Université de Technologie de Compiègne

NF26

Data Warehouse

# Rendu TP Cassandra

Zineb SLAM

June 25, 2017

**Abstract**

Dans cette deuxième partie du TP de NF26 nous allons traiter de gros volumes de données avec Cassandra. Nous allons d'abord commencer par décrire nos données, puis présenter une modélisation des Datamarts et pour finir nous allons expliquer comment nous avons exploités ces données. Vu le cours temps dont nous avons disposé et la nouveauté de cette méthode de stockage ainsi que les langages CQL et Cassandra, nous n'avons malheureusement pas pu aller en profondeur dans tous les concepts et nous ne sommes restreint à une seule méthode d'analyse statistique. Néanmoins nous avons essayé de les détailler au maximum les méthodes appliquées en soulignant les prochaines démarches à prendre pour la continuité du projet. Comme le contexte du projet ne nous pas été définie, il a fallu émettre des hypothèse au cours de notre étude pour argumenter notre modélisation.

# Contents

# 1 Description des Données

Nous disposons d'un ensemble de données précis décrivant l'année complète (du 01/07/2013 au 30/06/2014) des trajectoires effectuées par 442 taxis dans la ville de Porto au Portugal (c'est-à-dire 3Gb de données Stocké dans un seul fichier CSV). Ces taxis fonctionnent à travers un centre d'expédition de taxi, en utilisant des terminaux de données mobiles installés dans les véhicules (GPS et autres). Chaque trajet est caractérisé en trois catégories: A) basé sur le taxi centralisé, B) à base de stand ou C) sur n'importe quelle rue. Chaque entrée de données correspond à un un trajet complété. Il contient au total 9 attributs, décrits comme suit:

- **TRIP_ID**: est un String qui identifie chaque trajet du taxi
- **CALL_TYP**E est un caractère désignant comment la demande de service a été faite
  - A appel du centre/station
  - B une demande directe au taxi dans un point d'arrêt
  - C une demande faite dans n'importe quel point de la rue
- **ORIGIN_CALL** est un entier contenant le numéro de téléphone que le client a utilisé pour demander un service. Cet attribut est défini que si CALL_TYPE=A si non la valeur est nulle
- **ORIGIN_STAND**: est une identifiant du point d'arrêt/station du départ de du taxi, si CALL_TYPE=B si non la valeur est nulle
- **TAXI_ID**: est entier contenant l'identifiant du taxi effectuant le trajet
- **TIMESTAMP**: est un entier désignant la date et heure du début du trajet en UNIX TimeStamp.
- **DAYTYPE**: est un caractère identifiant le type de jour du début du trajet. Il prend une de ces valeur:
  - 'A' pour un jour normal (jour de semaine ou week-end)
  - 'B' si le voyage a commencé en un jour de vacances ou un jour de fête.
  - 'C' si le voyage a commencé la veille d'un jour de type B
- **MISSING_DATA** est un booléen qui est *TRUE* si la suite de localisations du taxi par GPS est complète, *FALSE* si des coordonnées sont manquantes.
- **POLYLINE** est une suite de coordonnées enregistrés par le GPS du taxi toutes les 15 secondes; sous la forme [longitude, latitude]. La première entrée est le point de départ tandis que la dernière est la destination du voyage.

# 2 Modélisation

Nous avons identifié 5 dimensions pour notre base de données:

## 2.1 Dimensions

Notre modélisation est représentée par le diagramme ci-dessous:

