

# RAPPORT TECHNIQUE

Objectif : L'objectif de ce travail était de concevoir et d'implémenter un entrepôt de données en utilisant le paradigme et la logique proposés par Cassandra. Cet entrepôt de données permettra d'offrir, in fine, différents datamarts correspondants à des requêtes clients types.

## Plan

---

En première partie et en guise d'introduction, nous détaillerons le contexte de nos données : leur nature et leur sens intrinsèque. Aussi, nous en profiterons pour proposer un premier modèle 'conceptuel' de notre data-warehouse : modèle en étoile. Dans une seconde partie, nous verrons que ce modèle n'est pas adapté en NoSQL et en base de données Cassandra. Nous détaillerons donc chaque table de faits dupliquée (mais avec un partitionnement et un clustering différents), correspondant à nos différents datamarts. En troisième et dernière partie, nous verrons un exemple d'utilisation de nos datamarts avec un utilisation d'un algorithme de machine learning : l'algorithme des k-means appliqué sur nos données stockées. Github des fichiers sources (<https://github.com/AlexisDrch/NF26-taxi-cassandra>)

## I. Contexte et analyse conceptuelle

---

Dans ce travail, nous avions à notre disposition un jeu de données récupéré sur le site de l'UCI. Ce dataset est mis à disposition de manière open-source. Les données contenues correspondent à un reporting des trajectoires effectuées par les 442 taxis de la ville de Porto au Portugal. Le volume de données est de 1710671 individus, décrits sur 9 variables explicatives.

Voici une brève description du sens des variables descriptives (attributs de chaque trajectoire recensée) :

- trip\_id : identifiant unique du trajet
- call\_type : identifie l'origine de la requête du taxi (A = de la centrale, B = depuis un stand, C = autres)
- origin\_call : identifie le numéros de téléphone depuis lequel le client a effectué sa requête de taxi (si call\_type = A)
- origin\_stand : identifie le stand depuis lequel le client a effectué sa requête de taxi (si call\_type = B)
- taxi\_id : identifie le conducteur de taxi
- timestamp : UNIX timestamp
- daytype : identifie le type de jour du trajet (B = période de vacance, C = jour avant période de vacance, A = jour normal)
- missing\_data : si une coordonnée GPS n'est pas disponible ou non
- polyline : liste des coordonnées GPS enregistrées durant le trajet (/15 sec)

Nous voyons ici que différentes informations peuvent être mise en relation et faire sens dans un contexte professionnel. En effet on peut imaginer des analyses business effectuées par l'entreprise en

charge des taxis afin d'optimiser ses rendus. Voici des exemples de requêtes clientes que nous avons trouvé intéressantes et dont les réponses seront proposées dans nos datamarts :

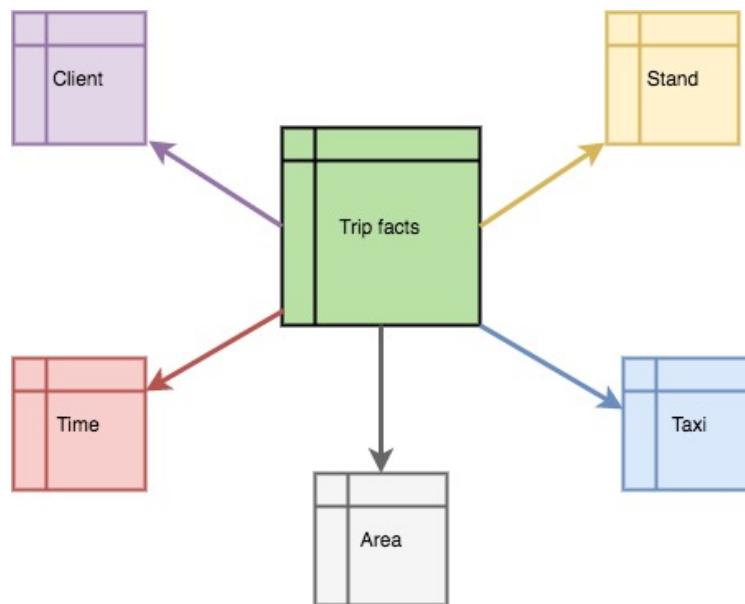
*Combien de trajets en taxi ont été réalisés le 20 avril 2017 / à quel jour de la semaine cela correspond-il ? / entre 8h et 10h30 ?*

