INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

Projet intégrateur CLS / Big Data

BALBLANC Nathan,

CAZANAVE Valérian,

DELORME Gautier,

PROUVOST Chloé,

VIOZELANGE Quentin

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES

Table des matières

Contexte du projet

L'objectif de ce projet a été d'utiliser à bon escient la technologie Big Data afin d'être capable de traiter une quantité très importante d'informations. Les données manipulées sont issues d'un système d'échanges de messages (AIS) qui permet à des navires entre autres, de connaître l'identité, le statut, la position et la route des navires se situant dans les zones de navigation (océans, mers, fleuves). Les données AIS sont émises par les navires et collectées ensuite par des stations terrestres ou des systèmes satellitaires.

Mises à notre disposition par la société **CLS**, nous devions exploiter les données maritimes renvoyant les indications suivantes :

- probabilité de présence d'un bateau sur le point d'un globe
- direction moyenne de navigation d'un bateau sur le point d'un globe
- vitesse moyenne d'un bateau sur le point d'un globe

Pour ce qui est du travail réalisé, nous avons seulement utilisé les données relatives à la *probabilité de présence* d'un bateau sur la carte. Grâce à cela, nous avons pu délivrer un service de génération du meilleur chemin maritime entre deux coordonnées. Nous allons donc vous présenter plus en détails comment nous sommes arrivés à un tel résultat et de quels outils nous nous sommes servis.

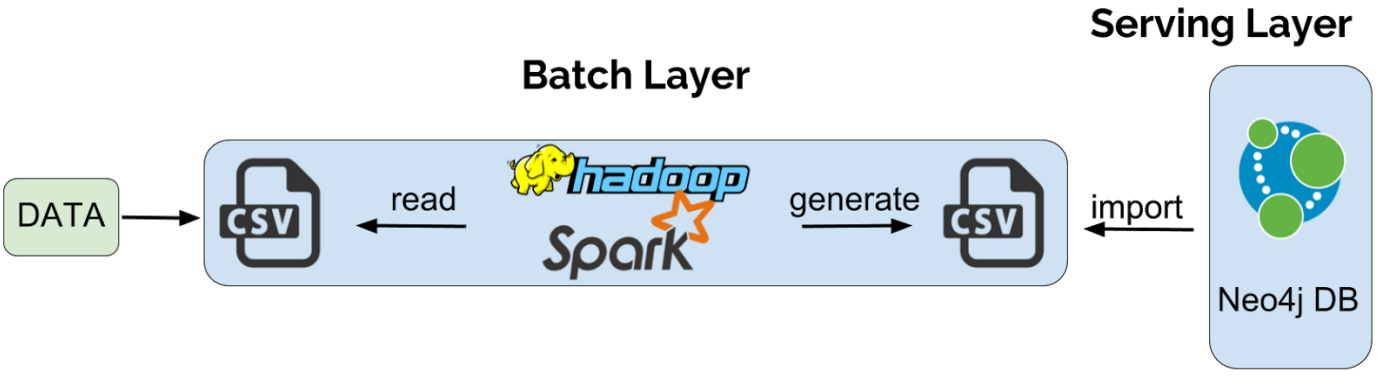
I. Interprétation et transformation des données maritimes

La première étape de ce projet a été de récupérer les données communiquées par CLS et d'analyser leur structure afin de savoir comment on allait pouvoir les exploiter. Nous avons commencé à traiter le fichier de précision 5 (⅕), afin d'économiser en temps de calcul pour le début quitte à perdre en précision.

Après une première analyse, nous avons décidé d'importer ces données sur la base de données **Neo4j**, à travers un programme écrit en Python et exploitant la librairie "py2neo". Cependant, suite à ce premier essai nous nous sommes vite aperçus des contraintes liées à la taille des données que nous manipulions. En ne prenant que les points de la carte avec une probabilité de présence supérieure à 0.5, ce qui ne représente que **0.79 %** des données totales, notre importation prenait 17 minutes… Nous avons donc commencé à mettre en place une architecture propre au Big Data.