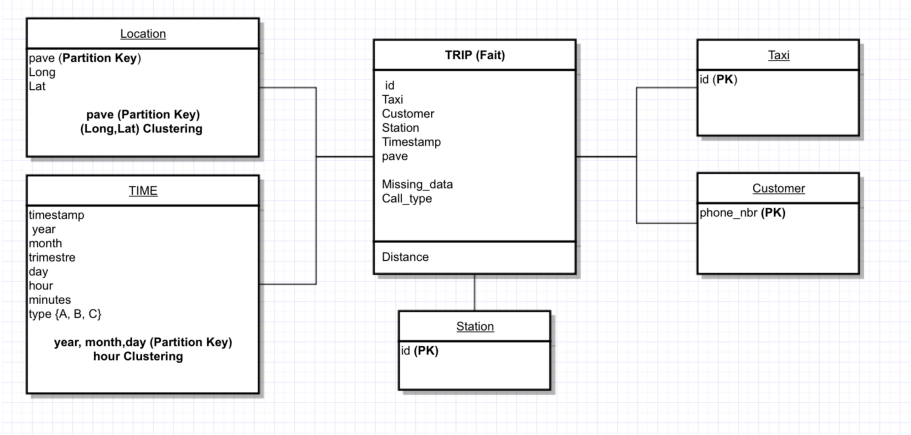



Figure 1: Modelisation de la Base de Données

Les dimensions sont:

- **Géographique**: notre dimension géographique est formée d'une table de localisation. Dans cette table nous avons implémenté un pavé comme partition key ensuite nous avons ordonné chaque partition par longitude et latitude. Nous allons détailler le calcul de ce pavé dans les sections suivantes.
- **Temporelle**: notre dimension Temporelle est composée d'une seule table Date. Dans nos données nous avons uniquement le TimeStamp et type de jour. Mais il sera pertinent d'avoir les attributs années, mois, jour et heure. Pour avoir une partition suffisamment grande mais qui ne croît pas indéfiniment on choisir de partitionner par année, par mois et par jour. Nous justifierons davantage ce choix dans la section suivante.
- **Taxi**: elle est composée d'une table Taxi avec tous les id des Taxi
- **Station** est composée d'une table Station avec les id de chaque station (ce qui correspond à origin_stand)
- **Client** est composée d'une table avec comme attribut un id qui correspond au numéro de téléphone du client (origin_call)

## 2.2 Faits

Nous avons modélisé nos table de faits en répondant à la question: "Que voudrai je afficher ou représenter?". Ainsi dans chacune des sous-parties qui suivent nous allons commencer par énoncer la ou les questions ensuite de trouver la modélisation de la table de fait pour y répondre.

## 2.3 Attributs

Notons que toutes nos tables de faits ont les mêmes attributs qui correspondent en majorité à ceux du fichier csv. Nous avons décidé de garder le maximum d'information pour que notre modèle soit extensible dans le temps. C'est a dire si on voudrait répondre à d'autres sous-question ou obtenir plus d'informations on ne sera pas amené à re-modéliser notre table de fait. Nous avons décidé dans le cadre de ce TP de ne pas insérer la liste Polyline dans notre table de fait et ce pour 2 raisons; la première est que cette liste va considérablement augmenter la taille de notre base de données alors que nous ne l'exploiterons pas dans nos questions. La seconde est que le matériel dont nous disposons à l'UTC pourrait ne pas supporter une aussi grande taille de partition. Il faudra quand même garder en tête que ceci n'est qu'une première modélisation et que plus tard si ce modèle sera ré-utilisé il faudra penser à rajouter cette information de trajet dans au moins un des tables pour des raisons d'évolutivité.

**Question 1**

- Quel est le nombre de trajets par mois et par jour?
- Comment le nombre de trajets évolue au cours de la journée?
- Quelle est la distance totale parcourue par période?