*Quel taxi a été le plus performant le 20 avril 2017 / quelle est la distance moyenne parcourue par chaque taxi / semaine / heure / jour de la semaine ? Pour chaque taxi, quels sont les zones moyennes qu'il ou elle privilégie pour aller chercher ses clients ?*

*Quelles sont les zones de départ les plus demandées (drop in) / h / semaine / jour de la semaine ? Quelles sont les zones d'arrivée les plus demandées (drop out) / h / semaine / jour de la semaine ?*

*Quelles sont les K trajets types dans la ville de Porto ?*

Afin de prendre un peu de perspective, un premier modèle en étoile, en essayant de copier la logique SQL avec cassandra, un data-warehouse plus 'classique' serait le suivant :



Dans un tel modèle chaque dimension contiendrait l'identifiant correspondant comme primary key et serait référencé par une foreign key correspondante dans la table de fait. Ex pour la dimension taxi avec la synthaxe Cassandra :

*Dimension Taxi référençant tous les taxis de la compagnie / id : identifiant unique*

**CREATE TABLE DIM\_TAXI (id int, primary key(id));**

On voit ici l'inutilité de telles dimensions pour notre cas, car chaque dimension ne contiendrait alors que l'information identifiant ... Deplus, dans un modèle Cassandra, les jointures ne sont pas aussi simples qu'en SQL. Une jointure en Cassandra peut se faire à la main, mais leur logique n'est pas intégrée dans le modèle NoSQL que Cassandra propose. Dans un modèle en étoile comme celui-ci, basé sur des jointures entre table de faits & dimensions annexes, les requêtes auraient très peu de sens et seraient extrêmement lourdes.

Pour retrouver nos différents datamarts, nous allons plutôt chercher à dupliquer notre table des faits selon différentes partition (partition key) et sur des clustering (clustering key) qui permettent de

répondre aux requêtes proposées plus haut. Nous pourrons donc retirer toutes nos dimensions puisque toutes les réponses aux requêtes seront directement accessibles dans des ‘duplicatas’ de la table des faits.

## II. Modèle NoSQL | Cassandra

---

Précisions en premier lieu la notation.

La distance n'est pas une distance « à vol d'oiseau » entre l'origine et la destination. C'est bien une distance totale calculée sur l'ensemble de la polyline. Elle est relativement représentative et précise car la fréquence des relevés GPS nous permet d'avoir une bonne estimation de la distance parcourue (cf dist.py)

Nous utiliserons une notion de « pavés de localisation » que l'on appellera « area ». Ces areas permettront de quadriller la ville en grille. Chaque relevé GPS pourra alors correspondre à l'une de ces areas afin d'être utilisé comme partition key. En effet, il était impossible de concevoir une partition key basée sur des relevées GPS dont la précision en latitude et longitude était au 8<sup>e</sup> de décimal près. Ici nous avons choisi d'arrondir à 3 décimal. Cela formera des pavés couvrant environ 111.11 m<sup>2</sup>.

La notion o\_lon ou o\_lat correspondra à longitude et latitude de l'origine du trajet (premier relevé GPS). Parallèlement, d\_lon et d\_lat correspondra à la longitude et latitude de la destination du trajet (dernier relevé GPS).

### Trip by time

---

Cette table proposera des lignes partitionnées sur des notions de temps. Ici, nous avons choisi **annee**, **moi** et **jour** pour conserver une partition assez large. Les données seront ensuite clustérées par **heure**, par **minute** et enfin par **trip** pour proposer un clustering très fin (au trip près).

Les notions de temps ici ont été construits à partir du timestamp et de la librairie python **datetime**. La notion de **dayOfWeek** (jour de la semaine correspondant au timestamp) sera implémentée ici en simple field car nous utiliserons plus en détail cette information dans les autres tables. Les notions de locations GPS (**origin** et **destination**) ainsi que les distances totales calculées seront aussi présentes dans cette table afin de répondre à des requêtes clientes, basées sur des notions de temps, en proposant une réponse enrichie par un contexte géographique.

TRIP_BY_TIME	
Part K	annee
Part K	moi
Part K	jour
clust K	heure
clust K	minute
clust K	trip
field	dayOfWeek
field	distance
field	o_lon
field	o_lat
field	d_lon
field	d_lat

## Trip by taxi

---

Cette table proposera des lignes partitionnées sur chaque conducteur de taxi. Nous avons ici choisi un clustering des données par **dayOfWeek**, **heure** et par **trip**.