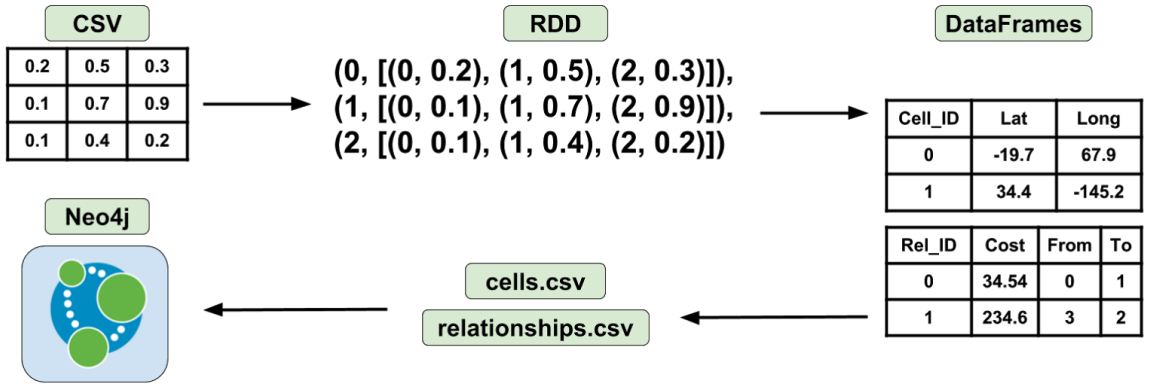
A. Couche Batch processing

A travers cette couche de batch processing, nous avons donc reproduit l'importation des données du fichier CSV dans la base de données Neo4j. Pour cela nous avons utilisé les solutions **Spark** et **Hadoop**. Voici ci-dessous l'architecture employée :



L'objectif est donc, grâce à Spark maintenant de lire et traiter **toutes les données** du fichier CSV. On détermine alors une structure d'un noeud du graphe que l'on va générer, ainsi qu'une structure d'une relation reliant deux noeuds entre eux. Puis en parcourant le fichier, on modélise chaque donnée en noeud du graphe, et on établit la liste des relations existantes dans un second temps.

Ensuite deux fichiers CSV sont générés : un recensant tous les noeuds découverts, l'autre recensant les relations.



Voici ci-dessus l'évolution des données que l'on reçoit en entrée jusqu'à la phase d'importation dans Neo4j. Le bloc **RDD** représente le mapping en mémoire qui est effectué par Spark pour accélérer son temps de traitement, et **DataFrames** est une transformation du RDD afin de faciliter la génération des fichiers CSV. On aperçoit donc les deux structures qui seront employées dans la base de données : *Cell* et *Rel*. La première correspondant à un noeud du graphe possède un identifiant, et deux caractéristiques : sa latitude et sa longitude. Ces deux paramètres sont calculés à partir de leurs index dans le fichier d'origine (ligne, colonne). En ce qui concerne les relations, elles possèdent un identifiant, un coût, et l'identifiant du noeud de départ et du noeud d'arrivée. Ce qu'on appelle le coût en fait, c'est ce qui représente l'inverse de la probabilité de présence d'un bateau. Par conséquent, on peut dire que les noeuds figurant dans la colonne "To" ont un coût associé à la valeur renseignée dans la colonne "Cost" : plus cette probabilité est élevée, plus le coût sera faible.

"Il existe une probabilité égale à x pour se rendre d'un point A à un point B, ce qui veut dire que la probabilité en elle-même de se trouver sur B est de x."

