Université de technologie de Compiègne

NF26 - Data Warehouse et outils décisionnels

Compte-rendu d'un travail pratique
Cassandra - Base de données NoSQL

Elliot BARTHOLME

4 décembre 2017

1 Introduction

Apache Cassandra est un système de gestion de base de données open source qui appartient à la famille NoSQL [1]. Conçu pour gérer un nombre important de données distribuées sur plusieurs serveurs, il assure disponibilité, cohérence et tolérance aux pannes grâce à des réplications asynchrones qui malgré tout ne compromettent pas la rapidité et l'efficacité des traitements et des requêtes.

Les bases de données Cassandra ont une architecture hybride orientée colonnes [2] et clé-valeur [3] avec une distribution par clé primaire.
Une clé primaire est composée d'une clé de *partitionnement* qui détermine la répartition des données sur les noeuds de Cassandra (de manière ordonnée ou aléatoire), et une clé de *clustering* qui détermine le classement et l'organisation (tri) de ces données au sein d'une partition.

Dans l'UV NF26, Data Warehouse et Outils Décisionnels, nous avons été introduits à l'utilisation de ce type de base de données en utilisant un *dataset* contenant des trajets de taxi à Porto (Portugal) pendant les années 2013 et 2014. Initialement destiné à un challenge de prédiction des trajectoires, ces données ont été utilisées pour découvrir Cassandra, établir un modèle de conception, puis implémenter ainsi qu'alimenter une base de données *en masse* avec le langage CQL (Cassandra Query Language) et Python.

Ce document présentera donc toutes ces étapes, ainsi que l'exploitation à des fins d'analyse de données.

2 Données

Un fichier *train.csv* a été mis à disposition, contenant *1 710 671* lignes correspondant chacune à un trajet de taxi. Neuf composantes/attributs caractérisent chacun de ces trajets dans le fichier source :

- TRIP_ID (*string*) : identifiant unique du trajet
- CALL_TYPE (*char*) : détermine la manière dont a été demandé le taxi :
  - 'A' si le service a été dispatché depuis le centre de la compagnie de taxis
  - 'B' si le taxi a été demandé directement à un conducteur depuis un stand spécial
  - 'C' autrement (par exemple demandé au hasard dans une rue quelconque)
- ORIGIN_CALL (*integer*) : numéro de téléphone du client ayant demandé le taxi. Cet attribut n'est présent que pour un CALL_TYPE égal à 'A', sinon il est supposé nul.
- ORIGIN_STAND (*integer*) : identifiant du stand auquel a été demandé le taxi. Cet attribut n'est présent que pour un CALL_TYPE égal à 'B', sinon il est supposé nul.
- TAXI_ID (*integer*) : identifiant unique du conducteur de taxi ayant effectué le trajet.
- TIMESTAMP (*integer*) : *timestamp* au format Unix identifiant le moment du début du trajet.
- DAYTYPE (*char*) : détermine le type de jour du trajet, avec trois valeurs possibles :
  - 'B' si le trajet a lieu un jour férié ou spécial
  - 'C' si le trajet à eu lieu le jour précédent un jour 'B'
  - 'A' autrement, pour un jour standard (jour de semaine, week-end...)
- MISSING_DATA (*boolean*) : vaut FALSE si les coordonnées GPS (voir suivant) sont complètes, TRUE si certaines n'ont pas pu être récoltées (il manque donc des données)
- POLYLINE (*char*) : contient une liste de points aux coordonnées GPS [longitude, latitude] récoltés toutes les 15 secondes lors du trajet.

3 Conception

Tout comme dans la première partie de l'UV, toute implémentation d'un datawarehouse ou d'un modèle de données doit être précédée d'une phase de conception. Durant cette phase on s'intéresse à l'identification des différentes entités présentes, que l'on représente chacune sous forme de dimension. Dans un modèle relationnel classique, chacune est divisée jusqu'au grain souhaité pour obtenir des dimensions organisées en hiérarchies d'entités atomiques.

1. Les bases NoSQL (not only SQL) s'écartent du paradigme relationnel en permettant la distribution des données et une échelonnabilité (*scalability*) qui s'adapte très bien aux grands volumes de données et aux analyses.
2. Stockage des données par colonne, nombre de colonnes évolutif et propre à chaque enregistrement
3. Absence de structure et de typage permettant une forte évolutivité des données

Une des premières découvertes liées aux entrepôts de données et notamment au datamart (magasin de données spécialisées qui précède les outils d'analyse et de décisionnel) est la non-application de ce principe afin d'obtenir des modèles dénormalisés avec des hiérarchies *concaténées* et regroupées autour d'une table de faits (modèle en étoile, *star schema*). Il permet de se passer de jointures souvent coûteuses, mais au prix d'une redondance des informations, ce qui contrairement au cas des outils transactionnels d'entreprise, n'est pas réellement gênant dans les outils d'analyse.

