**Introduction............................................................................................................................1 I. Amorçage du projet............................................................................................................1 II. Réponse aux questions.....................................................................................................2 III. Ambiguïtés identifiés et perspectives d’évolution.......................................................13 Conclusion............................................................................................................................15**

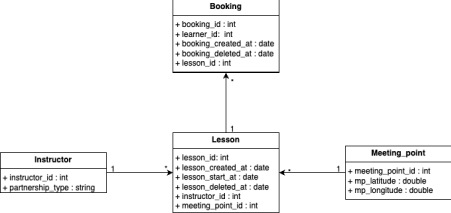
Introduction:

Ce document est un rapport de projet de Data Engineer dont la société fictive se nomme Datakar. Il nous met en face d’un cas d’étude réel , qui est l’analyse des leçons dans des établissements d’enseignement ou de formation (Auto - école) avec les instructeurs et les réservations de ses leçons .L’étude réalisée permettrait donc de suivre les périodes auxquelles les instructeurs les plus actifs enseignent, ainsi que l’évolution du nombre de cours par département dans un premier temps , et ensuite de proposer des améliorations de performance de l’infrastructure. Pour mener à bien cette étude ,les ressources utilisées ont été un entrepôt de données BigQuery, le langage de programmation Python et le langage SQL pour l’interaction avec l’entrepôt de données .

I. Amorçage du projet

On a donc procédé à la connexion à BigQuery via l’utilisation des credentials et du module python “google-cloud-bigquery .Suite à cela s’en ai suivi la modélisation du DataWarehouse dont le projet est “datakar-22042526” avec pour schéma “test\_dataset” . Le schéma ci-dessous est la représentation de ce DataWarehouse :

1. Schéma du data warehouse :



2. Explication des tables :

Le schéma ci-dessus contient 4 tables à savoir :

- *Instructor* : qui correspond aux instructeurs, il est doté de deux champs à savoir la clé primaire ainsi que d’un champ pour le type de partenariat. Un instructeur dispense une ou plusieurs leçons.

- *Booking :* correspond aux réservations effectuées par les apprenants . Elle contient l’id de réservation , l’identifiant de l’apprenant , la date de création de la réservation , éventuellement la date de suppression de la réservation ainsi que la référence de la leçon , car une réservation concerne une leçon bien précise.

- *Meeting\_point* : qui bénéficie des informations sur la géolocalisation du rendez-vous à savoir la référence unique de ce point , la latitude et la

longitude du point .Dans un point de Meeting sont dispensés une ou plusieurs lessons

- *Lesson* : qui contient les informations sur les leçons à savoir l’identifiant unique de la leçon , la date de création de la leçon , la date début de celle-ci et éventuellement une date de suppression.Une leçon est sujette à une ou plusieurs réservations et est dispensé par un instructeur dans un point de meeting précis .

II. Questions de l’Etude :

1. Analyse du jeu de donnée :

A. La date à laquelle les cinq enseignants ont donné leur 50 ème leçon sur le troisième trimestre de 2020:

a) Logique et implémentation :

Pour répondre à cette question, j’ai tout d’abord décortiqué , puis ramené aux tables et enfin j’ai ressorti la requête SQL avec l’utilisation de “ctes”(common table execution) qui convenait et je l'ai exécutée en Python. Ainsi donc on a:

- L’ensemble des leçons réservées et non supprimées qui est le premier compartiment de la requête que suit :

WITH lesson\_booked AS (

SELECT l.lesson\_id

FROM `test\_dataset.lessons` as l

INNER JOIN `test\_dataset.bookings` as b

USING (lesson\_id)

WHERE b.booking\_created\_at IS NOT NULL

AND b.booking\_deleted\_at IS NULL

),

Elle fait intervenir une jointure sur la table des leçons et celle des

réservations tout en s’assurant que la date de création est non nulle et celle de suppression nulle.

- L’ensemble des leçons effectuées au 3ème trimestre de l’année 2023 ,que constitue le second compartiment de la requête :

lesson\_with\_instructors AS (

SELECT l.lesson\_id,l.instructor\_id,l.lesson\_start\_at

FROM `test\_dataset.lessons` as l

INNER JOIN lesson\_booked as lb

ON l.lesson\_id = lb.lesson\_id

WHERE l.lesson\_created\_at IS NOT NULL

AND l.lesson\_deleted\_at IS NULL

AND l.lesson\_start\_at BETWEEN "2020-07-01" AND

"2020-09-30"

),

Elle joint les leçons aux leçons reservées tout en s’assurant que les leçons sont créées et non supprimées avec une date comprise entre le ‘01/07/2020’ et le ‘30-09-2020’. - La liste des 5 enseignants ayant dispensé le plus de cours au 3ème trimestre, le compartiment est celui qui correspond :

five\_instructors AS (

SELECT lw.instructor\_id , count(lw.lesson\_id) AS number\_lessons FROM lesson\_with\_instructors as lw

GROUP BY (lw.instructor\_id)

ORDER BY number\_lessons DESC

LIMIT 5

),

Ici on compte pour chaque instructeur son nombre de leçons du 3ème trimestre de l’année 2020 et on limite les 5 premiers par ordre décroissant .