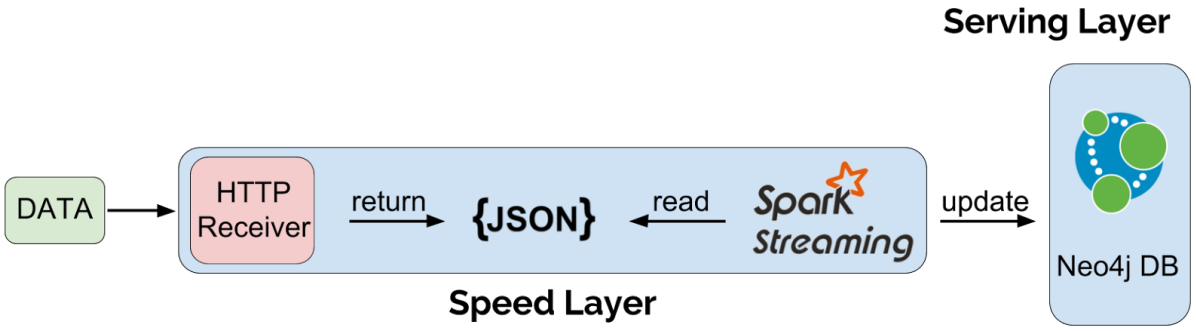
1. *Performances*

Une fois après avoir bâti cette architecture, nous avons pu tester son efficacité et améliorer de manière itérative sa vitesse de traitement tout en essayant les différents jeux de données fournis. Voici alors les résultats obtenus du temps mis pour générer la base de données en fonction de l'échelle de précision :

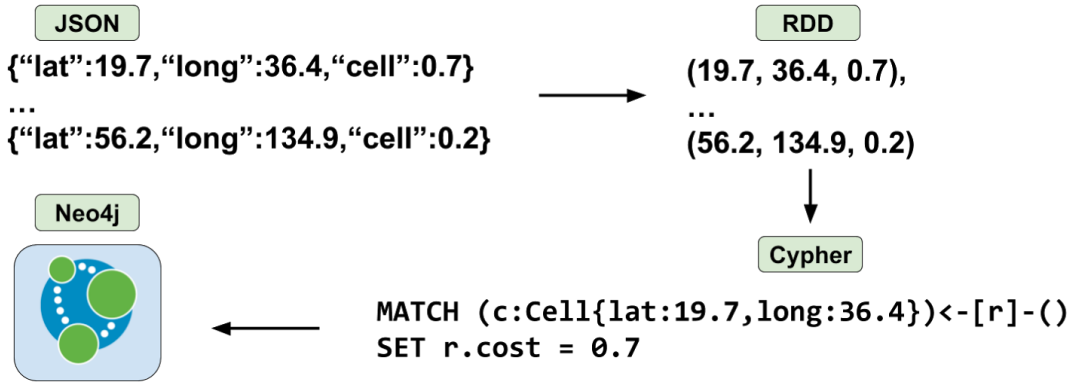
échelle	temps (minutes)
1/5	1
1/40	30

B. Couche Streaming

En parallèle, nous avons voulu simuler une génération de données type AIS comme celles que l'on pourrait recevoir pour pouvoir modéliser une couche de streaming. On imagine alors que l'on reçoit de manière directe des coordonnées de bateau, qu'on les traite et qu'on les intègre à notre base de données sur Neo4j afin de mettre à jour nos calculs de plus courts chemins :