Dans une base de données NoSQL telle que dans Cassandra, c'est ce principe qui est l'axe majeur de la conception. On ne définit pas les tables selon des dimensions normalisées au maximum comme dans un SGBDR classique où l'on aurait stocké les données par entité (par exemple une table *conducteur de taxi*, *temps*, *géographie*...) car les opérations de jointure sont impossibles. Le but est vraiment d'implémenter le modèle dans une approche orientée requête (*query oriented design*), c'est-à-dire où chaque table doit répondre à un type de requête particulier (tout comme un datamart répond aux besoins d'un groupe d'utilisateurs ou d'un type d'affaires en particulier) et possède toutes les informations nécessaires sur les autres dimensions.

Ce n'est pas un problème d'avoir de nombreuses tables avec des données redondantes et beaucoup de colonnes (jusqu'à 2 milliards dans Cassandra), car c'est ce pour quoi ce modèle de stockage a été conçu. Grâce à son stockage réparti, Cassandra partitionne les données grâce à la première partie de la clé primaire (voir paragraphe 4) sur différents noeuds physiques.

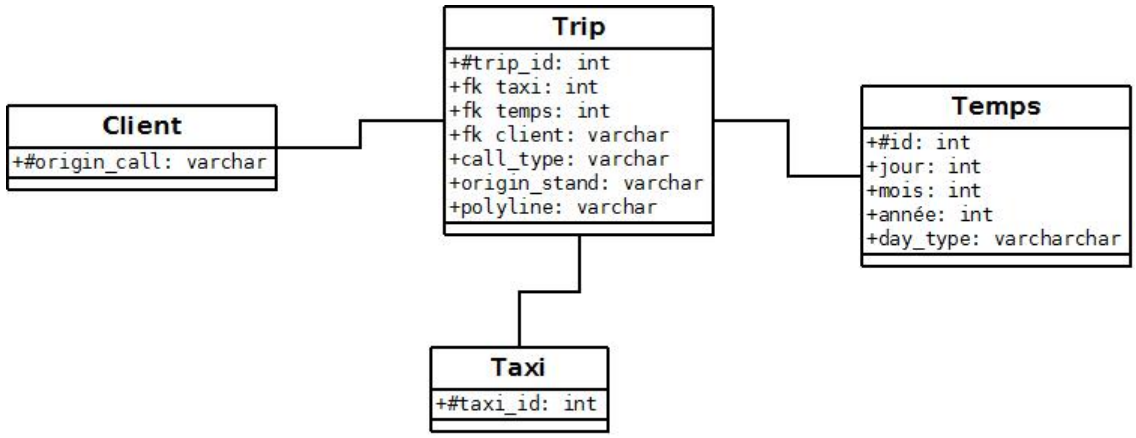

FIGURE 1 – MCD défini à partir de l'analyse des données fournies

4 Implémentation

Un keyspace[4] a été créé dans Cassandra pour chaque utilisateur. Ensuite, au sein de ce keyspace, l'objectif était de créer des familles de colonnes, que l'on appelle aussi tables, afin de modéliser proprement les données fournies par le fichier *csv* source.

Il faut donc choisir, pour chaque table, la ou les colonne(s) sur laquelle ou lesquelles on partitionne (on distribue sur les noeuds) les données : *partition key*, et les colonnes sur lesquelles on trie les données dans chaque partition : *clustering key*.

Il est important de noter que contrairement aux bases SQL, on ne peut pas faire de restriction sur des attributs autres que les clés, et qu'il faut obligatoirement faire des restrictions dans leur ordre hiérarchique : si l'on veut restreindre sur une clé de clustering, il faut alors restreindre auparavant sur chaque clé de partitionnement et sur les clés de clustering précédentes. Cassandra gère des données distribuées et prévient par conséquent de l'exécution de requêtes inefficaces dont les performances seraient aléatoires ; l'ordre et le choix des clés est donc

4. Un keyspace, semblable au schéma dans un SGBDR, est un espace de nom qui définit la stratégie de réplication sur les noeuds de Cassandra et qui contient les familles de colonnes (tables)

primordial.

4.1 Colonnes et nettoyage

On profite de la création manuelle des tables pour ajouter ou retirer de l'information aux données qui nous sont présentées. En effet, à des fins d'analyse, il ne nous a pas semblé par exemple très utile de stocker tout le chemin GPS (polyline, parfois très long) ou encore le timestamp au format Unix. Nous avons donc effectué tous les traitements suivants :

- Transformation du timestamp en année, mois, jour, heure et minute.
- Simplification de l'attribut polyline en deux colonnes *starting_point* et *ending_point* afin d'avoir le point de départ et d'arrivée du trajet uniquement. Un arrondi à deux chiffres est aussi fait pour pouvoir utiliser les points comme des emplacements plus larges (utile pour des tentatives de *clustering* par quartier et des sélections par point de départ/arrivée).
- Calcul de la valeur de distance du trajet grâce à une fonction Python prenant des longitudes et latitudes de départ et d'arrivée en argument. Une attention particulière a été portée pour certaines données manquantes lorsqu'il n'y a qu'une seule ou aucune coordonnées GPS pour le trajet.
- Calcul du *day_type* car cette information est manquante dans les données (on ne trouve que la valeur 'A'). Pour cela, les jours spéciaux sont stockés dans une liste de jours fériés sur les deux années. Ainsi, après comparaison avec cette liste, lorsqu'un trajet est inséré dans la base, son *day_type* correct est ajouté en même temps.
- Calcul de la durée du trajet *duration* sur la base qu'un point GPS est récolté toutes les 15 secondes pendant chaque trajet.
- Insertion des colonnes *origin_stand* et *origin_call* uniquement si le type de trajet (*call_type*) correspond à une existence de cet attribut

