Rapport de Projet : Data Engineering avec Apache Spark

Par Jean Marie Fande NDIAYE

Enseignant : Dr. Djibril Mboup

Cours : BigData – ING3 GEIT IPSL 2024

Table des matières

1. Introduction

1.1 Contexte

Le projet de fin d'études en Data Engineering consiste à créer un pipeline ETL (Extract, Transform, Load) en utilisant Apache Spark. Le pipeline vise à charger, transformer et sauvegarder des données volumineuses dans un environnement distribué, en tirant parti des capacités de traitement massivement parallèle offertes par Spark.

1.2 Objectifs

L'objectif principal de ce projet est de développer un pipeline ETL capable de :

- Charger les données brutes depuis des fichiers JSON.
- Nettoyer et transformer les données pour en extraire des informations pertinentes.
- Calculer des statistiques importantes telles que le revenu par source de trafic.
- Sauvegarder les données transformées dans un format de stockage approprié pour une analyse future.

2. Préparation de l'Environnement

2.1 Configuration de Spark

Le projet utilise Apache Spark version 3.5.1, une version compatible avec les dernières fonctionnalités et optimisations disponibles dans Spark. Le développement a été réalisé en utilisant Scala 2.12, un langage de programmation fonctionnel performant pour le traitement de données en masse.

2.2 Outils Utilisés

- **IDE :** IntelliJ IDEA avec Scala Plugin
- **Gestionnaire de versions :** Git pour le suivi des modifications du code source.
- **Outil de build :** SBT (Scala Build Tool)
- **JDK :** OpenJDK 17
- **Spark :**version 3.5.1
- **Système de gestion de versions distribuées :** Git

2.3 Commandes de Base

Pour exécuter le projet dans l'environnement local, les commandes suivantes ont été utilisées :

- **Compilation du projet :** `sbt compile`
- **Exécution des tests :** `sbt test`
- **Génération d'un JAR exécutable :** `sbt package`

3. Conception du Pipeline ETL

3.1 Structure du Code

Le pipeline ETL a été conçu en trois classes principales :

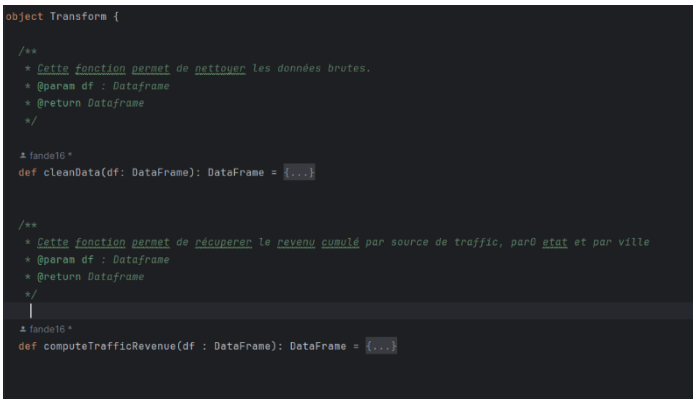
- **Extract.scala :** Cette classe est responsable de charger les données depuis une source donnée, ici un fichier JSON.
- **Transform.scala :** Cette classe gère la transformation des données brutes, y compris le nettoyage des données, le calcul des revenus par source de trafic, etc.
- **Load.scala :** Cette classe s'occupe de la persistance des données transformées dans le système de fichiers local sous format Parquet.

3.2 Classe Extract

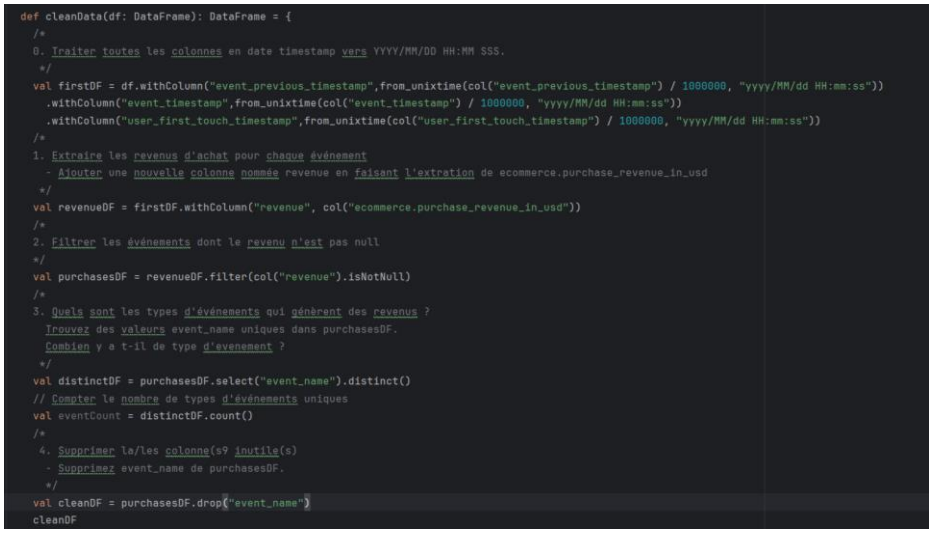
La classe `Extract` contient une méthode principale `read_source_file`, qui lit un fichier JSON et retourne un DataFrame. La méthode utilise Spark pour inférer automatiquement le schéma des données.