Il est bien claire d'après ces questions qu'on voudrait obtenir des informations par rapport à une date et donc on devrait avoir une partition par date. La question qui se pose ici est quels sont les attributs à regrouper dans la clés de partitionnement. Comme il a déjà été cité dans la modélisation de la dimension date. Il faut que notre partition n'augmente pas indéfiniment au cours du temps et qu'elle ne soit pas trop fine. On a alors hésité entre (year,month) et (year, month, day). En choisissant (year,month) nous avons 12 partitions par an avec environ 400,000 entrées par partition. Tandis qu'avec (year,month,day) nous avons 365 partitions par an et environ 6000 entrée par partition. Comme on ne connaît pas a priori la taille des nœuds dans le serveur on ne peut pas trancher entre les 2 propositions. Mais si on suppose que nos données vont augmenter et qu'on aura une croissance de trajets au cours des années on choisira plutôt la clés de partitionnement (year, month,day). De plus le fait de l'utilisation de 3 colonnes au lieu de 2 pour acheminer vers une partition peut diminuer les *hotspots.*Ainsi on aura une partition pour chaque jour de l'année. Cette partition sera ordonnée par heure puis par trip_id pour différencier entre 2 trajets qui ont

lieu à la même date et heure. Ci-dessous la modélisation de notre table de fait. Autre remarque, on ne connaît pas le nombre de serveurs dont on dispose mais avec 365 partitions par an il nous faut au moins 10 serveur pour avoir 36 partitions par serveur.

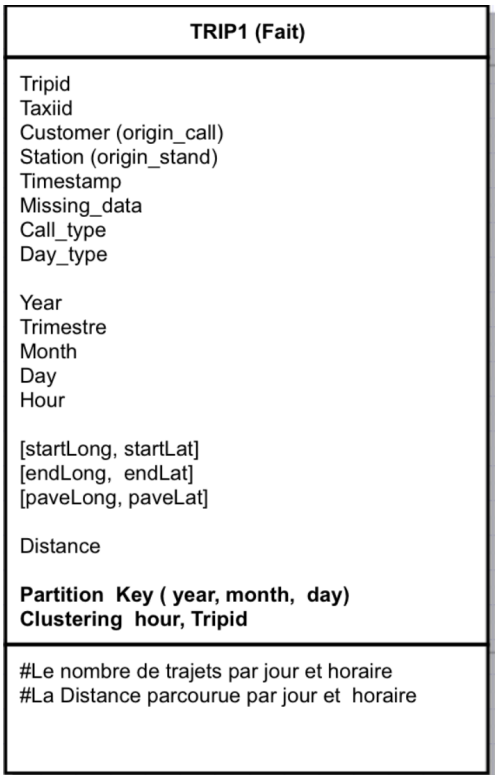



Figure 2: Modélisation de la table de fait Trip1

**Question 2**

- Quel sont les trajets/ ou le nombre de trajets qui partent d'un pavé entre 2 intervalles de temps( par exemple entre le 10 Juillet 2013 et le 10 Septembre 2013)?

Dans cette modélisation nous utilisons le pavé pour regrouper les coordonnées de départ des trajets. Nous arrondissons le pavé au dixième près, c'est a dire $pave_{longitude} = (longitude * 10)/10, pave_{latitude} = (latitude * 10)/10$. Nous avons trouvé plus judicieux d'utiliser comme clés partitionnement (pavé, year) plutôt que (pave, year, month) car avec la deuxième nous aurons 12 fois plus de partitions par année ce qui peut être considérablement lourd comme on a un grand nombre de pavé. Ensuite nous ordonnons les données par month, day, hour et enfin Tripid pour différencier les trajets.