Nous avons donc choisi selon le modèle de conception de définir chaque table avec les colonnes suivantes, avec les types associés :

- *double trip_id* : identifiant trajet
- *double taxi_id* : identifiant taxi
- *int year* : année
- *int month* : mois
- *int day* : jour
- *int hour* : heure (départ)
- *int min* : minute (départ)
- *int day_type* : type de jour (cf. §2, DAYTYPE)
- *varchar starting_point* : premières coordonnées de polyline (cf. §2, POLYLINE)
- *varchar ending_point* : dernières coordonnées de polyline (cf. §2, POLYLINE)
- *double dist* : calcul de distance entre starting_point et ending_point
- *varchar call_type* : type de la demande de taxi (cf. §2, CALL_TYPE)
- *varchar origin_stand* : identifiant du stand de taxi, présent si $call\_type = B$ (cf. §2, ORIGIN_STAND)
- *varchar origin_call* : téléphone du client si $call\_type = A$ (cf. §2, ORIGIN_CALL)
- *varchar duration* : durée du trajet en secondes

4.2 Tables

Il faut noter que les lignes contenant des valeurs nulles pour les clés ne sont bien évidemment pas insérées. Pour ces tables (par exemple *trip_origin_stands, trip_origin_calls...*) il y aura donc beaucoup moins de lignes(Un peu plus de respectivement 175000 et 360000 lignes chacune), l'idée étant de pouvoir réaliser de petites statistiques pour chaque catégorie avec par exemple les stands les plus populaires ou encore le numéro de téléphone le plus utilisé pour commander un taxi...

Les tables que nous avons finalement créées sont les suivantes :

Nom de la table	Clé(s) de partitionnement	Clé(s) de clustering
trip_taxis	taxi_id	trip_id
trip_distances	dist	trip_id
trip_call_types	call_type	trip_id
trip_origin_stands	origin_stand	trip_id
trip_origin_calls	origin_call	trip_id
trip_years	year	month, day, hour, min, trip_id
trip_months	month	year, day, hour, min, trip_id
trip_days	day, month, year	hour, min, trip_id
trip_hours	hour	year, month, day, hour, min, trip_id
trip_departures	starting_point	ending_point, trip_id
trip_arrivals	ending_point	starting_point, trip_id
trip_duration	duration	trip_id
trip_day_types	day_type	trip_id

TABLE 1 – Tables implémentées dans Cassandra

L'objectif lors de l'implémentation est d'obtenir des tables le plus orientées possible vers les requêtes qui y seront exécutées car Cassandra n'est pas fait pour pouvoir filtrer/restreindre sur des colonnes quelconques en raison de la distribution des données qui rend beaucoup trop aléatoire les performances. Une table donnée doit, comme cela a été vu, pouvoir répondre à des questions et des problématiques précises uniquement.

La table *trip_hours* permettra par exemple de voir à quelles heures de la journée les taxis sont le plus demandés, la table *trip_call_types* de savoir par quel moyen les taxis sont le plus sollicités, et ainsi de suite...

Les clés de *clustering* ordonnées permettent de définir comment seront affichées ces données ligne par ligne : dans la table *trip_days* donnera ainsi les trajets par jour, mois et année (donc par date), et ensuite par heure puis minute et finalement selon leur identifiant. On pourra alors répondre rapidement à la question "Quels trajets ont été effectués le 4 avril 2014 à 11h ?".

5 Analyse

L'intérêt principal de ce travail pratique est de découvrir les problématiques qui découlent de l'utilisation de gros volumes de données, mais outre celle du stockage et de l'alimentation, celles liées à leur analyse ne peuvent être négligées. Les masses de données comme celles en temps réels, les données de log ou autres... sont difficile à étudier à cause de la quantité d'information et du bruit généré, ainsi que de leur volatilité ; on ne les aborde donc pas de la même manière.

Le CQL n'est pas aussi permissif que le SQL car les données sont distribuées et non stockées ligne par ligne ; il est par exemple souvent beaucoup plus rentable d'ajouter une colonne d'index qui permet de compter que d'effectuer des requêtes *count* hasardeuses.
Les opérations de dénombrement sur un aussi gros volume de données ajoutées à la concurrence d'accès aux ressources du serveur des différents étudiants ont été très souvent compliquées. Les requêtes infructueuses soldées par un *timeout* de réponse ont été nombreuses et l'insertion fastidieuse. Les clauses de restriction sont très différentes, les opérations de groupage avec *group by* n'ont été permises qu'à partir de la version 3.10... on recommande souvent à l'utilisateur de créer lui-même ses fonctions d'agrégation avec un langage de programmation, comme nous le faisons parfois avec Python.

Deux types d'analyse principales seront développées : Une méthode de classification automatique non supervisée ou *clustering*, et des analyses à base de requêtes pour obtenir des statistiques sur les familles de colonnes comme par exemple des *Top 5* par catégorie.

