NF26 - Rapport de projet

LE Tran Hoang Long - BRIZARD Clément

23 juin 2019

1 Introduction

Ce projet conclut la deuxième partie du cours de NF26 (*Datawarehouse et décisionnel*), dédiée aux problématiques du stockage de données en haute volumétrie.

Nous travaillons sur les données METAR (*Meteorological Aerodrome Report*) collectées dans trente-six stations entre 2005 et 2014. Le fichier pèse 477 Mo. Pour une station donnée, nous observons environ un relevé (une ligne) toutes les demi-heures. À noter qu'avant 2010, nous ne possédons de données que pour la station "EFKI".

L'objectif est de déterminer le stockage adapté pour répondre à trois questions :

1. pour un point donné, avoir un historique du passé, avec des courbes adaptées. On veut pouvoir mettre en évidence la saisonnalité et les écarts à la saisonnalité ;
2. pour un instant donné, obtenir une carte représentant n'importe quel indicateur ;
3. pour une période de temps donnée, clusteriser l'espace, et représenter cette clusterisation.

Nous voyons d'emblée que la première question relève d'une approche spatiale, et les deux autres d'une approche temporelle. Nous avons ainsi implémenté deux stockages. Le deuxième a nécessité de trouver un compromis entre les besoins des deux questions (instant donné *versus* période).

Dans ce rapport nous détaillons dans un premier temps le stockage choisi pour la première question liée à l'espace, ainsi que la stratégie choisie pour fournir une fonction répondant à la question. Nous faisons ensuite de même pour les deux questions liées à l'espace.

2 Réalisation

2.1 Exploitation spatiale

2.1.1 Technologies

Nous avons utilisé les technologies de stockage et de manipulation de données vues en cours et en TP : *Spark* et *Cassandra*.

Pour la réalisation de nos *plots*, nous avons utilisé des *packages* très connus de *Python* : *matplotlib* et *seaborn* entre autres.

2.1.2 Choix du stockage

Pour permettre des requêtes pour un point donné de l'espace, nous avons choisi d'utiliser la clé primaire suivante :

— partitionnement : `station, longitude, latitude`

— tri : `year, month, day`

Une station possède une latitude et longitude uniques. Les trois sont conservés dans la clé de partitionnement pour permettre aussi bien des requêtes par station, que par un couple `latitude-longitude`. Dans le deuxième cas, on renverra les données de la station la plus proche.

Nous obtenons ainsi autant de partitions qu'il y a de stations, à savoir trente-six. Si, comme on l'a constaté, une station réalise deux mesures par heure entre 2005 et 2014, une partition contient : $2 * 24 * 365 * 9 = 157680$ lignes. Ce n'est pas le cas cas puisqu'on a vu en 1 qu'avant 2010, nous ne possédons de données que pour une station.

2.1.3 Traitement des données

Nous stockons les données dans un *DataFrame* de *Spark*. On n'utilisera donc aucune re-

quête CQL, mais des opérations `filter` sur le *DataFrame*. Nous réalisons d'abord sur ce *DataFrame* des opérations `MapReduce` avec *Spark* pour calculer les statistiques de base (min, max, moyenne et écart-type). Grâce à *Spark*, on accède aux données qui nous intéressent nettement plus rapidement qu'en passant par des requêtes *CQL*.

Pour visualiser l'historique d'une station, nous avons décidé de proposer deux principales fonctionnalités :

- visualiser sur un intervalle d'années ;
- visualiser pour une année spécifique.

Pour chaque fonctionnalité, nous proposons différents graphes donnant un aperçu rapide et pertinent sur notre jeu de données :

2.1.4 Intervalle d'années

```
$ requete_interval(station, indicateur,
from_year, to_year)
```

Ce premier graphique permet de voir les variations de l'indicateur par mois et par année en carte de chaleur :

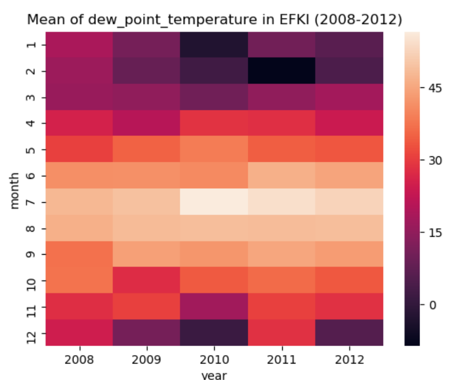

FIGURE 1 – Point de rosée de la station EFKI entre 2008 et 2012

Ici, le même indicateur, sur la même période de temps, mais avec des courbes continues représentant trois indicateurs statistiques :