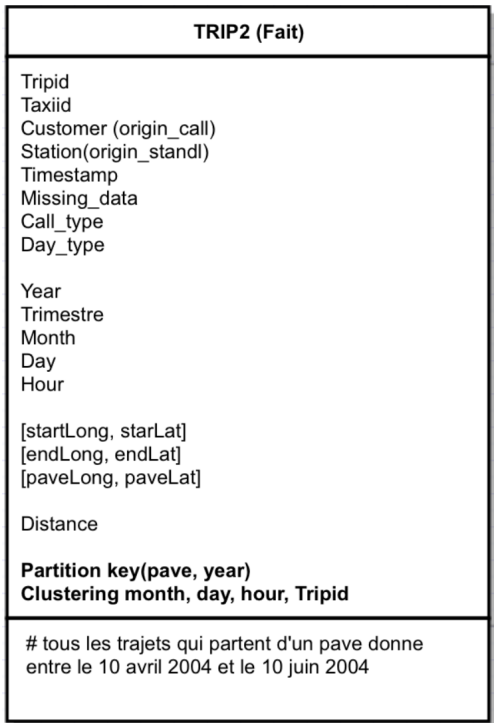



Figure 3: Modélisation de la table de fait Trip2

**Question 3**

- Le meilleure Taxi du mois
- Nombre moyen de trajet par Taxi

Pour répondre à des question spécifiques a chaque Taxi, il est évident qu'il faut partitionner par l'id du Taxi. Nous avons ensuite considéré le partitionnement par année et par mois. Ainsi si un Taxi fait en moyenne 10 trajets par jour notre partition aura une taille de 3650 ce qui est raisonnable. Le seule inconvénient ici est qu'on aura une partition pour chaque taxi et donc un grand nombre de partitions il serait donc mieux d'avoir plusieurs serveur pour pouvoir les partager.

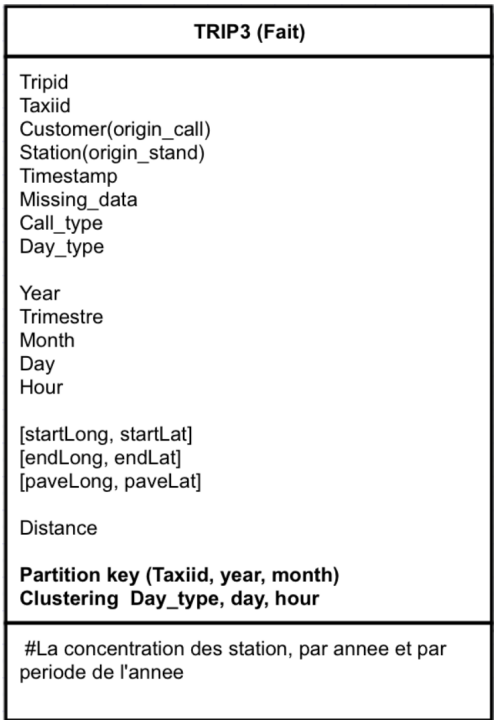



Figure 4: Modélisation de la table de fait Trip3

**Question 4**

- La concentration des stations par année, mois , type de jour
- les stations les plus fréquentées

On suit ici le même principe que pour les Taxi. Nous partitionnons par l'id de la station, l'année et le mois. Nous ne sommes pas allés jusqu'à partitionner par jour parce qu'en exploitant les données mises à disposition nous remarquons qu'il n'y' a pas vraiment beaucoup de trajets de type B.

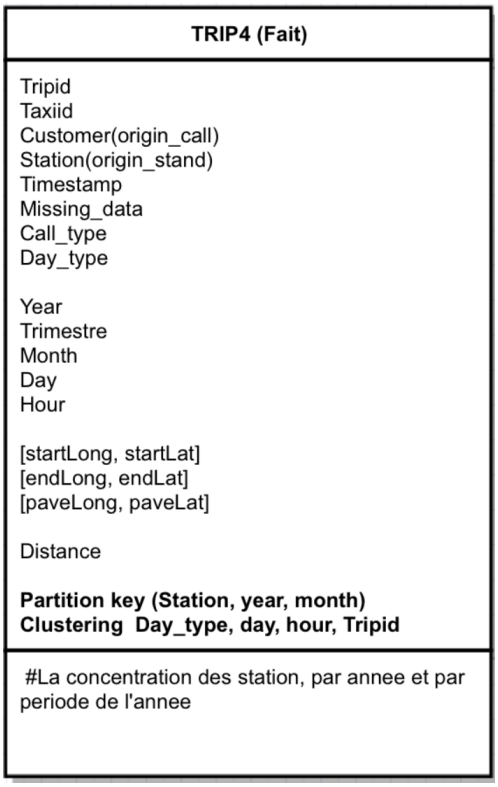



Figure 5: Modélisation de la table de fait Trip4

**Question 4**

,

- Le meilleur client
- Les destinations favorites d'un client
- Les jours ou types de jours ou un client utilise le plus les taxi

