb_reply
ORDER BY nb_reply DESC
LIMIT 20
WITH t
```

```
MATCH (x:USER)-[:WRITE]->(t:TWEET)<-[:REPLY]-(z:TWEET) <-[:WRITE]-(y:USER)
RETURN *
```

On retrouve de nouveau BFMTV qui, de par l'actualité, parle beaucoup de Laurent Wauquiez. On a donc une forte cohésion entre les deux. Le graphe suivant montre les liens M. Wauquiez et BFMTV qui sont représentés sous la forme de deux nœuds bleus entourés par des tweets (nœuds rouges).

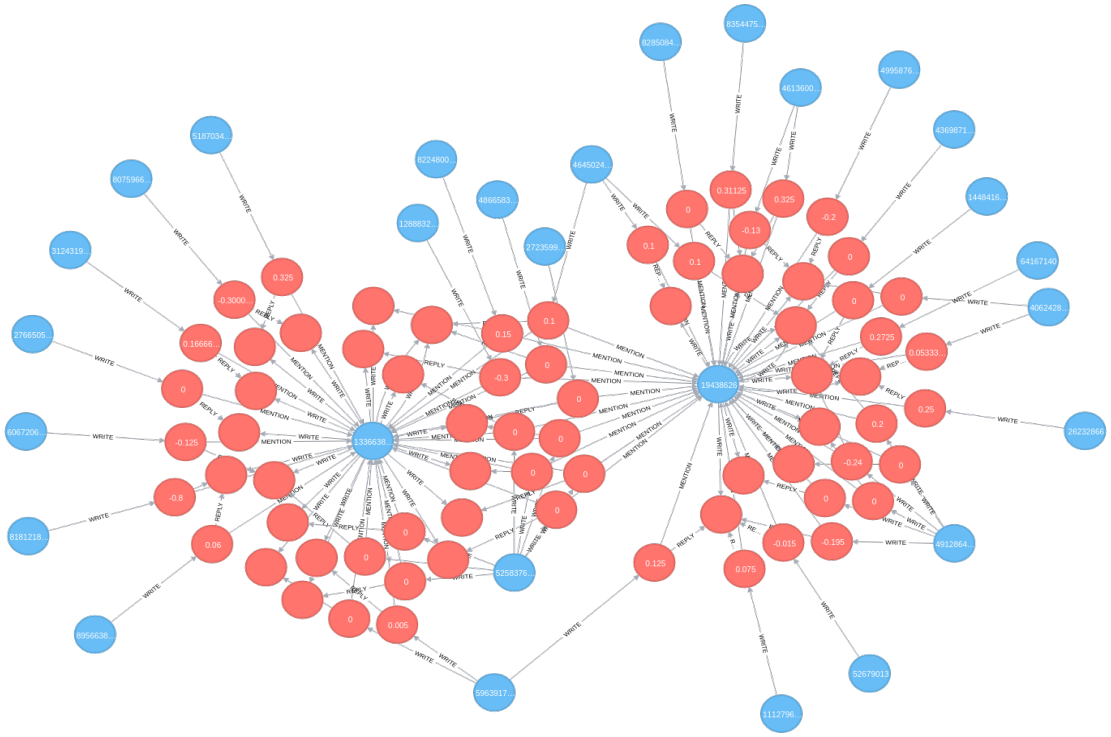


Illustration 14: @BFMTV - @LaurentWauquiez

On trouve un deuxième graphe présentant une structure similaire avec deux individus connus : « @TPMP » et « @Cyrilhanouna », une émission de télévision très influente et son présentateur vedette. Les sentiments des différents tweets sont globalement positifs. Cela confirme bien les différentes analyses qui donnent un rôle important à cette émission auprès du grand public, notamment d'un public assez jeune (et grand utilisateur des réseaux sociaux).

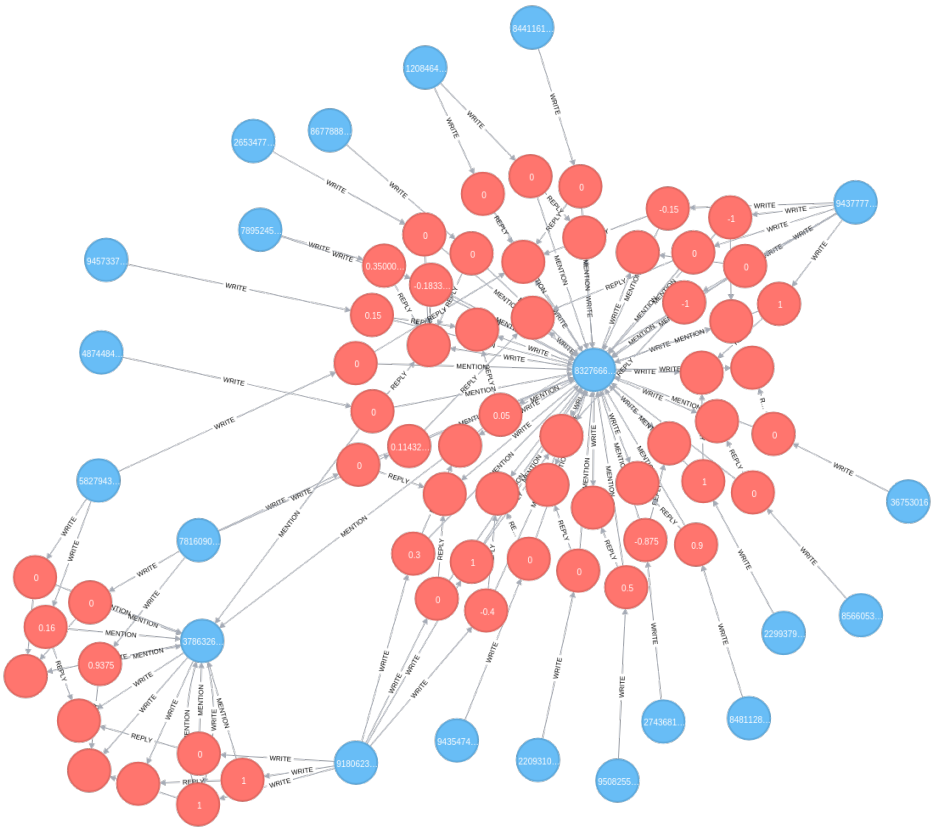


Illustration 15: TPMP - Hanouna

V.4 Problèmes rencontrés

Quelques problèmes ont été rencontrés lors de la réalisation de ce projet. Tout d'abord, la montée en compétence sur les différents sujets à demander un certain temps. Il a été nécessaire de comprendre neo4j, Twitter et le format des messages, le problème à résoudre, les librairies Python… Il faut noter que le projet a grandement été facilité par la simplicité des librairies et du langage Cypher qui a été réalisé via un programme de moins de 100 lignes.

De plus, la complexité inhérente à une langue vivante rend difficile une analyse de sentiment efficace. Des tweets subtils utilisant l'ironie ou le sarcasme sont facilement classés comme étant positif.

Un autre problème a été la présence de bugs qui ont été longs à découvrir. Une base de données sans schéma, bien que pratique, rend difficile la détection de bugs dus à des mauvaises insertions, incohérentes par rapport à la modélisation initialement prévue.

Enfin, le dernier problème est le temps nécessaire pour pouvoir avoir un jeu de données conséquent. Il est possible d'obtenir quelques milliers de tweets par jour pour la France. En comparaison des 6000 tweets par seconde au niveau mondiale, c'est bien peu.

VI. Limites et développements possibles

VI.1 Possibilités d'évolution

Ce projet m'a permis d'étudier et de mettre en œuvre neo4j sur un contexte intéressant, offrant de belles perspectives d'évolution.

D'autres pistes pourraient être suivies comme la possibilité d'étendre à d'autres réseaux. Par exemple, il est possible de récupérer les commentaires sur les vidéos Youtube. Souvent, un même utilisateur se sert de plusieurs plateformes de réseaux sociaux différents. La notion temporelle n'est pas non plus prise en compte. La rapidité à répondre à retweeter pourrait montrer un fort attachement à un twitto.