5.1 K-means

L'algorithme des K-means (K-moyennes) est un algorithme de classification automatique non supervisée, c'est-à-dire de partitionnement de données. Considérant des données, l'objectif est de les séparer en K groupes ou classes souvent appelés *cluster* de telle manière à minimiser une fonction. En pratique, le critère que l'on minimise par itérations successives est l'inertie intra-classe c'est-à-dire que l'on cherche à obtenir des groupes les

plus concentrés possibles et les plus éloignés les uns des autres au sens de la distance euclidienne.

Il consiste à choisir initialement k centres de gravité représentant les k *cluster* et à affecter chaque nouveau point au groupe dont le centre de gravité est le plus proche de lui. Chaque centre de gravité est par la suite recalculé (repositionné) selon l'ajout où le retrait des points. Le moment de l'algorithme où est effectué ce repositionnement conduit à plusieurs variantes. L'algorithme s'arrête lors de la convergence (pas de réaffectation des points).

Dans notre cas, nous avons développé une fonction en Python (voir fichier *analysis.py*) qui choisira de manière aléatoire k points dans la table afin de définir les k centres de gravité initiaux. Ensuite, on commence à itérer dans la table et chaque point est affecté au groupe dont le centre de gravité est le plus proche en utilisant la fonction de calcul de la distance avec les coordonnées GPS. L'algorithme fonctionne pour un point donné au format GPS, il faut donc l'exécuter sur les points de départ et d'arrivées des taxis.

Une autre version développée par un collègue (Alexis Durocher) se charge elle de considérer les trajets en entier et donc de faire des calculs de distances 4d entre départs et arrivées de deux trajets. Nous avons décidé d'afficher les résultats obtenus avec, ceux-ci étants meilleurs.

On peut donc en sortie afficher les coordonnées des k-moyennes (centre de gravité de chaque *cluster*) et leur poids : le nombre de trajets de chaque classe. Il est alors possible d'afficher sur une carte interactive comme celle de Google Maps les coordonnées des centres des *cluster* et l'on aurait même pu afficher les nuages points pour voir une représentation en deux dimensions des trajets associés à chacun.

Voici une représentation pour un k-means sur les trajets de taxi

FIGURE 2 – Positionnement des trajets moyens pour chaque cluster avec un k égal à 4

On voit qu'un des *cluster* correspond à un trajet qui va du centre à l'aéroport de Porto (rouge). C'est un résultat assez logique et prévisible. En revanche un trajet bien étonnant qui s'éloigne en campagne témoigne du manque de robustesse de l'algorithme aux données atypiques, conjugué à l'imprécision de certaines données.

5.2 Analyse par requête

L'objectif ici n'est pas de lister de manière exhaustive chaque résultat qu'il est possible d'obtenir grâce aux tables, de rapidement en présenter quelques uns. Une statistique par table sera fournie en moyenne à titre d'exemple, quelques requêtes et traitements étant écrits dans le fichier annexe *select.py* :

— Pourcentage de chaque *call_ type* :

1. B = 47.93%
2. C = 30.22%

3. A = 21.85%

On voit donc que majoritairement les taxis sont demandés depuis un stand spécifique.

— Les deux stands de taxi les plus populaires (un fichier de métadonnées est disponible sur le web pour faire correspondre les numéros de stand aux lieux dans Porto) :

1. Stand n°15, devant le monument Almeida Garrett, en plein centre historique de Porto, avec 18261 trajets recensés.
2. Stand n°57, pas loin du stand précédent, qui représente la gare de São Bento, l'une des plus belles du monde et la gare principale de Porto en plein centre-ville. 11771 trajets y ont été recensés.

— Jour parmi toutes les dates du *dataset* pour lequel les trajets ont été les plus nombreux : 01/01/2014 soit le jour du nouvel an. Une autre requête permet de voir que 4382 trajets y ont été effectués entre minuit et 9h du matin soit plus de 60% du total de la journée.

— Heures de la journée toute date confondue à laquelle les trajets sont le plus nombreux :

1. 10h
2. 11h
3. 15h

Plus de 270000 trajets (90 000 chacun) ont été effectués sur ces plages horaires tandis que la nuit on dépasse très rarement les 50 000 pour une heure donnée.

— Identifiant des numéros de téléphone des 3 meilleurs clients (au sens abstrait) sur la période 2013/2014 :

1. Numéro : *2002* avec 57026 trajets
2. Numéro : *63882* avec 6371 trajets
3. Numéro : *2001* avec 2416 trajets

À savoir que la moyenne du nombre de trajets pour chaque client est d'environ 6.3, on a ici une variance très élevée. Ces numéros sont difficilement interprétables car on ne sait pas ce qu'il signifie. Ce peut être un vrai numéro comme un identifiant artificiel analogue à ceux des stands, or nous n'avons pas les correspondances. On peut peut-être supposer que les numéros principaux correspondent à des agences de réservation de taxis, ou encore à des entreprises ou groupes etc. en raison de leur très gros nombre d'appels par rapport à la moyenne.