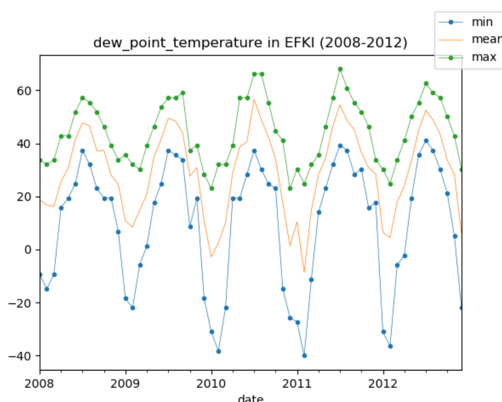

FIGURE 2 – Min, max et moyenne du point de rosée de EFKI entre 2008 et 2012

L'axe du temps est plus fin, avec une mesure par semestre. Le graphe permet notamment de voir que l'indicateur a atteint des *minima* particulièrement bas à l'hiver 2010 et 2011.

On présente ce dernier graphique composé de plusieurs sous-graphiques, pour voir la tendance d'un indicateur selon des séries temporelles [?]. Grâce à ce graphique, on voit bien que les informations sur les tendances et la saisonnalité extraites de la série semblent raisonnables. Ce indicateur (Point de rosée) pour la station EFKI semble avoir des fluctuations stables et aucune augmentation ou diminution notable. Il n'y a pas de changement climatique dans cette période.

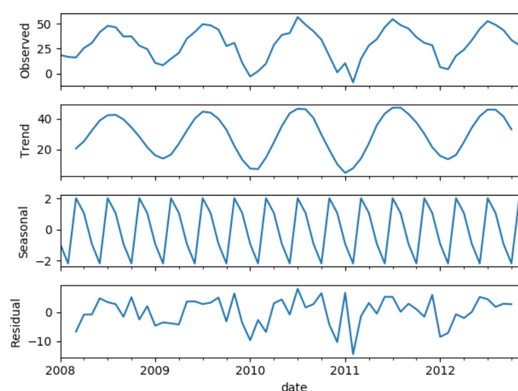

FIGURE 3 – Séries temporelles du point de rosée de EFKI entre 2008 et 2012

2.1.5 Année spécifique

`$ requete_in_year(station, indicateur, year)`

Pour une année spécifique, l'idée est de voir les changements de chaque trimestre et chaque mois :

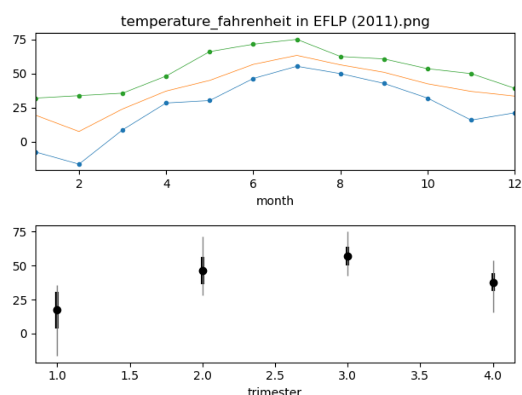

FIGURE 4 – Visualisation des températures d'une année spécifique

2.1.6 Les requêtes par longitude et latitude

`$ requete_lon_lat_interval(lon, lat, indicateur, from_year, to_year)`

`$ requete_lon_lat_in_year(lon, lat, indicateur, year)`

Un point peut-être entré par nom de station, ou par ses cordonnées GPS (longitude, latitude). On peut imaginer un utilisateur qui connaît les coordonnées d'un point géographique qui lui est familier (ex : chez lui), mais qui n'a aucune idée des stations de mesure. Notre approche est de calculer la distance de ce point avec la liste des stations dans notre jeu de données, puis trouver la station la plus proche. Les indicateurs de ce point sont donc ceux de la station la plus proche que nous ayons trouvée.

2.2 Exploitation temporelle

Technologies Mêmes *packages* que pour la visualisation statistique d'un indicateur, en ajoutant ici *Basemap* pour la visualisation spatiale.