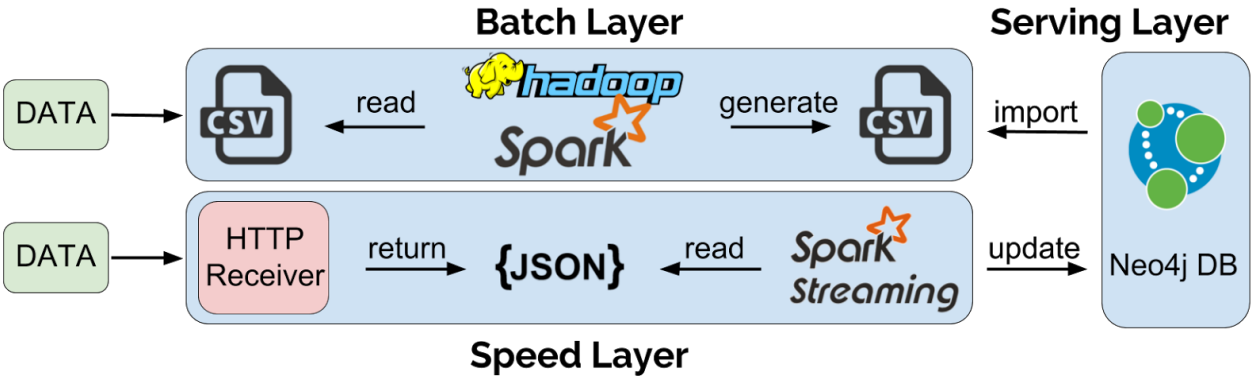


Pour ce faire, on a modélisé ce "flux de données" à l'aide d'un **HTTP Receiver**. En fait, grâce à ce composant on crée un port d'écoute sur lequel on peut poster des coordonnées au format JSON de manière continue. Ensuite, grâce à un parseur on décortique le flux entrant de données et Spark récupère les coordonnées des noeuds à mettre à jour. Par la suite, on procède à la mise à jour de la base de données avec une valeur du coût actualisée pour chaque noeud traité :



C. Architecture Lambda

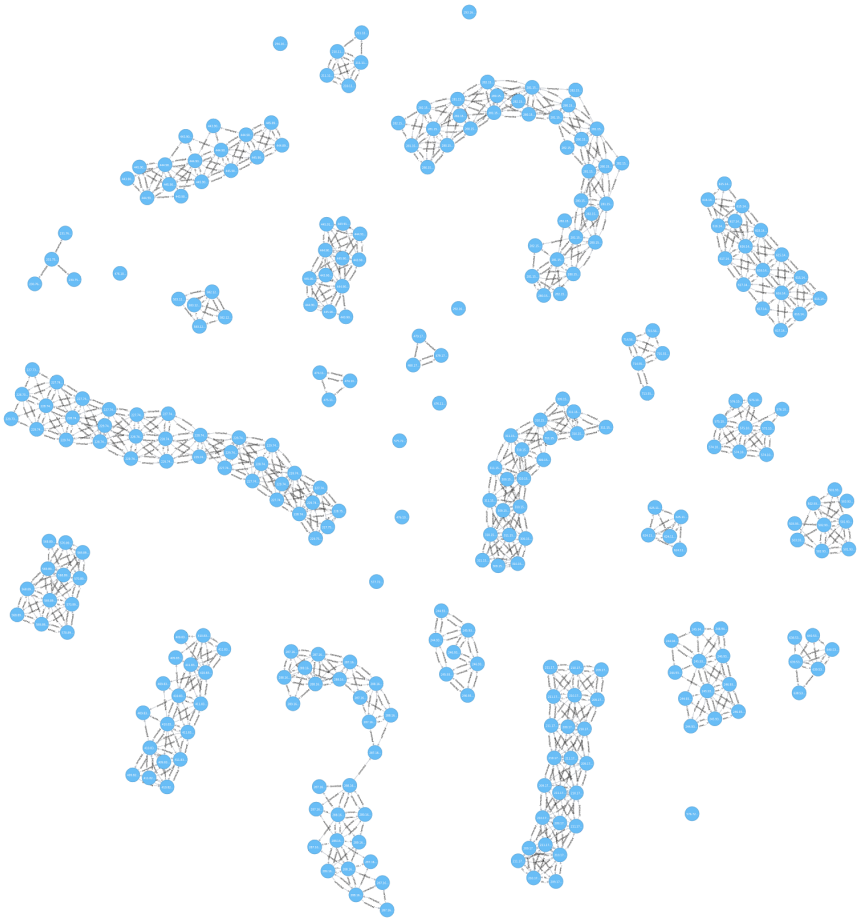
Maintenant, notre application est dotée de deux couches. La combinaison de ces deux couches qui permet en même temps de traiter des gros volumes de fichiers à la demande (batch), et des flux continus de données (streaming) s'appelle une architecture "Lambda". Voici sa structure :