Une partition par identifiant de taxi nous permettra de connaître les informations relatives à chaque taxi. Le fait de clusteriser chaque trajet par jour de la semaine ou par heure nous permettra de connaître les ‘performances’ du conducteur en fonction de ces notions temporelles. Cela peut être utile pour des questions de maintenance de voiture, de repos du conducteur ou même de salaire.

Sont aussi présents dans cette table les notions spatiales (**origin** et **destination**) et les **distances** calculées. Ces dernières pourront aussi être des indicateurs intéressants à tirer de ce datamart. Elles peuvent en effet donner l’indication de zones moyennes où se rend et d’où part chaque taxi, ou bien la distance totale parcourue de ce taxi pour une certaine durée.

TRIP_BY_TAXI	
Part K	taxi
clust K	dayOfWeek
clust K	heure
clust K	trip
field	distance
field	o_lon
field	o_lat
field	d_lon
field	d_lat

## Trip by origin area

---

Cette table proposera une partition sur l’area d’origine. C’est-à-dire que nous utiliserons la notion de quadrillage décrite plus haut pour partitionner nos données. Le fait d’utiliser un clustering sur **dayOfWeek**, **heure** et **trip** permettra alors de tirer des informations sur les zones de départs et agir en conséquence (Notion de trafic / heure, notion de trafic / jour de semaine).

On pourrait penser à des aménagements en fonction des analyses : Plus de taxi autour des zones populaires pour une certaine heure ou un certain jour. Moins de taxis autour des zones ‘creuses’ pour une certaine heure / jour. Cela permettrait de réguler le trafic des taxis en limitant les attentes des clients et en réduisant les distances parcourues sans clients.

TRIP_BY_ORIGIN_AREA	
Part K	o_lon
Part K	o_lat
clust K	dayOfWeek
clust K	heure
clust K	trip
field	distance

## Trip by destination area

Ici, le raisonnement est le même que pour la table précédente. La partition est faite sur les relevés GPS de la destination. La notion de distance parcourue est ici encore plus intéressante.

Prenons par exemple les résultats d'une analyse factice des données de cette table qui montrerait une distance moyenne parcourue beaucoup plus élevé pour les trajets arrivant dans une zone (pavé particulier) à une certaine heure de la journée (prenons par exemple, 19h le vendredi : retour des travailleurs du centre-ville dans leur maison en périphérie). Il serait pertinent, alors, que la compagnie de taxi cherche des réductions pour 19h le vendredi, sur les prix d'essence / de maintenance dans les stations/garages proche de ces zones afin que les voitures puissent être remises en état après leur long trajet.

TRIP_BY_DESTINATION_AREA	
Part K	d_lon
Part K	d_lat
clust K	dayOfWeek
clust K	heure
clust K	trip
field	distance

## Trip by origin and destination areas

Cette dernière table proposera une partition en 4 dimensions, sur l'**origine** et la **destination** des trajets enregistrés. On retrouvera à nouveau un clustering par **dayOfWeek**, par **heure** et par **trip**.

Cette table permettra essentiellement des requêtes liées à des algorithmes de Machine Learning. Nous détaillerons cette méthode plus en profondeur dans la troisième partie.

TRIP_BY_ORIGIN_AND_DESTINATION	
Part K	o_lon
Part K	o_lat
Part K	d_lon
Part K	d_lat
clust K	dayOfWeek
clust K	heure
clust K	trip
field	distance

### III. Apprentissage de trajets | kmeans

Nous avons trouvé intéressant de chercher à tirer des modèles globaux de nos trajectoires enregistrées. En effet, le fait de connaître les informations d'origine et de destination (en coordonnées GPS) permet à des algorithmes de clustering d'extraire des clusters de nos données et, ainsi, proposer les K trajets moyens que la compagnie enregistre.

Nous avons choisi d'utiliser les kmeans (méthode des centres mobiles) car c'est un algorithme qui peut être effectué sur de gros volumes de données et proposent des performances/conclusions intéressantes. Rappelons toutefois que la méthode des kmeans ne possède pas forcément d'optimum global, mais bien des optimums locaux : il n'y a pas unicité des solutions retournées. Aussi, une bonne partition est dépendante du nombre de K (nombre de cluster recherché) choisis à l'initialisation. Nous prendrons donc deux valeurs différentes de K (4 et 7) afin de voir différents angles du modèle global retourné.

L'algorithme des kmeans permet deux choses. L'une est de trouver des clusters et leur centroïd respectif, l'autre est l'attribution des différents points à tel ou tel cluster. Ici, nous ne chercherons qu'à connaître le modèle global (les centroïds des clusters obtenus) afin d'observer les trajets types.