— 3 plus gros conducteurs de trajets sur la période :

1. ID : *20000403* avec 7833 trajets
2. ID : *20000483* avec 7618 trajets
3. ID : *20000364* avec 7416 trajets

— Statistiques sur la durée d'un trajet de taxi :

1. Plus long : 16h 10m, à noter que les points de départ et d'arrivée de ce trajet sont les mêmes, on peut alors éventuellement supposer un enregistrement non voulu d'une période où le taxi n'exerçait pas et était statique.
2. Plus court : 15s, correspond aux trajets ne possédant que deux points mesurés (un de départ et un d'arrivée)
3. Moyenne : environ 12m 11s

— Types de jour où les taxis sont le plus empruntés :

1. A avec plus de 1 500 000 trajets effectués un jour de cette catégorie (jours classiques),
2. B avec plus de 39 000 trajets effectués un jour férié donc,
3. C avec environ 42 000 effectués donc un jour avant un jour B

On remarque donc que les jours B et C (jours fériés et jour d'avant) représentent un peu plus de 5% des trajets répertoriés par le jeu de données. Par ailleurs, avec 72 dates sur les années 2013 et 2014, une rapide approximation montre que cela représente envirion 10% du total des dates dans la période. Ces jours n'incitent donc pas vraiment une consommation de taxis plus élevée par rapport à d'habitude.

Les statistiques par année et mois n'ont pas été jugées utiles. Il aurait été en revanche très utile de déterminer avec un système de calcul calendaire quel jour de la semaine est le plus fréquenté par les utilisateurs de taxis...

6 Conclusion

De nombreuses améliorations auraient pu être faites lors de l'insertion en insérant plus d'informations utiles, ou encore en préparant encore plus les données : par exemple en insérant directement les valeurs de latitude/-longitude d'arrivée et de départ sous forme de quatre nombres flottants et en effectuant d'autres calculs. On aurait aussi pu faire les insertions par batch (lots de données) afin d'assurer une alimentation progressive qui en cas d'erreur et d'arrêt permet de reprendre uniquement les lots concernés. On peut aussi critiquer certains choix d'implémentation comme les arrondis de longitude et latitude ainsi que la suppression du *polyline* représentant le trajet, ce qui aurait permis de l'analyser lors de la rencontre de données atypiques.

Le stockage de données formatées et propres dans un deuxième fichier csv aurait finalement amélioré l'efficacité du script python *insert_ train.csv* qui se serait concentré sur la copie des données seulement et non leur nettoyage.

Une des difficultés a été la rencontre de tous ces problèmes au fur et à mesure des tests, ce qui nous a mené à effectuer de petites améliorations à chaque fois sur le tas, même si les résultats sont présents.

Table des matières

Table des figures

7 Annexes

7.1 create_train.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri May 19 10:58:28 2017

@author: nf26p006
"""
from cassandra.cluster import Cluster

cluster = Cluster()

session = cluster.connect('e28')


session.execute("""
DROP TABLE trip_taxis;
""")
session.execute("""
DROP TABLE trip_distances;
""")
session.execute("""
DROP TABLE trip_call_types;
""")
session.execute("""
DROP TABLE trip_origin_stands;
""")
session.execute("""
DROP TABLE trip_origin_calls;
""")
session.execute("""
DROP TABLE trip_years;
""")
session.execute("""
DROP TABLE trip_months;
""")
session.execute("""
DROP TABLE trip_days;
""")
session.execute("""
DROP TABLE trip_hours;
""")
session.execute("""
DROP TABLE trip_arrivals;
""")
session.execute("""
DROP TABLE trip_departures;
""")
session.execute("""
DROP TABLE trip_duration;
""")
session.execute("""
DROP TABLE trip_day_types;
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_taxis (trip_id double, taxi_id double, year int, month int, day
                                                    int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(taxi_id, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_distances (trip_id double, taxi_id double, year int, month int,
                                                    day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
```

```
origin_call double, primary key(dist, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_call_types (trip_id double, taxi_id double, year int, month int
                                                , day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(call_type, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_origin_stands (trip_id double, taxi_id double, year int, month
                                                int, day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(origin_stand, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_origin_calls (trip_id double, taxi_id double, year int, month
                                                int, day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(origin_call, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_hours (trip_id double, taxi_id double, year int, month int, day
                                                int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(hour, year, month, day, min, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_days (trip_id double, taxi_id double, year int, month int, day
                                                int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key((day, month, year), hour, min, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_months (trip_id double, taxi_id double, year int, month int,
                                                day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(month, year, day, hour, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_years (trip_id double, taxi_id double, year int, month int, day
                                                int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(year, month, day, hour, min, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_departures (trip_id double, taxi_id double, year int, month int
                                                , day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(starting_point, ending_point, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_arrivals (trip_id double, taxi_id double, year int, month int,
                                                day int,
hour int, min int, daytype varchar, starting_point varchar,
```

```
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, primary key(ending_point, starting_point, trip_id))
""")



session.execute("""
CREATE TABLE IF NOT EXISTS trip_duration (trip_id double, taxi_id double, year int, month int,
                                                day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, duration double, primary key(duration, trip_id))
""")

session.execute("""
CREATE TABLE IF NOT EXISTS trip_day_types (trip_id double, taxi_id double, year int, month int,
                                                day int,
hour int, min int, daytype varchar, starting_point varchar,
ending_point varchar, dist double, call_type varchar, origin_stand double,
origin_call double, duration double, primary key(daytype, trip_id))
""")
```

7.2 insert_train.py