3.3 Classe Transform

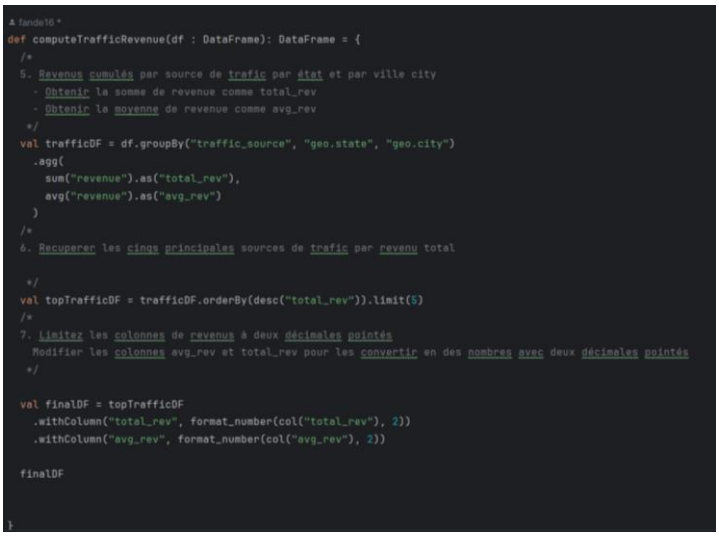
La classe `Transform` est responsable de deux fonctions principales :



1. **cleanData :** Cette fonction nettoie les données en normalisant les colonnes de date et en filtrant les événements avec des revenus non nuls.



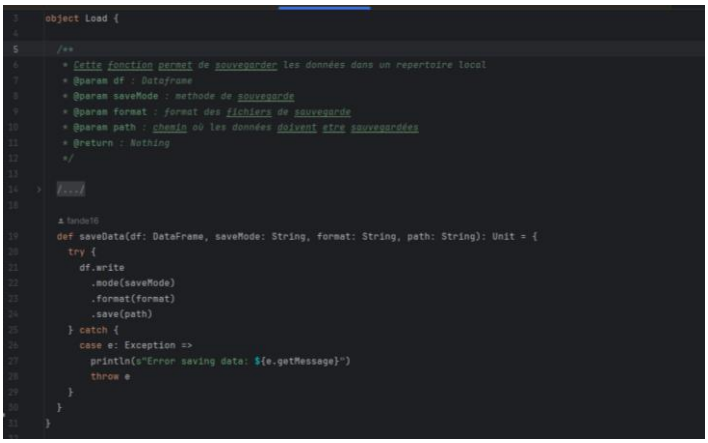
2. **computeTrafficRevenue :** Cette fonction calcule le revenu total et moyen par source de trafic, état et ville.



```
def computeTrafficRevenue(df : DataFrame): DataFrame = {
  /*
  5. Revenus cumulés par source de trafic par état et par ville city
    - Obtenir la somme de revenue comme total_rev
    - Obtenir la moyenne de revenue comme avg_rev
   */
  val trafficDF = df.groupBy("traffic_source", "geo.state", "geo.city")
    .agg(
      sum("revenue").as("total_rev"),
      avg("revenue").as("avg_rev")
    )
  /*
  6. Recuperer les cinqs principales sources de trafic par revenu total

   */
  val topTrafficDF = trafficDF.orderBy(desc("total_rev")).limit(5)
  /*
  7. Limitez les colonnes de revenus à deux décimales pointés
    Modifier les colonnes avg_rev et total_rev pour les convertir en des nombres avec deux décimales pointés
   */

  val finalDF = topTrafficDF
    .withColumn("total_rev", format_number(col("total_rev"), 2))
    .withColumn("avg_rev", format_number(col("avg_rev"), 2))

  finalDF


}
```

3.4 Classe Load

La classe `Load` sauvegarde les données transformées sous format Parquet dans le chemin spécifié.