Rappelons aussi qu'un trajet est défini au minimum par une origine et une destination, toutes deux définies par une longitude et une latitude. Nous utiliserons donc un kmeans sur un nuage à 4 dimensions pour chercher les similitudes et donc les clusters qui en ressortent. La table **trip\_by\_origin\_and\_destination** contiendra ces points en 4 dimensions (cf partie précédente).

#### Algorithme

Voici, en pseudo code, la logique interne de mon algorithme k-means. Il ne parcourt qu'une seule fois la table et pour chaque point (ligne de la table), recalcule le centre le plus proche en se basant sur une distance euclidienne. (cf k\_means.py)

On utilisera un vector avec les centroïds ck. Dans ce vector, on retrouvera la cardinalité et les coordonnées (olon, olat, dlon, dlat) du centroïd de chaque cluster k.

Random initialisation : take 5 random points in whole table trip\_by\_o\_and\_d

For each table's row = (olon', olat', dlon', dlat'):

    Find min dist in each dist(ck, row)

    Return k

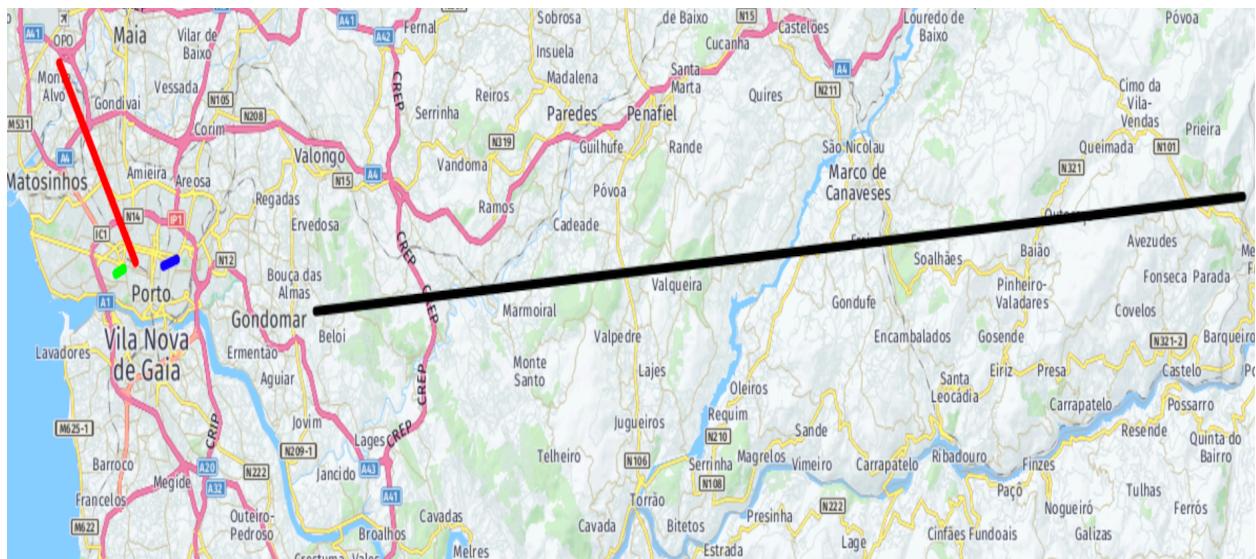
    Update ck (centroid) with ck = ((nk\*ck) + (olon', olat', dlon', dlat')) / (nk + 1)

## Résultats

Comme précisé plus haut, nous effectuerons, ici, deux partitions ( $k = 4$  et  $k = 7$ ). Pour visualiser le modèle obtenu (différents centroïds en 4 dimensions), nous utiliserons une API de visualisation sur une carte, ici celle de HERE (<https://developer.here.com/api-explorer/maps-js/v3.0/geoshapes/polyline-on-the-map>).

Nous afficherons une polyline (trait coloré) entre l'origine et la destination des centroïds de chaque cluster. L'idée étant de voir les  $k$  trajets types. Nous ferons ici une capture d'écran du site [html/javascript](#) implémenté pour visualiser (le code est dans le répo [github](#)).