II. Utilisation des données transformées pour le calcul d'itinéraires

A. Création d'une base de données avec Neo4j

Nous avons fait le choix d'utiliser une base de données orientée graphes, et plus particulièrement Neo4j, car elle intègre déjà des algorithmes de plus court chemin. De plus, la problématique de routes à modéliser est beaucoup plus adaptée dans une base de données orientée graphes plutôt qu'une base de données relationnelles, car il y a une forte connectivité entre les données. Pour illustrer notre propos, voici ci-dessous un aperçu d'une première base de données qui comporte plus de 7000 noeuds soit moins de 1% de nos données totales et déjà pas loin de 50 000 relations ; ce qui constitue une entrave majeure à la performance de notre système si on utilise une base de données relationnelles qui n'est pas faite pour supporter une telle dépendance entre ses tables. Elle supporterait encore moins le "passage à l'échelle", et y serait très sensible.



B. Calcul de plus court chemin

La deuxième phase du travail maintenant est d'utiliser la base de données (graphe) que l'on a modélisée pour appliquer un calcul de plus court chemin entre deux noeuds. Comme expliqué précédemment, l'utilisation de Neo4j nous simplifie la vie puisqu'une fonction de plus court chemin est déjà intégrée dans la base de données ; et nous n'avons plus qu'à nous en servir pour générer nos routes maritimes. On lui passe en paramètre le noeud de départ et le noeud d'arrivée, et on exécute une requête CYPHER (langage de requêtage de Neo4j) :

```
"""
MATCH (start:Cell{latitude:{from_latitude}, longitude:{from_longitude}})
WITH start
MATCH (end:Cell{latitude:{to_latitude}, longitude:{to_longitude}})
CALL apoc.algo.aStar(start, end, 'LINKED>', 'cost','lat','long') YIELD path
RETURN path
"""
```

Pour un souci d'optimisation encore, nous avons fait le choix d'appliquer l'algorithme **aStar** qui est le plus rapide de tous les algorithmes de plus courts chemins.

III. Exportation et exploitation des résultats obtenus

Maintenant que nous avons pu générer nos chemins maritimes à travers une base de données orientée graphes, et en s'appuyant sur les données communiquées par CLS il nous faut être capable de les exploiter.

A. Export des résultats en KML

Tout d'abord, nous avons développé un service pour exporter nos données au format KML et utiliser l'outil "Maps" afin d'afficher sur **Google Maps** un tracé visuel de notre itinéraire. Voici les résultats obtenus pour certains tracés :

Départ : *Rotterdam*

Arrivée : *Kuala Lumpur*

Précision : *1/40*

Temps de calcul : *2 s.*

Parcours : *Main, Danube, Canal de Suez*

Départ : *San Francisco*

Arrivée : *Caracas*

Précision : *1/40*

Temps de calcul : *200 ms.*

Parcours : *Canal de Panama*

Départ : *Brest*

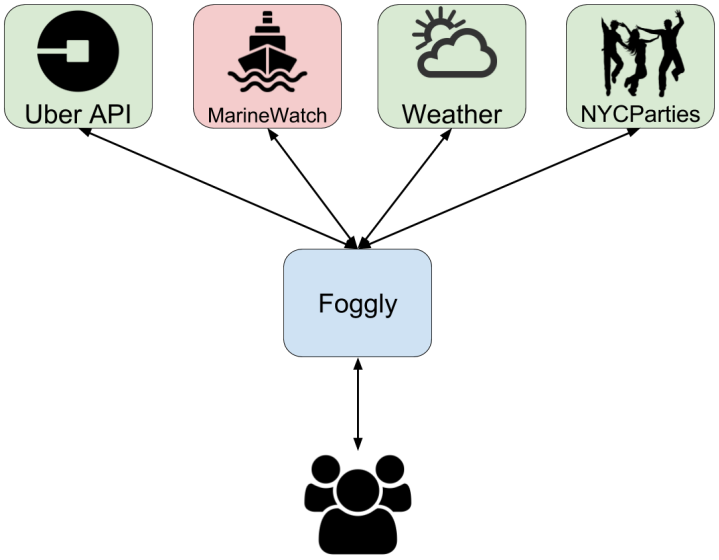
Arrivée : *Rio de Janeiro*

Précision : *1/40*

Temps de calcul : *250 ms.*

Ces images proviennent d'un calcul de plus court chemin effectué avec la précision la plus haute (1/40). On s'apercevait quand on utilisait la plus petite précision que, sur des couloirs maritimes étroits (canaux, détroits) ou littoraux, le tracé passait parfois sur des bandes de terre, ce qui est normal au vue de la quantité d'informations traitée en moins.