```
# -*- coding: utf-8 -*-
"""
Created on Fri May 19 10:58:28 2017

@author: nf26p006
"""

import csv
import datetime
import pandas as pd

file = pd.read_csv("/home/e28/train.csv", sep=";", encoding = 'latin_1')

from cassandra.cluster import Cluster

cluster = Cluster()

session = cluster.connect('e28')
session.default_timeout = 9999

def dist(lon1, lon2, lat1, lat2):
  import numpy as np
  RT = 6371008
  d = np.sqrt(
     ((lon1-lon2)*np.cos((lat1+lat2)/2/180*np.pi))**2
  )/180*np.pi*RT
  return d

def insert(table_name):
  with open('/train.csv') as f:
    nbligne = 0
    l=f.readline()
    while True:
      l=f.readline()
      nbligne += 1
      if len(l)==0:
        break

      data = l.split("\",\"")
      id_trip = data[0][1:]
      call_type = data[1]
      origin_call = data[2]
      origin_stand = data[3]
      id_taxi = data[4]
```

```
timestamp = data[5]
day_type = data[6]
data_missing = data[7]
chemin = data[8][2:-4]
positions = chemin.split("],[")


date = datetime.datetime.fromtimestamp(int(timestamp))
year = date.year
month = date.month
day = date.day
hour = date.hour
minute = date.minute
duration=0


Bdays = ['112013','2932013','3132013','2542013','152013','1062013','1582013','8122013','
                                        25122013','112014','1842014','2042014','2542014
                                        ','152014','1062014','1582014','8122014','
                                        25122014']
Cdays = ['31122012','2832013','3032013','2442013','3042013','962013','1482013','7122013',
                                        '24122013','31122013','1742014','1942014','
                                        2442014','3042014','962014','1482014','7122014'
                                        ,'24122014']

daymonthyear = str(day) + str(month) + str(year)

if daymonthyear in Bdays:
    day_type='B'

if daymonthyear in Cdays:
    day_type='C'


d_requete = "INSERT INTO e28." + table_name + " (trip_id, taxi_id"
f_requete = "VALUES (%s, %s" % (id_trip, id_taxi)

d_requete += ", year, month, day, hour, min, daytype"
f_requete += ", %s, %s, %s, %s, %s, '%s'" % (year, month, day, hour, minute, day_type)

d_requete += ", call_type"
f_requete += ", '%s'" % (call_type)

if(call_type == "A"):
  if call_type:
    d_requete += ", origin_call"
    f_requete += ", %s" %(origin_call)
  else:
    continue

elif(call_type == "B"):
  if origin_stand:
    d_requete += ", origin_stand"
    f_requete += ", %s" %(origin_stand)
  else:
    continue

if (len(positions)>=2):
  duration = len(positions) * 15
  duration = duration - 15
  lon1= float(positions[0].split(",")[0])
  lat1= float(positions[0].split(",")[1])
  lon2= float(positions[-1].split(",")[0])
  lat2= float(positions[-1].split(",")[1])
  distance = dist(lon1,lat1,lon2,lat2)
  lon1= "%.2f" % float(positions[0].split(",")[0])
  lat1= "%.2f" % float(positions[0].split(",")[1])
  lon2= "%.2f" % float(positions[-1].split(",")[0])
  lat2= "%.2f" % float(positions[-1].split(",")[1])

  d_requete+= ", starting_point, ending_point, dist, duration"
```

```
        f_requete+= ", '[%s,%s]','[%s,%s]', %s, %s" % (lon1,lat1,lon2,lat2,distance,duration)
      else:
        continue

      d_requete += ") "
      f_requete += ");"


      session.execute(d_requete+f_requete)
      print(nbligne)

#Normally other tables names in table_names!
table_names = ["trip_day_types"]

for table_name in table_names:
  insert(table_name)
```

7.3 insert_stands_calls.py

Variante pour l'insertion des tables trip_origin_stands et trip_origin_calls

```
# -*- coding: utf-8 -*-
"""
Created on Fri May 19 10:58:28 2017

@author: nf26p006
"""

import csv
import datetime
import pandas as pd

file = pd.read_csv("/home/e28/train.csv", sep=";", encoding = 'latin_1')

from cassandra.cluster import Cluster

cluster = Cluster()

session = cluster.connect('e28')
session.default_timeout = 9999

def dist(lon1, lon2, lat1, lat2):
  import numpy as np
  RT = 6371008
  d = np.sqrt(
     ((lon1-lon2)*np.cos((lat1+lat2)/2/180*np.pi))**2
  )/180*np.pi*RT
  return d

def insert(table_name):
  with open('/train.csv') as f:
    nbligne = 0
    l=f.readline()
    while True:
      l=f.readline()
      nbligne += 1
      if len(l)==0:
        break

      data = l.split("\",\"")
      id_trip = data[0][1:]
      call_type = data[1]
      origin_call = data[2]
      origin_stand = data[3]
      id_taxi = data[4]
      timestamp = data[5]
      day_type = data[6]
      data_missing = data[7]
      chemin = data[8][2:-4]
      positions = chemin.split("],[")
```