### Avec 4 clusters

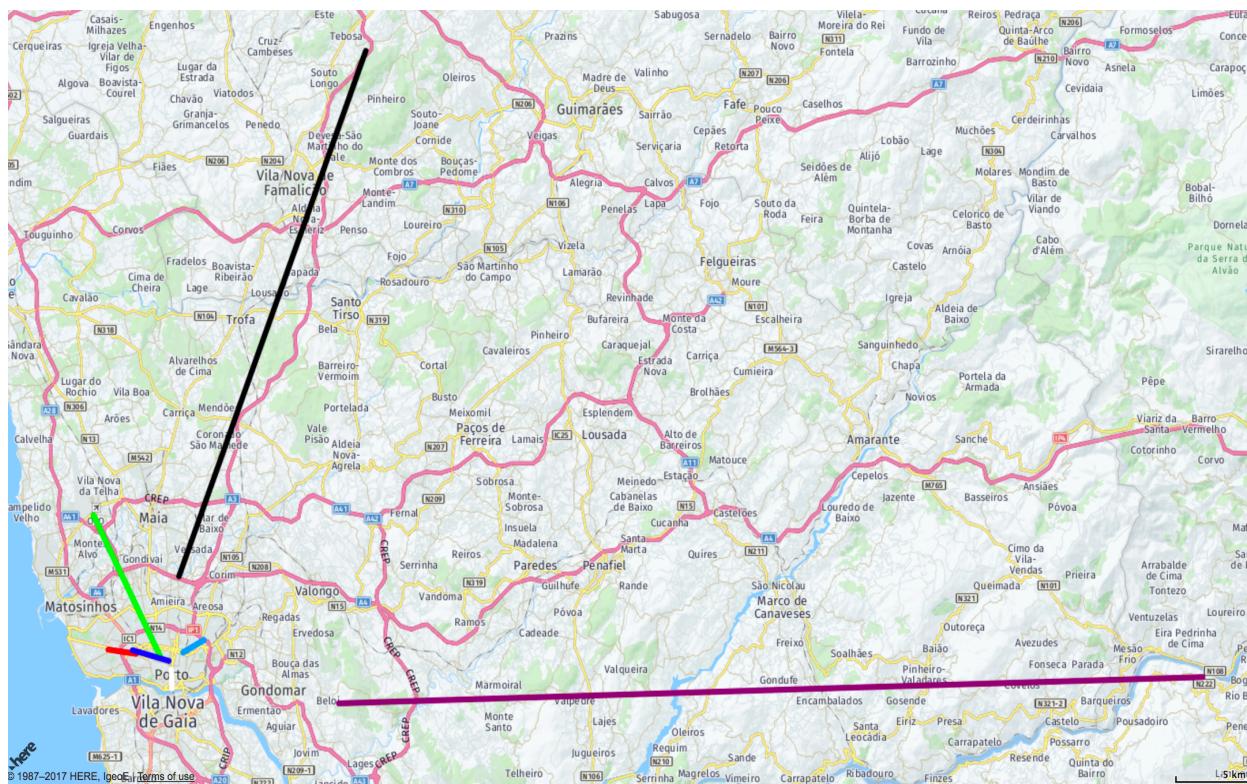


### Interprétation :

On retrouve 4 centroïds correspondant au 4 trajets types obtenus. Nous pouvons les interpréter ainsi :

- **Trajet banlieue-est / centre-ville**
- **Trajet banlieue-ouest / centre-ville**
- **Trajet aéroport / centre-ville**
- **Trajet Porto / autre ville**

## Avec 7 clusters



### Interprétation :

On retrouve 7 centroïds correspondant aux 7 trajets types obtenus. Nous pouvons les interpréter ainsi :

- **Trajet banlieue-est / centre-ville**
- **Trajet banlieue-ouest / centre-ville**
- **Trajet aéroport / centre-ville**
- **Trajet banlieue-ouest / résidence-ouest**
- **Trajet Porto / autre ville nord**
- **Trajet Porto / autre ville est**

### Conclusion

Au cours de ce TP, j'ai été amené à réfléchir aux différentes problématiques qu'impliquent le stockage de jeux de données riches et volumineux. Les questions de performances et de complexité sont à considérer et le type de base de données doit être remis en question. Ici, l'objectif était d'utiliser un modèle NoSQL CASSANDRA qui propose un modèle performant et offrant des datamarts riches en sens et en possibilité (de nombreuses requêtes clientes peuvent être répondues en utilisant les tables mis à disposition). Finalement, j'ai aussi eu l'occasion de concevoir, d'implémenter et d'appliquer un algorithme d'apprentissage automatique (clustering kmeans) sur des données géographiques afin d'en tirer des modèles généraux des données sur lesquelles j'ai travaillé.