- Le classement des leçons effectuer par les instructeurs ayant dispensés le plus de cours au 3ème trimestre de l’année 2020 qui est contenu dans le 4ème compartiment :

classify\_lessons AS (

SELECT lw.instructor\_id , lw.lesson\_start\_at ,

ROW\_NUMBER() OVER (PARTITION BY lw.instructor\_id ORDER BY

lw.lesson\_start\_at ASC) as course\_rank

FROM lesson\_with\_instructors as lw

INNER JOIN five\_instructors fi

ON lw.instructor\_id = fi.instructor\_id

)

Là on ordonne les leçons de chacun des 5 instructeurs part date de début de cours d’où l’utilisation du ‘ROW\_NUMBER() OVER ( PARTITION BY … ORDER BY … ASC) et de la jointure pour les instructeurs.

- Les dates auxquelles chacun des instructeurs ont dispensé leur 50ème leçon ,correspond au dernier compartiment :

SELECT instructor\_id , lesson\_start\_at

FROM classify\_lessons

WHERE course\_rank = 50

Ici on récupère la date et l’identifiant de l’instructeur depuis les leçons rangées dans l’ordre croissant où le rang vaut 50 .

b) Résultat

Le tableau présent représente la date à laquelle chaque instructeur à effectuer sa 50ème leçon au 3ème trimestre ainsi que identifiants de ces derniers

| Identifiant de l’instructeur Date de la 50 ème leçon |
| --- |
| 447 2020-07-22 14:00:00+00:00 |
| 499 2020-07-22 18:00:00+00:00 |
| 292 2020-07-19 17:00:00+00:00 |
| 296 2020-07-21 13:00:00+00:00 |
| 280 2020-07-22 13:00:00+00:00 |

B. Le nombre de leçons par département en fonction du type de partenariat et d’une date de début ainsi qu’une date de fin (Méthode) :

Pour répondre à cette question, j’ai tout d’abord décortiqué , puis ramener aux tables et enfin j’ai ressorti la requête SQL avec l’utilisation de

“ctes”(common table execution) qui convenait et l’ai exécuté en Python et ensuite j’ai utilisé des librairies de manipulation de données pour croiser les données et effectuer l'agrégation. L’on obtient donc :

a) Logique et implémentation

i) Pour la requête SQL :

- L’ensemble des leçons réservées et non supprimées qui est le premier compartiment de la requête que suit :

WITH lesson\_booked AS (

SELECT l.lesson\_id

FROM `test\_dataset.lessons` as l

INNER JOIN `test\_dataset.bookings` as b

USING (lesson\_id)

WHERE b.booking\_created\_at IS NOT NULL

AND b.booking\_deleted\_at IS NULL

),

Elle fait intervenir une jointure sur la table des leçons et celle des

réservations tout en s’assurant que la date de création est non nulle et celle de suppression nulle.

- L’ensemble des leçons effectuées entre la date ‘start\_date’ et ‘end\_date’,que constitue le second compartiment de la requête :

lesson\_with\_instructors AS (

SELECT l.lesson\_id,l.instructor\_id,l.meeting\_point\_id

FROM `test\_dataset.lessons` as l

INNER JOIN lesson\_booked as lb

ON l.lesson\_id = lb.lesson\_id

WHERE l.lesson\_created\_at IS NOT NULL

AND l.lesson\_deleted\_at IS NULL

AND l.lesson\_start\_at BETWEEN @start\_date AND

@end\_date

),

Elle joint les leçons aux leçons reservées tout en s’assurant que les leçons sont créées et non supprimées avec une date comprise entre les paramètres de requêtes que sont ‘start\_date’ et ‘end\_date’.

- Le 3ème compartiment correspond aux leçons effectuées ayant les types de partenariats souhaité :

partnershipping AS (

SELECT l.lesson\_id,l.meeting\_point\_id

FROM lesson\_with\_instructors as l

INNER JOIN `test\_dataset.instructors` as i

USING (instructor\_id)

WHERE partnership\_type in UNNEST(@partners)

)

Ici une jointure est effectuée sur l’identifiant de l’instructeur entre les leçons effectuées par des instructeurs sur des instructeurs ayant un partenariat inclus dans le paramètre qui est une liste de partenariats .