```
date = datetime.datetime.fromtimestamp(int(timestamp))
year = date.year
month = date.month
day = date.day
hour = date.hour
minute = date.minute


Bdays = ['112013','2932013','3132013','2542013','152013','1062013','1582013','8122013','
                                        25122013','112014','1842014','2042014','2542014
                                        ','152014','1062014','1582014','8122014','
                                        25122014']
Cdays = ['31122012','2832013','3032013','2442013','3042013','962013','1482013','7122013',
                                        '24122013','31122013','1742014','1942014','
                                        2442014','3042014','962014','1482014','7122014'
                                        ,'24122014']

daymonthyear = str(day) + str(month) + str(year)

if daymonthyear in Bdays:
    day_type='B'

if daymonthyear in Cdays:
    day_type='C'


d_requete = "INSERT INTO e28." + table_name + " (trip_id, taxi_id"
f_requete = "VALUES (%s, %s" % (id_trip, id_taxi)

d_requete += ", year, month, day, hour, min, daytype"
f_requete += ", %s, %s, %s, %s, %s, '%s'" % (year, month, day, hour, minute, day_type)

d_requete += ", call_type"
f_requete += ", '%s'" % (call_type)

if(call_type == "A"):
  if call_type:
    d_requete += ", origin_call"
    f_requete += ", %s" %(origin_call)
  else:
    continue

elif(call_type == "B"):
  if origin_stand:
    d_requete += ", origin_stand"
    f_requete += ", %s" %(origin_stand)
  else:
    continue

if (len(positions)>=2):
  lon1= float(positions[0].split(",")[0])
  lat1= float(positions[0].split(",")[1])
  lon2= float(positions[-1].split(",")[0])
  lat2= float(positions[-1].split(",")[1])
  distance = dist(lon1,lat1,lon2,lat2)
  lon1= "%.2f" % float(positions[0].split(",")[0])
  lat1= "%.2f" % float(positions[0].split(",")[1])
  lon2= "%.2f" % float(positions[-1].split(",")[0])
  lat2= "%.2f" % float(positions[-1].split(",")[1])

  d_requete+= ", starting_point, ending_point, dist"
  f_requete+= ", '[%s,%s]','[%s,%s]', %s" % (lon1,lat1,lon2,lat2,distance)
else:
  continue

d_requete += ") "
f_requete += ");"

if (origin_call!=""):
```

```
        session.execute(d_requete+f_requete)
        print(nbligne)
      else:
        print(nbligne-1)


table_names = ["trip_origin_calls", "trip_origin_calls"]


for table_name in table_names:
  insert(table_name)
```

7.4 analysis.py

Code des k-means :

```
import matplotlib.pyplot as plt
'''
umap.openstreetmap.fr
plt.plot(x,y,'v')
plt.show()
plt.savefig()

result = session.execute(select...) retourne un iterable
-> for row in result

j le num de centroid
    s[j]+=x
    n[j]+=1

C=s/n


1) Randomly select 'c' cluster centers.

2) Calculate the distance between each data point and cluster centers.

3) Assign the data point to the cluster center whose distance from the cluster center is
                                                minimum of all the cluster centers..

4) Recalculate the new cluster center

5) Recalculate the distance between each data point and new obtained cluster centers.

6) If no data point was reassigned then stop, otherwise repeat from step 3).


New center cluster 2 --> n2 * center2 + point / n2 + 1
New center cluster 1 --> n1 * center1 - point / n1 - 1
'''



import numpy as np
from cassandra.cluster import Cluster
from cassandra.query import SimpleStatement

def dist(lon1, lon2, lat1, lat2):
  import numpy as np
  RT = 6371008
  d = np.sqrt(
     ((lon1-lon2)*np.cos((lat1+lat2)/2/180*np.pi))**2
  )/180*np.pi*RT
  return d

def kmeans_initial_centers(number):
    global centers
    centers = [0] * number
    centers_string = [""] * number
    for n in range(number):
        min = 1
```

```
        point = "[00.00,00.00]"
        rows = session.execute("SELECT starting_point FROM e28.trip_departures;")
        for row in rows:
            rand = np.random.uniform(0,1)
            if rand < min:
                if row.starting_point not in centers_string:
                    point = row.starting_point
                    min = rand
        coordinates = point.split("[")[1]
        coordinates = coordinates.split("]")[0]
        lat, lon = coordinates.split(",")
        centers_string[n] = [lat,lon]
        lat = float(lat)
        lon = float(lon)
        centers[n] = [lat,lon]

    return centers



def assign_point_and_recalculate_centers(point):
    global labels
    global centers
    global center_cards
    global changed
    global table_iter
    current_distance = 999999999999999999999999
    # Find which cluster the points belong to
    cluster = labels[row_number]
    # Find closest center
    for center in centers:
        dist_to_center = dist(point[0],center[0],point[1],center[1])
        if  dist_to_center < current_distance:
            current_distance = dist_to_center
            cluster = centers.index(center)
    # If closest center changed
    if cluster != labels[row_number]:
        old_cluster = labels[row_number]
        new_cluster = cluster
        # Change label
        labels[row_number] = cluster
        #print(labels[row_number])
        # Recalculate new cluster center
        centers[new_cluster] = list(np.divide(np.multiply(center_cards[new_cluster], centers[
                                                    new_cluster]) + point, center_cards[
                                                    new_cluster] + 1))
        # Change new cluster cardinality
        center_cards[new_cluster] += 1
        if table_iter != 1:
            # Recalculate previous cluster center
            centers[old_cluster] = list(np.divide(np.multiply(center_cards[old_cluster],
                                                        centers[old_cluster]) + point,
                                                        center_cards[old_cluster] - 1))
            # Change previous cluster cardinality
            center_cards[old_cluster] -= 1
        # Assert that the point has switched of cluster
        changed = 1