Un autre point est la mise en œuvre de la reprise sur incident pour le driver Bolt Python : en effet, le driver ne gère pas nativement le cluster HA. Cependant, le driver se connecte de manière efficace à un cluster causal neo4j.

VI.2 Neo4J, une base pour le Big Data ?

Le présent rapport ainsi que l'utilisation de Neo4j pour la détection des influenceurs nous aide à avoir une vision critique de la base de données graphe.

VI.2.a) Avantages

En résumé, Neo4J possède les qualités suivantes dans un contexte Big Data :

- Réponse à un besoin spécifique avec une solution adaptée et optimisée pour ce problème
- Haute-disponibilité via les différents modes de clustering (par réplication)
- Scalabilité (par réplication, grâce au cache sharding, ...)
- Volumétrie compatible avec le Big Data (plusieurs milliards d'éléments)
- Support des données non structurée
- Support des données graphes performantes comparativement aux autres solutions SQL et NoSQL

Lors du projet, l'absence de schéma a permis de faire des modifications en cours d'insertion. De nouvelles informations ont été ajoutées ou retirées sans impact sur le fonctionnement global du système.
Neo4J peut se déployer facilement dans une architecture de type cloud. A noter que pour une entreprise, il est nécessaire de payer une licence

VI.2.b) Inconvénients

Cependant, comme toute base NoSQL, il est prévu pour un cas d'usage en particulier. Ainsi, neo4j possède des limitations intrinsèques au format de représentation de données utilisées. Il n'est pas possible de partitionner les données, ce qui réduit les performances en haute-disponibilité.

En prenant en compte le but du cours NFE204, qui est l'étude des systèmes NoSQL dans un contexte de données massives, on peut donc en conclure que Neo4J est une solution NoSQL pour le BigData.

VII. Annexes

VII.1 Le langage Cypher

Ce paragraphe présente les requêtes Cypher qui serviront de base à la réalisation du projet. Il aidera notamment à comprendre les requêtes présentées dans la partie « V - Mise en œuvre du projet ». Cette partie se base sur le tutoriel « Movie Graph » accessible via « Neo4J Browser » en tapant « :play movie-graph » dans l'invité de commande.

Les requêtes ont la particularité de représenter le graphe sous forme d'ASCII Art.

Neo4J propose un aide-mémoire pour Cypher, disponible à l'adresse suivante : https://neo4j.com/docs/cypher-refcard/current/

VII.1.a) Insertion de données

Le tutoriel est basé sur des films, des acteurs et des rôles. On peut insérer un nouveau film et un acteur via la requête suivante :