B. Exposition de ces résultats au travers d'une API

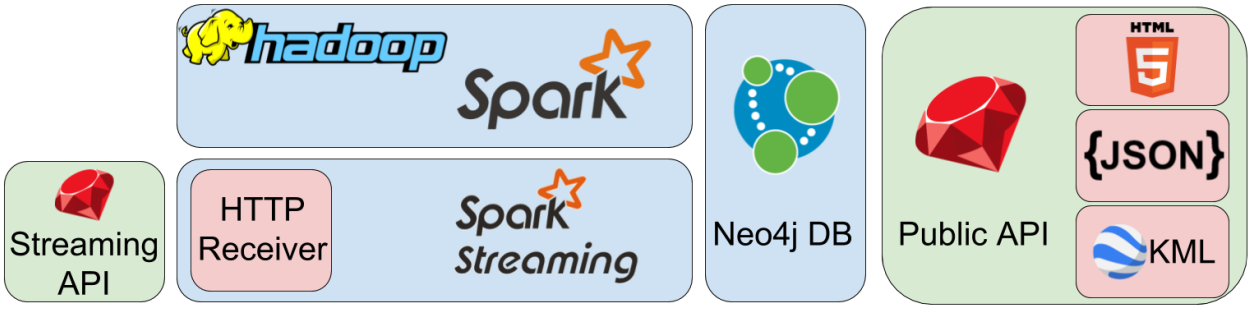


Afin de compléter ce projet, et de pouvoir l'intégrer à notre projet SOA/IL nommé **Foggly** nous avons donc repris le service KML précédemment cité et l'avons intégré dans une API Web écrite en Ruby. Cette API calcule ainsi les plus courts chemins maritimes demandés par l'utilisateur et expose ses résultats sous plusieurs formats : HTML, KML, JSON.

Les "endpoints" résultant d'une demande de chemin entre deux coordonnées sont par la suite accessibles via la plateforme Web Foggly sous les différents formats. Voici un exemple d'endpoint :

```
# http://localhost:4567/route?from=39.425,6.825&to=6.225,103.050
```

Au final, voici ce que nous obtenons comme architecture pour notre service baptisé **Marinewatch** :



IV. Technologies employées

Conclusion

Ce projet nous a permis de mettre en pratique les notions de Big Data acquises durant ce semestre et sortir d'un cadre minimaliste à base d'exemples. Il a fallu appréhender les différentes technologies qui se présentaient à nous et nous les approprier par rapport à notre cas de figure (données maritimes). Ceci a demandé un certain temps de réflexion et de la patience. Cependant nous avons pu exploiter toute une série de données, à savoir tous les fichiers de densité transmis par CLS (toutes échelles). De cette base là, nous avons été en mesure d'effectuer deux types de traitement de Big Data : à la volée et en continu, ce qui nous a permis d'établir une architecture "Lambda" basique. Grâce à cette technologie, il nous a été possible de traiter des millions de lignes de code et de satisfaire des requêtes en un temps raisonnable. Par la suite, un bon choix au niveau de la base de données nous a permis de facilement déployer le calcul de plus courts chemins, une fois après avoir stocké ces données.

Dans les améliorations futures de Marinewatch, nous pourrions compléter notre travail en incluant les deux autres types de données (direction de navigation, vitesse) à notre calcul de meilleurs chemins.

Annexes

Code source Marinewatch : https://github.com/fogglyorg/marinewatch