Ce modèle a été proposé pour exploiter les données pour des fins commerciales. En effet on pourrait se demander quels sont les trajets préférés d'un client pour qu'on puisse envoyer a ces derniers des messages de promotions ou de réductions. Ceci est utilise par exemple par Uber ou Lyft. Nous pourrons aussi étudier les trajets d'un client afin de prédire ses prochains déplacements et lui proposer de bons plans. Nous allons donc partitionner dans ce cas par idClient (le numéro de téléphone) et l'année. Cette solution n'a pas été mise en place dans la suite par faute de temps et aussi parce que nous n'avons pas autant d'informations du les clients (c'est à dire un client peut prendre un taxi sans appeler et donc on ne saura pas que c'est lui), mais on a trouvé judicieux de la mentionner pour l'evolutivite du projet.

# 3 Nettoyage et Insertion des données en Python

## 3.1 Nettoyage

Nous avons commencé par passer le fichier csv en gardant les même types que ceux dans la description des données. Nous avons passé la liste de GPS polyline pour n'en garder que les coordonnées de départs et d'arrivée. Nous avons aussi remplacé tous les attributs nuls de origin_call et origin_stand par 0 lorsqu'ils sont vides.

## 3.2 Insertion

Une fois que nos données sont nettoyées l'insertion n'est pas difficile du moment qu'on fait attention à bien faire les conversions des types. Pour éviter de faire la requête d'insertion ligne par ligne on pourrait utiliser les *BATCH STATEMENT* ou on pourrait insérer chaque 500 lignes par exemple. Le code avec le *BATCH* est mis en commentaire à la fin du fichier d'insertion *taxi.py* en annexe.

# 4 Exploitation et Visualisation des données

## 4.1 Trip1

Nous interrogeons ici notre table de fait Trip1 pour répondre aux questions posées dans la section précédente. Comme notre clés de partitionnement est (year,month,day) il faut que notre requête *SELECT* contienne des conditions sur ces 3 attributs avec le *WHERE*. Dans un premier temps pour obtenir l'évolution du nombre de trajet par heure nous avons utilisé la requête suivante:

```
1    SELECT hour, COUNT(tripid) as nb FROM TRIP1 WHERE
         year=2013 AND month=9 AND day=9 GROUP by hour
```

Ensuite nous avons décidé d'opter le *GROUP BY* car celui alourdit la requête en filtrant la partition. Nous récupérons donc une requête avec tous les trajets pour toutes les heures,et comme nous avons définit les heures comme clés de clustering le résultat sera trié par ordre croissant des heures, il suffira donc juste de compter le nombre de trajets. Notre nouvelle requête devient alors:

```
1    SELECT day, hour, tripid FROM TRIP1 WHERE year=2013 AND
         month=9 AND day=9
```

Les résultats pour le nombre de trajets par heure et la distance parcourue par heure pour le 09/09/2013 sont représentés graphiquement ci-dessous:

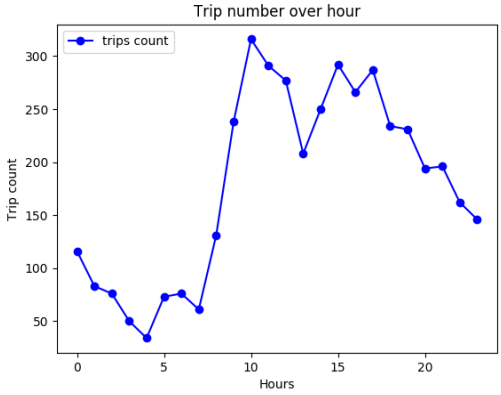



Figure 6: Evolution du nombre de trajets par heure

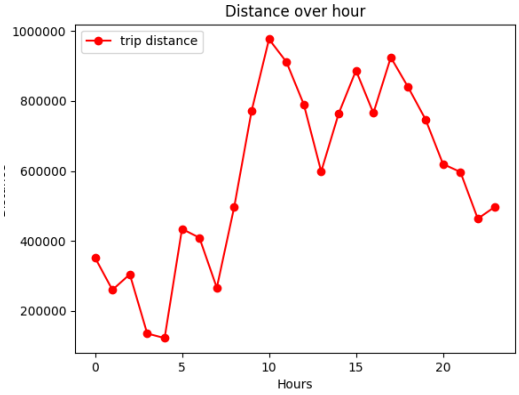



Figure 7: Evolution des distances totales par heure

On remarque par exemple que pour le 09/09/2013 le flux des taxi est plus important entre 10h a 14. Il faudra néanmoins faire plusieurs requêtes pour pouvoir généraliser cette règle.
Notons ici que pour obtenir les données d'un mois il faudra itérer et interroger notre base 30 fois, c'est a dire interroger 30 partitions (une partition par jour).

## 4.2 Trip2

Nous souhaitons obtenir le nombre de trajets qui ont été pris à une "zone" donnée (le pavé) durant une année définie. La requête utilisée est illustrée ci-dessous,

```
1    SELECT month, daytype, pavelong, pavelat, tripid FROM
         TRIP2 WHERE year=2013 AND pavelong=-8.1 AND
         pavelat=41.1
```

Le résultat renvoyé par cette requête est affiché par le tableau qui suit:

| month | daytype | startlong | pavelong | startlat | pavelat | calltype | tripid |
|---|---|---|---|---|---|---|---|
| 9 | A | -8.07189 | -8.1 | 41.26602 | 41.2 | C | 1380007556620000351 |
| 12 | A | -8.07468 | -8.1 | 41.26431 | 41.2 | C | 1386628992620000486 |
| 12 | A | -8.07059 | -8.1 | 41.2661 | 41.2 | C | 1386630320620000486 |

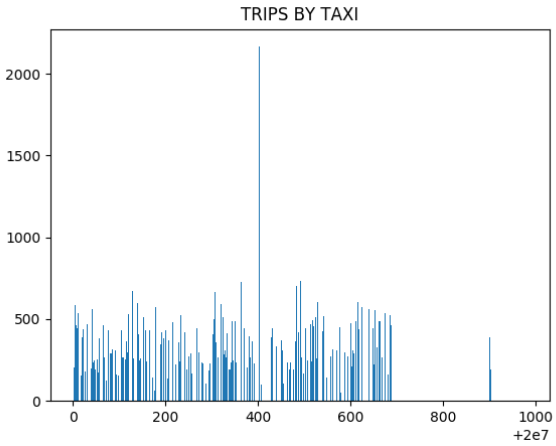
Notons qu'ici aussi nous n'utilisons ni *GROUP BY* ni le *COUNT*.Les trajets seront comptés dans le code Python.

## 4.3 Trip3

Pour obtenir le nombre de trajets par mois d'un taxi il faut qu'on précise son id dans la requête. On utilise pour cela une table taxi qui va stocker les ids des taxis. Ainsi il suffit juste d'itérer sur les id et concaténer l'id a la requête interrogeant la table TRIP3 comme il est illustré ci-dessous:

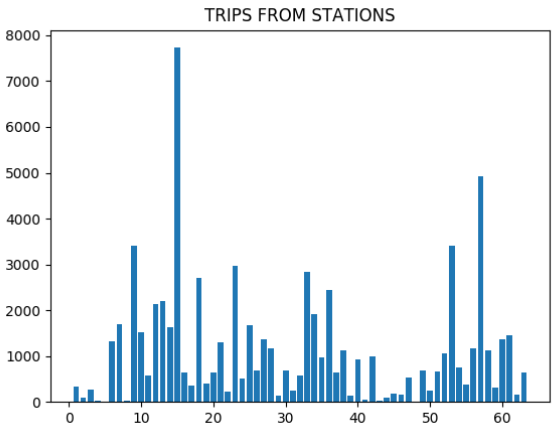
```
query_taxi = session.execute("SELECT id FROM TAXI;")
for user_row in query_taxi :
"SELECT COUNT(tripid) FROM TRIP3 WHERE year=2013 AND
    month=9 AND taxiid=" +str(user_row.id)+ ";"
```

Nous avons essayé de représenter graphiquement les résultats sous forme d'histogramme. Ceci n'est qu'un exemple d'affichage in sera bien évidemment amélioré avec l'avancement du projet en utilisant des librairies de Python plus adaptées.




## 4.4 Trip4

Nous abordons la même démarche que pour les taxi pour questionner et représenter le nombres de trajets qui ont été pris à chaque stations pour un mois donné.



## 4.5 kmeans

### Implémentation

Pour pouvoir visualiser les trajets dans une carte il faudrait pouvoir les regrouper selon des critères. Nous utilisions pour cela les kmeans qui est une méthode non supervisée permettant de regrouper les trajets par *cluster*.
Nous choisissons dans notre cas de regrouper les trajets qui se ressemblent en terme de point de départ et d'arrivée. Ainsi nous avons 2 variables explicatives [startLong, startLat] et [endLong, endLat]. La méthode des kmeans consiste a partir de centroïdes initialisés au départ par des points de notre ensemble $\Omega$ et calculer de nouveaux centroïdes qui minimisent l'inertie intra-classe. Nous supposons ici que la distance d'un point par rapport à un centroïde est calculée en sommant la distance par rapport a la variable 1 et la distance par rapport à la variable 2.
Les fonctions implémentées dans la librairie de Python ne sont pas forcément faite pour traiter de gros jeux de données comme le nôtre, ainsi il est plus conseillée d'implémenter nous même la fonction des kmeans. Pour des soucis d'optimalité nous avons choisi de ne pas garder les points en mémoire car cela peut être très coûteux. Ainsi on interroge notre base de données a chaque itération de l'algorithme pour récupérer nos points. Comme on ne garde pas les points en mémoire nous calculons les coordonnées de chaque centroïde en sommant les coordonnées des points qui lui sont les plus proches.
Les étapes de l'algorithme sont ci-dessous:

1. initialisation des centroïdes par des points de l'ensemble $\Omega$
2. Tant que les nouveau centroïdes calculés sont différent des anciens, Faire:
   (a) une requête SELECT pour récupérer tous points
      i. pour chaque point(i) calculer sa distance aux centroïdes et choisir la distance minimale
      ii. sommer les coordonnées du centroid le plus proche avec ceux du point(i)
   (b) Calcul des nouveaux centroïdes.

Voilà ci dessous les critères pris en compte lors de l'implémentation de l'algorithme:

- Nous initialisons nos centroïdes en choisissant au hasard, grâce a la fonction *randint*, des points de notre ensemble.
- Nous gardons en mémoire les centroïdes précédents a chaque itération pour pouvoir tester la convergence de l'algorithme.
- Nous définissons le nombre de *clusters* a 3

Notre fonction kmeans retourne les centroïdes minimisant l'inertie intra-classe pour cet ensemble et aussi pour cette initialisation des centroïdes. Notons

bien ici que les kmeans trouvent un minimum local de l'inertie intra-classe. Il nous reste ensuite que d'affecter chaque observation au cluster le plus proche. Pour cela on calcule la distance de chaque point aux 3 centroïdes et on affecte celui-ci au centroïde dont il est le plus proche.