- Le dernier compartiment quant à lui représente le point de rencontre des leçons ayant été effectué durant une période avec des types de partenariats spécifiés :

SELECT m.mp\_latitude , m.mp\_longitude ,p.lesson\_id

FROM partnershipping as p

INNER JOIN `test\_dataset.meeting\_points` as m

USING (meeting\_point\_id)

Ici on croise les leçons effectuées ayant les types de partenariats souhaité avec le lieu de rencontre en faisant une jointure sur le ‘meeting\_point\_id’

ii) Pour le script Python :

Une fois que la requête sql est exécutée, elle renvoie le résultat dans une structure de donnée appelé dataframe du module ‘pandas’. Et les étapes suivantes sont effectués :

- Une transformation en DataFrame ‘PySpark’ pour des raisons de traitements parallèles au vue de la taille des données

- Un partitionnement du cluster proportionnelle à la taille du dataframe - L’exécution d’une méthode qui permet de récupérer le département en fonction des coordonnées d’un point de rencontre et ceci via la

consommation de l’API ayant pour url :

**"https://api.opencagedata.com/geocode/v1/”**

- Ensuite un partitionnement selon les départements pour des raisons de performance suivie d’un cache

- Une sélection de la colonne département

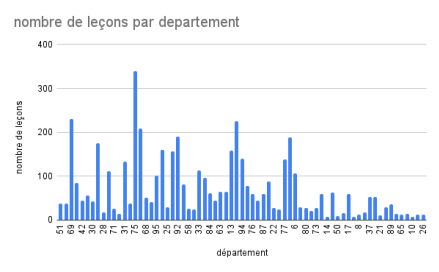
- Une répartition des départements sur un noeud du cluster en vue de l’écriture du résultat sur disque.

b) Résultat :

A titre d’exemple :

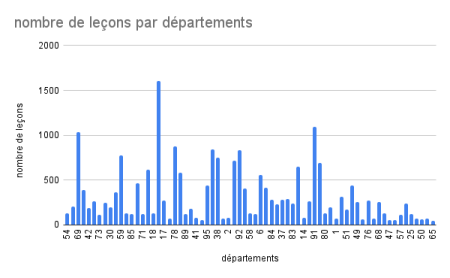
- Pour les types de partenariats SARL, SAS et ME entre le 30/01/2020 et le 06/02/2020 on obtient un total de 4853 leçons concernées par 69

départements. Le diagramme en colonne suivant illustre la répartition des leçons par départements (sur le diagramme uniquement 35 départements sont affichés pour lisibilité) :



- Pour les types de partenariats SARL, SAS et ME entre le 05/01/2020 et le 31/05/2020 on obtient un total de 21980 leçons concernées par 69

départements. Le diagramme en colonne suivant illustre la répartition des leçons par départements (sur le diagramme uniquement 35 départements sont affichés pour lisibilité) :



2. Optimisation des performances

A. Les limites identifiées dans le précédent script et les propositions d’optimisation

a) Les limites identifiés dans le précédent script :

Ces limites sont les suivantes :

- La requête SQL renvoie un grand nombre de lignes due aux différents créneaux

- La requête SQL contient plusieurs jointures (3)

- La requête SQL peut devenir lente avec le large volume de donnée - La requête SQL est plus ou moins complexe , plusieurs jointures - Lorsqu’on la réexécute on parcours tous les données.

- Côté python pyspark , l’utilisation du groupBy pour grouper par département est très coûteux malgré que l’appel d’API est rapide pour le croisement de données.

- Côté répartir les données sur un noeud pour l’écriture est coûteux

b) Les propositions de solution :

Il y’en a 3 des propositions :

i) Utilisation de BigQuery , Pandas , Pyspark ,API REST ,SQL

Concernant le SQL , c’est à dire la requête il faudrait :

- Créer d’index sur les dates (début, fin) afin d’améliorer les

performances et d’optimiser les jointures.

- La réduction de la latence , en sauvegardant le résultat de la précédente requête avec les 3 jointures dans une vue matérialisée. - En encapsulant des opérations répétitives ou coûteuses dans une procédure SQL, avec les paramètres que l’on passe.il permettrait de réduire le nombre d'allers-retours entre BigQuery et python .

