Test Python DE
Proposition Anas Kezibri

I) Python et Data Engineering

Ce document présente la proposition pour le développement d'un data pipeline qui traitera des données liées aux médicaments, articles PubMed et essais cliniques, dans l'objectif d'établir un graphe de liaisons entre ces entités.

J'ai opté pour la stack technique suivante pour atteindre la solution souhaitée :

- Le **framework Python Kedro** pour l'implémentation du data pipeline,
- **Docker** pour la conteneurisation et l'encapsulation des dépendances,
- **Airflow** pour l'orchestration des tâches,
- Et enfin **Git** pour le versionning du code.

La sortie du pipeline est un fichier JSON formatée pour s'intégrer efficacement dans une base de données orientée graphe type Neo4j, permettant ainsi une représentation claire et intuitive des relations entre les médicaments et les journaux en passant par les articles PubMed et essais cliniques.

Justification des Choix Technologiques

1. Kedro

Kedro est un framework open-source pour la création de pipelines de données en Python. Ci-après les raisons pour lesquelles j'ai choisi Kedro :

- **Modularité et Structure** : Kedro encourage une architecture modulaire, facilitant la gestion de grandes bases de code. Cela permet de construire, tester et réutiliser les pipelines dans différents projets, réduisant ainsi la complexité et le temps de développement.
- **Scalabilité** : Kedro est scalable pour de grands volumes de données grâce à sa modularité, permettant le découpage et l'exécution parallèle des pipelines. En outre, il s'intègre avec des moteurs distribués comme Apache Spark pour traiter efficacement des téraoctets de données. Kedro gère intelligemment les datasets avec des formats optimisés et le chargement à la demande. De plus, il peut être déployé sur des infrastructures cloud scalables et s'intègre avec des orchestrateurs comme Airflow pour une gestion optimale des ressources.
- **Pratiques de Développement** : Il propose aussi des bonnes pratiques intégrées pour le développement, comme la gestion des données, le versioning, facilite la collaboration et l'implémentation des différentes étapes du pipeline en même temps et le test des pipelines, assurant ainsi une qualité de code élevée.

2. Docker

Docker crée des conteneurs légers qui encapsulent l'application et ses dépendances. Ses avantages incluent l'isolation des environnements pour éviter les problèmes de compatibilité, une facilité de déploiement dans différents environnements, et une scalabilité permettant d'adapter les conteneurs à des charges de travail plus importantes.

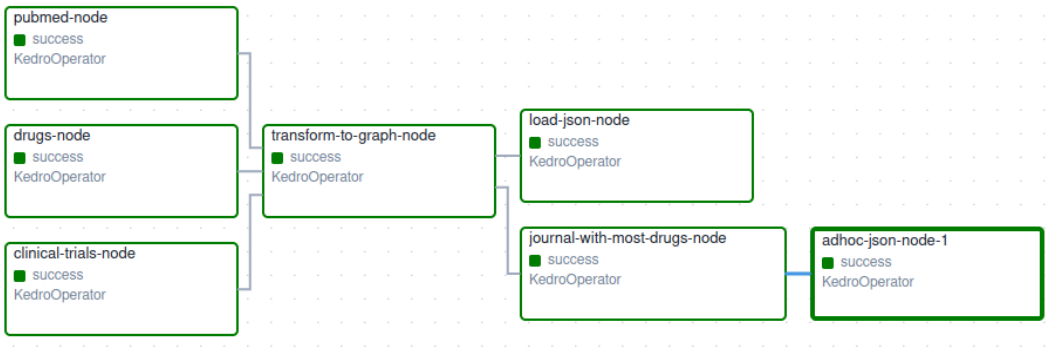
3. Airflow

Airflow permet d'orchestrer et d'automatiser les tâches avec une gestion claire des dépendances entre les étapes. Son interface intuitive facilite la visualisation et le débogage des pipelines, et il est hautement extensible, permettant d'ajouter de nouvelles fonctionnalités facilement.