Le code de ces 3 fonctions (*distance(), kmeans() et cluster()*)est à trouver en annexe. La fonction cluster prend en paramètre la requête. Cette fonction fait appel à la fonction kmeans pour récupérer les centroïdes ensuite affecté chaque observation a un cluster et retourne une liste avec 3 listes. Chaque sous-liste contient les observations appartenant à ce cluster.
Dans la section qui suit nous avons testé notre fonctions et avons essayé de représenter les résultats obtenus.

### Performance, résultats et extension

Nous avons essayé de comprendre la différence entre les trajets pour un même jour à 10h du matin et 20h du soir. Nous représentons ci-dessous les graphes obtenus après les kmeans. La représentation est faite selon la longitude et la latitude de départ.

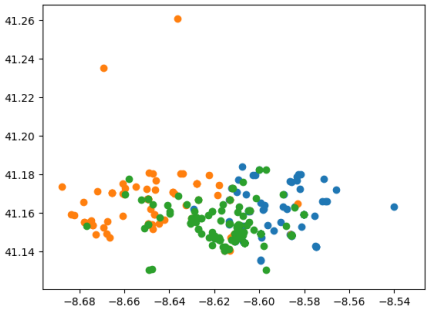



Figure 8: Clustering des trajets de 10h du matin

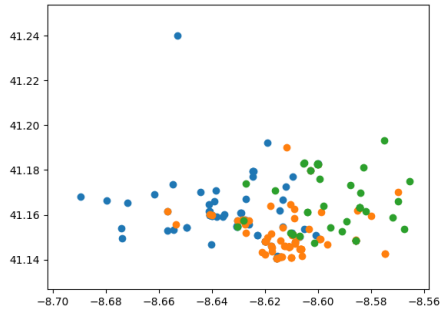



Figure 9: Clustering des trajets de 20h du soir

En raison du temps cours que nous avons eu pour développer ce projet, nous avons d'abord essayé de développer un modèle qui marche, voici donc des points que nous serons amené à améliorer dans la suite pour de meilleures résultats:

- **Initialisation des centroides:** Comme la méthode des kmeans trouve le minimum local selon l'initialisation des centroïdes, il serait judicieux de faire plusieurs appel de la fonction jusqu'à trouver l'initialisation qui minimise le critère d'inertie intra-classe.
- **Nombre de clusters:** Il serait judicieux de tester avec différents nombre (entre 2 et 5 par exemple)et choisir celui qui minimise le critère d'inertie intra-classe.
- **Paramètres:** On pourrait éventuellement changer les paramètres de la fonction *cluster*() pour que celle ci prenne en paramètre que les valeurs de

clés de partitionnement et clustering (l'année, le mois le jour et l'heure)

- **Affichage:** Nous n'avons malheureusement pas pu installer les librairies Python (*folium*) sur les postes de l'UTC qui nous auraient permis de visualiser les données sur la carte.
- **CQL User Defined Function (UDF):** En lisant la documentation *datastax* il s'est avéré qu'on peut écrire des fonctions sur CQL en utilisant le code Python (LANGUAGE Python). On pourra alors utiliser cette méthode de classification automatique directement en interne sans avoir à récupérer les données ou les stocker.

# 5 Conclusion

Ce premier mois de travail sur le projet nous a permis de faire une conception détaillée de notre base de données. Ainsi nous avons pu mettre en place différentes questions qui nous ont permis de modéliser les tables de faits correspondantes. Après nettoyage et insertion des données nous avons pu visualiser les réponses a ces questions. La méthode des kmeans nous permet de mieux étudier les flux des taxi et les visualiser.
Dans la suite du projet nous serons amené a améliorer cette méthode des kmeans et développer d'autres méthodes statistiques comme la régression logistique pour classifier les données.