```
: CREATE (TheMatrix:Movie {title:'The Matrix', released:1999})
: CREATE (Keanu:Person {name:'Keanu Reeves', born:1964})
```

Dans les deux requêtes, des variables ont été introduites (*TheMatrix* et *Keanu).* On peut ensuite les utiliser pour introduire une relation entre un film et un acteur qui sera automatiquement créé :

```
: CREATE (Keanu)-[:ACTED_IN {roles:['Neo']}]→(TheMatrix)
```

La relation qui lit le noeud *TheArtist* et *Keanu* est **ACTED_IN**. On peut noter que des propriétés peuvent être associées à la relation.

VII.1.b) Requêtes simples

Pour récupérer les films créés, la requête suivante peut être exécutée :

```
: MATCH (m:Movie) RETURN m LIMIT 10 ;
```

Cette requête pourrait être traduite en SQL de la façon suivante :

```
SELECT m.* FROM Movie m LIMIT 10 ;
```

On peut récupérer des nœuds selon certaines propriétés avec les requêtes suivantes, l'équivalent d'un ***WHERE*** en SQL. Un projection de certains attributs peut être fait dans la partie ***RETURN*** :

```
: MATCH (tom {name: "Tom Hanks"}) RETURN tom
: MATCH (nineties:Movie) WHERE nineties.released >= 1990
    AND nineties.released < 2000
    RETURN nineties.title
```

Il est aussi possible d'agréger les données. Si on veut le nombre d'acteurs nés par années :

```
: MATCH (m:Person) RETURN m.born, count(*)
```

VII.1.c) Requêtes par relation

Les relations sont aussi des objets qui peuvent être utilisées pour construire des requêtes. Par exemple, si on veut les films dans lesquels Tom Hanks a joué :

```
: MATCH (tom:Person {name: "Tom Hanks"})-[:ACTED_IN]->(tomHanksMovies)
    RETURN tom,tomHanksMovies
```

De même, on peut rechercher l'ensemble des personnes ayant joué dans les mêmes films que Tom Hanks

```
: MATCH (tom:Person {name:"Tom Hanks"})-[:ACTED_IN]->(m)<-[:ACTED_IN]-
(coActors) RETURN coActors.name
```

A noter que la variable m ne servant pas dans la requête, elle peut être supprimée.

VII.1.d) Requêtes avancées

Ce paragraphe présente quelques requêtes se servant au mieux de la puissance donnée par les graphes. Pour trouver qui se trouve à moins de 4 sauts de Kévin Bacon :

```
: MATCH (:Person {name:"Kevin Bacon"})-[*1..4]-(hollywood)
    RETURN DISTINCT hollywood
```

Comme indiqué précédemment, il existe de nombreux algorithmes de parcours de graphe pour trouver le chemin le plus court. Exemple :

```
: MATCH p=shortestPath(
  (bacon:Person {name:"Kevin Bacon"})-[*]-(meg:Person {name:"Meg Ryan"})
) RETURN p
```

Cette requête retourne les nœuds entre Kévin et Meg (à la fois les films et les acteurs).

Bibliographie

youtube: , YouTube : plus d'un milliard d'heures de vidéos vues quotidiennement, , http://www.cnetfrance.fr/news/youtube-plus-d-un-milliard-d-heures-de-videos-vues-quotidiennement-39849142.htm
youtube-money: , How to earn money from your videos, , http://www.cnetfrance.fr/news/youtube-plus-d-un-milliard-d-heures-de-videos-vues-quotidiennement-39849142.htm
youtube-ads: , Paid product placements and endorsements, , https://support.google.com/youtube/answer/154235?hl=en
wiki-social: , , , https://fr.wikipedia.org/wiki/R%C3%A9seau_social
oscon-twitter-graph: , ,

Liste des illustrations