- Si la machine ne peut supporter le large volume créer un OFFSET dans la requêtes

- Faire un trigger pour rajouter les nouvelles données dans la procédure

Concernant le traitement avec python , il faudrait :

- Récupérer les données sous dataframe pandas les convertir en dataframe pyspark

- Faire des appels d’api rest avec pyspark pour le croisement des départements

- Envoyer la colonne département dans une base de données SQL (une base de donnée NoSQL peut également jouer voir être plus optimale) - Effectuer l'agrégation (le groupBy suivie du count ) sous forme de requête SQL

- Retourner le résultat sous dataframe pandas

- Sauvegarder le résultat dans un fichier csv

Le schéma suivant illustre la procédure :

ii) L’utilisation de BigQuery , Pyspark, API REST :

Deux approches dans ce cas :

● 1ère Approche :

- Récupérer les tables sur une machine grâce au client

‘google-cloud-bigquery’ sous format csv.

- Lire les tables au formats csv dans des dataframes spark

- Ajouter la colonne département à la table ‘meeting\_point

- Appliquer les jointures et le groupBy au dataframe

- Ecrire la dataframe dans un fichier csv.

- Supprimer les tables une fois le traitement terminé

- Reprendre le processus

- Mettre à jour le nombre de leçons par département en sommant avec les leçons précédemment utilisé.

● 2ème Approche :

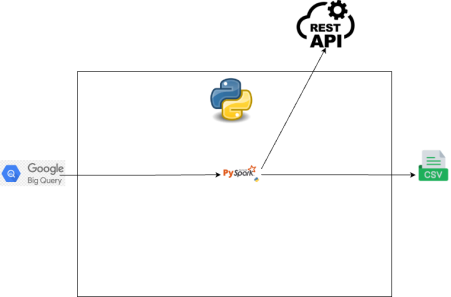
Elle est similaire à la première sauf qu'à la place de la récupération sous

format csv on lit directement les tables BigQuery dans des dataframes

pandas et le processus est le même. Il ya la possibilité de lancer des

procédures pour les nouvelles données ( nouveaux créneaux) .

Le schéma suivant illustre la procédure :



iii) L’utilisation de BigQuery, Kafka ,Pyspark, fichier csv :

Ici on utiliserait de la manière suivante beaucoup plus pour les nouvelles

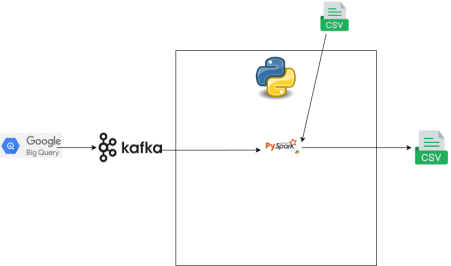
données :

- Utiliser du KafkaConnect pour récupérer les données depuis BigQuery

- Appliquer les traitements avec pyspark , notamment le croisement avec un fichier csv contenant les latitudes et longitudes des départements français ( les points les plus au nord, les plus au sud , les plus à l’ouest , les plus à l’est)

- Sauvegarder le résultat les données dans un fichier csv

Le schéma suivant illustre la procédure :



Il est important de noter que le traitement par lot est à utiliser pour faire face à la volumétrie notamment 100000000 de créneaux .

B. Proposition de la mise en production du modèle qui labellise les créneaux en fonction de la probabilité d’être réservé :

En considérant que le modèle est déjà entraîné sur des millions de données , les éléments à suivre sont nécessaire pour la mise en production :

- La Mise en place d’un pipeline de données en temps réel pour envoyer les nouveaux créneaux , lorsque ceux-ci sont créés en base de données. A titre d’exemple Kafka peut remplir la tâche.

- L’Utilisation d’un moteur de traitement de données en temps réel pour effectuer des prétraitements nécessaires et appliquer le modèle de

prédilection sur les nouveaux créneaux. Apache Spark rempli bien la tâche - Stocker les horaires labellisés dans une base de données NoSQL , à l'instar de Apache Cassandra pour stocker les données traitées ,les labels assignés et notamment bénéficier de scalabilité.

- Conteneurisé et déployé le modèle de production dans un outil

d’orchestration à l’instar de Docker pour la conteneurisation et de Kubernetes pour l’orchestration

- Tester , déployer et intégrer en continue le modèle grâce à des outils de CI/CD Tels que GitLabCI

- Exposer le modèle via une API pour être utilisable par les autres composants de l'architecture. Il peut être fait en FastAPI

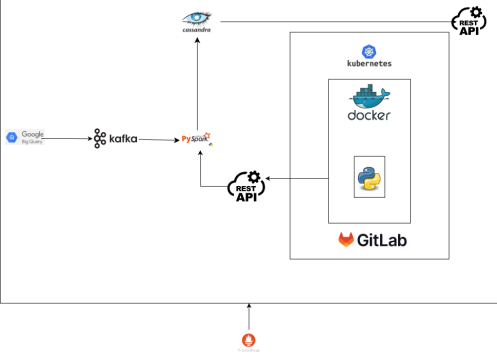
- Une création également une API qui va être connecté à cassandra pour lire les labels

- La mise sur pieds de monitoring la solution en temps réel la solution avec des outils comme Prometheus pour la surveillance des performances et

l’alerting en temps réels

- La réalisation de tests sur l’ensemble de l’infrastructure.