def k_means(number):
    # Flag to check if one point has switched of cluster during one iteration
    global changed
    changed = 1
    # List of cluster label for each table row
    global labels
    labels = [-1] * 1662809
    # List of clusters centers
    global centers
    centers = kmeans_initial_centers(number)
    # List of clusters cardinalities
    global center_cards
    center_cards = [0] * number
```

```
    global table_iter
    table_iter = 0
    # While points continue to move between clusters
    while changed:
        table_iter += 1
        print(table_iter)
        changed = 0
        statement = SimpleStatement("SELECT starting_point FROM e28.trip_departures;",
                                                                fetch_size=100)
        rows = session.execute(statement)
        # Row number of table
        global row_number
        row_number = 0
        for row in rows:
            row_number += 1
            # Parse latitude and longitude of the point
            coordinates_point = row.starting_point.split("[")[1]
            coordinates_point = coordinates_point.split("]")[0]
            lat_point, lon_point = coordinates_point.split(",")
            lat_point = float(lat_point)
            lon_point = float(lon_point)
            point = [lat_point, lon_point]
            assign_point_and_recalculate_centers(point)
    #print(labels)

cluster = Cluster()

session = cluster.connect('e28')
session.default_timeout = 9999

k_means(6)
print(centers)
print(center_cards)
# print(labels)

# plt.scatter(coordinates[:,0], coordinates[:,1], c=y);
# plt.show()
```

7.5 select.py

Analayse par requêtes :

```
# -*- coding: utf-8 -*-
"""
Created on Fri May 19 10:58:28 2017

@author: nf26p006
"""

import csv
import datetime
import numpy as np

from cassandra.cluster import Cluster

cluster = Cluster()

session = cluster.connect('e28')
session.default_timeout = 9999

nb_rows = 0
requeteA = "SELECT COUNT(call_type) FROM e28.trip_call_types WHERE call_type='A';"
rows = session.execute(requeteA)

for row in rows:
  print("Calls_types_count A :")
  print(row)

requeteB = "SELECT COUNT(call_type) FROM e28.trip_call_types WHERE call_type='B';"
rows = session.execute(requeteB)
```

```
for row in rows:
  print("Calls_types_count B :")
  print(row)

requeteC = "SELECT COUNT(call_type) FROM e28.trip_call_types WHERE call_type='C';"
rows = session.execute(requeteC)


for row in rows:
  print("Calls_types_count C:")
  print(row)

requete = "SELECT origin_stand, count(origin_stand) FROM e28.trip_origin_stands GROUP BY
                                                    origin_stand;"
rows = session.execute(requete)

for row in rows:
  print("Origin_stands_count C:")
  print(row)




requete = "SELECT hour, count(hour) FROM e28.trip_hours GROUP BY hour;"
rows = session.execute(requete)

for row in rows:
  print("Hours_count:")
  print(row)


requete = "SELECT max(dist) FROM e28.trip_distances;"
rows = session.execute(requete)

for row in rows:
  print("Max distance:")
  print(row)

requete = "SELECT min(dist) FROM e28.trip_distances;"
rows = session.execute(requete)

for row in rows:
  print("Min distance:")
  print(row)


requete = " SELECT day,month,year, COUNT(*) FROM e28.trip_days GROUP BY day,month,year;"
rows = session.execute(requete)
lines = []
for row in rows:
  line = (row.day, row.month, row.year, row.count)
  lines.append(line)

sorted_counts = sorted(lines, key=lambda tup: tup[3])
print(sorted_counts)

select hour, count(hour) from trip_days where day = 1 and month=1 and year=2014 group by hour;
 --> more during night


requete = "SELECT origin_call, count(*) FROM e28.trip_origin_calls GROUP BY origin_call;"
rows = session.execute(requete)
lines = []
for row in rows:
  line = (row.origin_call, row.count)
  lines.append(line)

sorted_counts = sorted(lines, key=lambda tup: tup[1])
print(sorted_counts)
print(np.mean([x[1] for x in sorted_counts]))
```

```
requete = "SELECT taxi_id, count(*) FROM e28.trip_taxis GROUP BY taxi_id;"
rows = session.execute(requete)
lines = []
for row in rows:
  line = (row.taxi_id, row.count)
  lines.append(line)

sorted_counts = sorted(lines, key=lambda tup: tup[1])
print(sorted_counts)
print(np.mean([x[1] for x in sorted_counts]))



requete = "select trip_id, max(duration) from trip_duration ;"
rows = session.execute(requete)

for row in rows:
  line = (row[0], row[1])

print(line)
```