Structure du projet

Les composants principaux du data pipeline sont :

- **Gestion des données :** La définition des données d'entrée, de traitements intermédiaires et de sortie dans le fichier Catalog. Ce fichier indique également l'emplacement de ces données sous le dossier data. Le graphe de liaison est généré sous le dossier data/03_output.
- **Traitement des données :** Le dossier src contient le code source du pipeline :
    - la définition des étapes du pipeline : extrait de données et preprocessing (Nettoyage de données : Supprime les lignes contenant des valeurs vides ou espaces, formatage de la date, concaténation des données PubMed), puis ensuite la transformation et enfin le chargement vers de fichiers JSON. Chaque étape est un fichier .py sous le dossier pipeline/nodes, sous forme de fonctions qui peuvent être reutilisées dans d'autres pipelines.
    - La définition de l'enchaînement de ces étapes et leurs données d'entrée et de sortie sont définies sous le fichier pipeline.py.
- **Orchestration :** Le dossier airflow-drugs-data-pipeline représente le projet Airflow exécuté en utilisant Astronome, qui est une plateforme gérée pour Airflow qui permet de gérer des clusters en production et facilite le démarrage local.
- **Packaging et containerisation** : Le code source du data pipeline est dockerisé et packagé en utilisant Python.



Les étapes du data pipeline orchestré sur Airflow

Les étapes pour exécuter le data pipeline en utilisant le DAG Airflow sont décrites sur le Readme du repository Git. Si le projet est exécuté en utilisant Astronomer, les fichiers json de sortie seront disponible sur le conteneur Docker.

Format de Sortie JSON

L'objectif du pipeline est de produire un fichier JSON qui représente un graphe de liaison entre les médicaments et leurs mentions dans différents journaux. J'ai choisi de construire un objet JSON qui suit la logique de la base de données orientée graphe Neo4j, car ce dernier est conçu pour analyser des relations complexes entre des entités.

Neo4j utilise une structure de **graphe** où les **nœuds** représentent les entités et les **relations** (arêtes) définissent les connexions entre ces nœuds.

1. Représentation des données

- **Nœuds** :

Trois types de nœuds :

Médicament

```
{
    "id": "A01AD",
    "label": "Drug",
    "properties": {
        "name": "EPINEPHRINE"
    }
}
```

Publication

```
{
    "id": "7",
    "label": "Publication",
    "properties": {
        "title": "The High Cost of Epinephrine Autoinjectors and Possible Alternatives.",
        "article_type": "PubMed",
        "date": "2020-01-02"
    }
}
```

Journal

```
{
    "id": "JID_the_journal_of_allergy_and_clinical_immunology._in_practice",
```

```
    "label": "Journal",
    "properties": {
        "name": "The journal of allergy and clinical immunology. In practice"
    }
}
```

- **Relations** :

Quatre types de relations.

Entre médicament et publication :

```
{
    "from": "A01AD",
    "to": "7",
    "type": "REFERENCE"
}
```

Entre publication et journal :

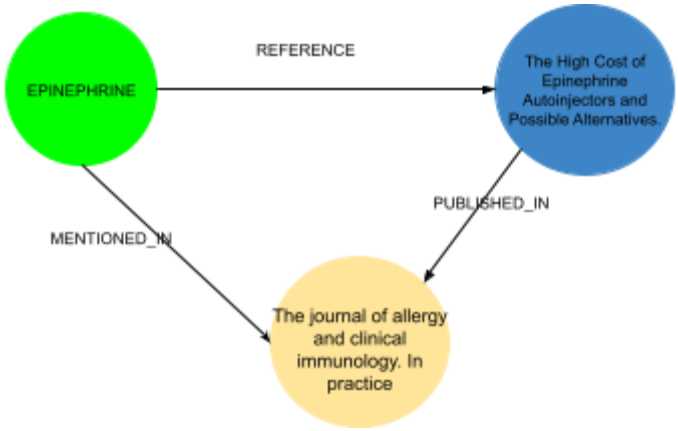
```
{
    "from": "7",
    "to": "JID_the_journal_of_allergy_and_clinical_immunology._in_practice",
    "type": "PUBLISHED_IN",
    "properties": {
        "publication_date": "2020-01-02"
    }
}
```