2.2.1 Choix du stockage

Nous considérons un instant comme une heure donnée, un jour donné. Nous ne descendons pas jusqu'à la minute. À raison de deux mesures par heure, la valeur pour une station sera la moyenne des deux mesures. Notons que cela ne change finalement pas grand chose, nous aurions tout aussi bien pu rester au seuil de la minute. Par ailleurs, nous considérons une période donnée comme un intervalle entre deux dates (année, mois, jour).

Le stockage doit permettre d'interroger le moins de partitions possibles. Nous avons choisi un unique stockage temporel, avec la clé primaire suivante :

- partitionnement : `year, month` ;
- tri : `day, hour, minute`.

Initialement, nous avions inclus la minute dans la définition d'un "instant donné". Nous n'avons pas recrée la table mais cet attribut pourrait être retiré de la clé de tri.

Avec ce stockage de $9 * 12 = 108$ partitions, on accède à un instant donné (date et heure données) en interrogeant une seule partition. Dans le pire des cas, si l'instant est le dernier de la partition, on doit parcourir : $2 * 24 * 30 * 36 = 51840$ lignes[1]. Pour accéder à une période donnée, le nombre de partitions à interroger est égal au nombre de mois inclus dans la période.

On aurait pu inclure le jour dans la clé de partitionnement, ce qui n'aurait rien changé pour l'accès à un instant donné, mais aurait démultiplié le nombre de partitions à interroger pour une période donnée.

2.2.2 À un instant donné

Dans un *shell* Python :

`$ map_by_indicator_and_time indicator year-month-day hour`

On regroupe les mesures par station, on réalise la moyenne, puis on réalise un affichage avec une échelle colorée graduée :

1. 2 mesures, 24 heures, 30 jours, 36 stations

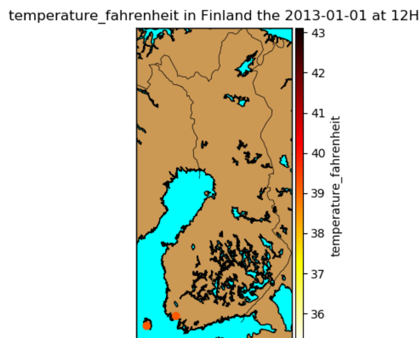

FIGURE 5 – Températures le 1er janvier 2013 à midi

On voit que même après 2010, pour une heure donnée, seules deux stations sur 36 présentent des relevés, preuve qu'elles ne suivent pas toutes le rythme d'un relevé toutes les demi-heures. Ceci nous fait penser que nous aurions pu ramener la définition d'instant à un jour donné.

Ce calcul peut être fait pour n'importe quel indicateur quantitatif.

2.2.3 Pour une période donnée

Préambule Initialement, nous considérions une période donnée comme un intervalle entre deux dates. Ceci a posé un problème. Supposons qu'on veuille un *clustering* des stations sur la période 2005-05-01 au 2006-06-01. Dans notre requête CQL, nous mettions une contrainte sur le mois pour qu'il soit compris entre 05 et 06. Problème : nous ne récupérions pas les données de 2005 au-delà du mois de juin. Il fallait que la contrainte sur le mois soit relative à l'année. Le problème se poserait de la même façon pour les jours. Il doit sans-doute pouvoir être réglé. Une solution simple à laquelle nous avons pensée aurait été de spécifier dans les arguments si la période était en années, mois (année est alors fixée) ou jour (année et mois sont alors fixés). Par manque de temps, nous avons décidé de simplifier notre définition d'une période pour la ramener à un intervalle d'années. Ce changement aurait pu impliquer de retirer le mois de la clé de partitionnement, ce qui aurait néanmoins multiplié par 12 le temps d'accès à un instant donné.

Le script est donc appelé de la manière suivante dans un *shell* Python :

```
$ cluster_by_period start_year end_year
```

Technologies Pour le *clustering* des stations, nous avons utilisé le *package* *KMeans* de la librairie Python `sklearn`.

Implémentation Dans un *shell* Python :

```
$ cluster_by_period start_year end_year
```

Nous récupérons d'abord par une requête CQL les relevés inclus dans la période renseignée en paramètre. Nous récupérons les attributs "temperature_fahrenheit", "dew_point_temperature" et "feel", qui sont tous trois quantitatifs, et permettront un *clustering* simple mais pas basé sur une seule variable. Nous "stockons" le résultat de la requête dans un générateur.

Via le *MapReduce* de *Spark*, nous réalisons la moyenne des trois indicateurs pour chaque station. Ces calculs sont parallélisés par *Spark*.