Le schéma ci-dessous illustre ses propos :



III. Ambiguïtés identifiés et perspectives d’évolution

1. Ambiguïtés identifiés:

A.Problématiques mineures :

Au début du projet j’ai eu à faire face à un problème concernant le nom d’une table , il s’agit de la table ‘meeting\_points’ , en effet il s’agit de

‘meeting\_points’ et pas de ‘meeting\_point\_location’ Pour corriger de l’erreur j’ai utilisé la requête :

'SELECT table\_name FROM `test\_dataset.INFORMATION\_SCHEMA.TABLES` ' Qui permet d’avoir l’ensemble des tables du projet. Une autre problématique a été lié à une librairie mais ,elle s’est vue réglé

B. Problématiques majeures

La problématique majeure a été le croisement des départements. En effet, ayant d’abord commencé par utiliser un fichier csv de géolocalisation des départements fourni par le gouvernement ,pour trouver les départements associés au point de rencontre. Je me suis rendu compte de l’impertinence de certains résultats car certains points de rencontre semblaient appartenir à plus d’un département, ce qui est absurde.Ainsi je me suis rabattu sur l’utilisation d’une API pour la récupération des départements en fonction de la latitude et de la longitude. Et ainsi je me suis lancé dans le traitement avec pandas de bout en bout pour manipuler la donnée , et je me suis très vite aperçu de sa lenteur pour effectuer les appels d’API vu qu’il est monothread. Ainsi donc je me suis lancé dans une étude comparative des modules de traitement de donnée en parallèle à savoir “polars” , “dask” et “pyspark” et pyspark en ait sorti largement vainqueur, comme le montre le tableau ci-dessous:

| Module de  manipulation de donnée | Pandas | Dask Polars Pyspark |
| --- | --- | --- |
| Temps  d’exécution  pour 500  meeting\_points en secondes | 1,32 | 1,37 1,34 10 |

Dans la continuité , pour le second cas j'ai fait face à un énorme souci de performance. En lorsque le résultat de la requête SQL en vient à être traiter pour l'agrégation après le croisement de donnée via l’utilisation de l’API , le cluster spark fait face au shuffling(le fait de ramener les données d’une partition vers d’autres et vice-versa) d’innombrable donnée et cela prend du temps. A titre d’exemple le tableau suivant :

| Nombre de  leçons | Environ 100 | Environ 4000 Environ 80000 Environ 240000 |
| --- | --- | --- |
| Temps de  traitement | Moins de 30  secondes | Environ 2  Environ 20  Plus d’une  minutes  minutes  heure |

Et ses résultats sont après l’application de répartition et de cache.

Une expérimentation a été effectué pour un second avis, celle ci consiste à récupérer toutes les tables du projets , croiser la table ‘meeting\_point’ avec les départements puis réécrire la table ‘meeting\_point’ avec la nouvelle colonne et effectuer la requête directement en spark, on obtient un résultat beaucoup plus rapide moins de 2 minutes voir 1 minute pour plus de 330000 leçons. En considérant que les ‘meeting\_points’ sont infinis on pourrait procéder à un traitement par lot.

2. Axes d’amélioration

Il concerne deux points à savoir :

A. L’optimisation des performances :

Etre en mesure d’examiner le code , voir de consulter les DAG pour trouver la meilleure façon d’agencer les partitionnements pour pallier à la latence due au croisement et au groupBy, s’il n’en ai rien éventuellement passer sur DataBricks.

B. L’optimisation du code :

Bien modulariser les classes , faire ressortir les méthodes qui ne sont pas en lien avec la classe et créer une nouvelle classe. Je parle de méthodes telles que les vérifications ,les copies de fichiers … Procéder à une meilleure couverture de tests , surtout écrire les tests pour les méthodes correspondant aux réponses aux questions .

Conclusion:

Ce projet fut très intéressant car il à mis en exergue la communication entre un DataWarehouse et un outil de traitement massif , afin de réaliser une étude sur un cas concrèt .