Entre journal et medicament

```
{
    "from": "A01AD",
    "to": "JID_the_journal_of_allergy_and_clinical_immunology._in_practice",
    "type": "MENTIONED_IN",
    "properties": {
        "mention_date": "2020-01-02"
    }
}
```

- **Graphe** :

Donc en combinant les objets JSON ci-dessus, on obtient le graphe de liaison suivant :



Scalabilité de la solution

Dans cette dernière partie, je donne une explication sur les éléments à considérer pour faire évoluer cette solution en cas de grosse volumétrie de données.
Pour pouvoir adapter notre solution, on va tirer parti de systèmes de traitement distribué, de stockage efficace et d'une gestion appropriée des ressources. Voici ce qu'on peut envisager comme modifications :

1. Utilisation de stockage distribué pour les datasets

- Configurer les datasets Kedro pour utiliser des backends de stockage distribué comme **S3** ou **HDFS**, en spécifiant ces emplacements dans le fichier `catalog.yml`.
- Lire les fichiers volumineux en **chunks** pour éviter de surcharger la mémoire. Cela permettra aussi d'accélérer la lecture/écriture et de traiter les données en parallèle si on distribue les chunks sur différents nœuds.

Exemple :

```
large_dataset:
  type: pandas.CSVDataSet
  filepath: s3://my-bucket/path/to/data.csv
  credentials: s3_creds
  load_args:
    chunksize: 100000  # Lire les données en chunks
```

2. Utilisation des Partitioned et Incremental Datasets

Le **PartitionedDataset** de Kedro permet de gérer efficacement de gros volumes de données en chargeant et sauvegardant de manière récursive des fichiers uniformes à partir de divers systèmes de fichiers (local, S3, GCS, etc.). Grâce à son approche de **lazy loading**, il ne charge les partitions de données que lorsqu'elles sont explicitement demandées, optimisant ainsi la mémoire et les ressources. De plus, le **lazy saving** via des **Callables** permet de différer la

persistance des données, ce qui le rend particulièrement adapté aux environnements distribués et aux pipelines de traitement massifs.

Pour adopter cette amélioration dans notre solution, il va falloir modifier le `catalog.yml`, pour specifier la logique de no partition puis ensuite modifier les étapes du pipeline pour introduire la logique de concatenantion des partitions de données.

Plus de details : Advanced: Partitioned and incremental datasets — kedro 0.19.8 documentation.

3. Traitement Distribué avec Apache Spark

- Intégrer **Apache Spark** via le plugin `kedro-spark` pour gérer le traitement distribué et en parallèle des données massives, en configurant des datasets Spark pour charger et traiter les données directement dans un environnement distribué.

Documentation officielle : PySpark integration — kedro 0.19.8 documentation.

II) SQL

1. Le chiffre d'affaires jour par jour

```sql
SELECT
 date AS date,
 SUM(prod_price * prod_qty) AS ventes
FROM TRANSACTIONS
WHERE
 date BETWEEN '2019-01-01' AND '2019-12-31'
GROUP BY date
ORDER BY date;
```

2. Les ventes meuble et déco réalisées par client

```sql
SELECT
 T.client_id AS client_id,
 SUM(CASE
         WHEN P.product_type = 'MEUBLE' THEN T.prod_price * T.prod_qty
              ELSE 0
         END) AS ventes_meuble,
 SUM(CASE
         WHEN P.product_type = 'DECO' THEN T.prod_price * T.prod_qty
              ELSE 0
         END) AS ventes_deco
FROM TRANSACTIONS T
JOIN PRODUCT_NOMENCLATURE P ON T.prod_id = P.product_id
WHERE T.date BETWEEN '2019-01-01' AND '2019-12-31'
GROUP BY T.client_id
ORDER BY T.client_id;
```