C'est sur ce nouveau jeu de données que nous appliquons ensuite l'algorithme *KMeans*. Nous utilisons ici une méthode vue en SY09 qui permet de déterminer le nombre optimal de *clusters*[2]. Nous appliquons *KMeans* pour k allant de 1 à 10, avec k le nombre de *clusters*. Nous stockons pour chaque k l'inertie intra-classe globale. On applique ensuite la "méthode du coude" sur notre tableau d'inerties, en utilisant le *package* Python `kneed`[3], disponible sur Github, et pouvant être installé *via* `pip`. Voici un exemple de graphe de "coude" :

2. Un article détaillant la méthode : https://bit.ly/2Q5IjxF
3. https://github.com/arvkevi/kneed

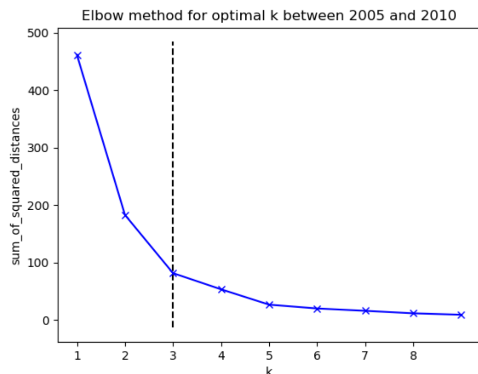

FIGURE 6 – Inertie intra-classe en fonction de k entre 2005 et 2010

Ce graphe est retourné par une méthode du *package*. On voit que le "coude" se situe à $k = 3$, c'est-à-dire qu'au-delà de trois clusters, la baisse de l'inertie intra-classe permise en augmentant le nombre de *clusters* ne "vaut pas le coup". Le *package* en question nous retourne directement cette valeur optimale après qu'on lui ait passé notre tableau d'inerties.

On applique donc *KMeans* avec k égal à la valeur renvoyée par le *package* `kneed`. On peut alors labelliser les stations, et construire une carte avec des couleurs correspondantes :

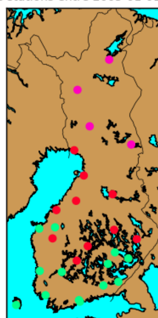

FIGURE 7 – Clustering des stations sur l'ensemble de la période disponible

Évidemment, le *clustering* fonctionne beaucoup mieux à partir de 2010 puisqu'avant nous n'avons les données que d'une station.

3 Conclusion

Nous sommes parvenus à proposer trois scripts répondant aux trois questions posées. Nous avons pu prendre conscience des problèmes liés au traitement de données très volumineuses, en témoigne la montée en température (un comble vu le sujet) inhabituelle de nos PC. Dans cette perspective, nous avons pu mesurer l'avantage d'utiliser des technologies telles que *Spark* et *Cassandra*.

Nous pourrions améliorer ce travail en corrigeant le stockage comme nous l'avons suggéré précédemment. Nous nous sommes rendus compte une fois que nous faisions les requêtes que telles ou telles données n'étaient que très peu disponibles, ou étaient plus difficiles à obtenir en CQL que ce que nous pensions. La difficulté est que pour tester ces requêtes et se rendre compte de ce qui va ou pas, il faut bien avoir déjà inséré des données, donc créé une table avec une certaine clé primaire. Ce travail a donc pour nous été un processus itératif, sans-doute l'est-ce toujours un peu, avant de trouver le bon stockage, avec les bons compromis.

Nous pourrions également fournir plus de graphes pour la question 1, avec aussi plusieurs indicateurs dans un même graphique (ex : température avec point de rosée, vitesse ou direction du vent, *etc.*)

La visualisation sur carte pourrait aussi être améliorée.

Enfin, nous pourrions enrichir les variables d'entrée de l'algorithme *KMeans*, notamment en binarisant des variables qualitatives. Ce fut cependant une satisfaction de pouvoir investir ici des méthodes vues ailleurs. Une critique serait que ce n'est pas le but de NF26, donc *quid* des étudiants n'ayant pas suivi d'enseignement de *machine learning* ? Reste que ces méthodes peuvent bien évidemment apprendre des choses sur de tels volumes de données.

Cf Annexes pour le lien du code source.

4 Annexe 1

4.1 Repository Gitlab du code source

https://gitlab.utc.fr/cbrizard/nf26-metar