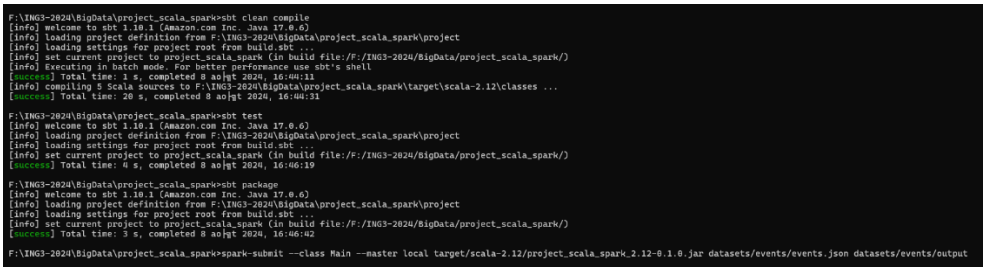


```
object Load {

  /**
   * Cette fonction permet de sauvegarder les données dans un repertoire local
   * @param df : Dataframe
   * @param saveMode : methode de sauvegarde
   * @param format : format des fichiers de sauvegarde
   * @param path : chemin où les données doivent etre sauvegardées
   * @return : Nothing
   */

  /....

  def saveData(df: DataFrame, saveMode: String, format: String, path: String): Unit = {
    try {
      df.write
        .mode(saveMode)
        .format(format)
        .save(path)
    } catch {
      case e: Exception =>
        println(s"Error saving data: ${e.getMessage}")
        throw e
    }
  }
}
```

4. Exécution du Pipeline

4.1 Compilation, test et Build du code

Le code source a été compilé, testé et buildé afin de générer le fichier jar.



```
F:\ING3-2024\BigData\project_scala_spark>sbt clean compile
[info] welcome to sbt 1.10.1 (Amazon.com Inc. Java 17.0.6)
[info] loading project definition from F:\ING3-2024\BigData\project_scala_spark\project
[info] loading settings for project root from build.sbt ...
[info] set current project to project_scala_spark (in build file:/F:/ING3-2024/BigData/project_scala_spark/)
[info] Executing in batch mode. For better performance use sbt's shell
[success] Total time: 1 s, completed 8 ao▯t 2024, 16:44:11
[info] compiling 5 Scala sources to F:\ING3-2024\BigData\project_scala_spark\target\scala-2.12\classes ...
[success] Total time: 20 s, completed 8 ao▯t 2024, 16:44:31

F:\ING3-2024\BigData\project_scala_spark>sbt test
[info] welcome to sbt 1.10.1 (Amazon.com Inc. Java 17.0.6)
[info] loading project definition from F:\ING3-2024\BigData\project_scala_spark\project
[info] loading settings for project root from build.sbt ...
[info] set current project to project_scala_spark (in build file:/F:/ING3-2024/BigData/project_scala_spark/)
[success] Total time: 4 s, completed 8 ao▯t 2024, 16:46:19

F:\ING3-2024\BigData\project_scala_spark>sbt package
[info] welcome to sbt 1.10.1 (Amazon.com Inc. Java 17.0.6)
[info] loading project definition from F:\ING3-2024\BigData\project_scala_spark\project
[info] loading settings for project root from build.sbt ...
[info] set current project to project_scala_spark (in build file:/F:/ING3-2024/BigData/project_scala_spark/)
[success] Total time: 3 s, completed 8 ao▯t 2024, 16:46:42

F:\ING3-2024\BigData\project_scala_spark>spark-submit --class Main --master local target/scala-2.12/project_scala_spark_2.12-0.1.0.jar datasets/events/events.json datasets/events/output
```

4.2 Commande d'Exécution

Le pipeline est exécuté en utilisant la commande `spark-submit`. Voici la commande utilisée :

```
spark-submit --class Main --master local target/scala-2.12/project_scala_spark_2.12-0.1.0.jar datasets/events/events.json datasets/events/output
```

4.3 Résultats de l'Exécution

Les données ont été chargées, nettoyées, et transformées avec succès. Les données résultantes ont été sauvegardées dans un fichier Parquet dans le répertoire spécifié.

traffic_source	state	city	total_rev	avg_rev
google	LA	Shreveport	2,949.00	1,474.50
google	TX	Ovilla	2,840.00	2,840.00
email	MO	Moberly	2,740.00	2,740.00
google	RI	Warwick	2,290.00	2,290.00
email	CA	Sacramento	2,061.00	2,061.00

6. Conclusion

Ce projet a permis de comprendre et de mettre en œuvre un pipeline ETL en utilisant Apache Spark. Le pipeline est capable de traiter de grandes quantités de données, de les nettoyer, de calculer des statistiques importantes, et de les sauvegarder pour une analyse future. Les défis rencontrés au cours du projet ont été résolus en adaptant la configuration et en ajustant le code pour qu'il soit compatible avec l'environnement utilisé.

7. Annexes

- **Code Source :** Disponible sur le dépôt public GitHub lien_du_depot .
- **Données d'Entrée :** Fichiers JSON utilisés pour tester le pipeline.
- **Dépôt Git :** Journal des commits et historique du projet.