Exo4 partie 1

Démarche adoptée

On a débuté par chargé les données depuis un fichier CSV en utilisant spark.read.csv, en prenant soin d'activer les options header pour interpréter correctement la première ligne comme les noms de colonne, et inferSchema pour que Spark déduise automatiquement les types de données des différentes colonnes.

Maintenant voici la démarche que nous avons suivi pour comparer la performance entre une UDF Python et une UDF Scala effectuer cette comparaison de manière précise :

1. Préparation des Données

Les deux scripts utilisent exactement le même jeu de données pour garantir une comparaison équitable. La taille du jeu de données est suffisamment grande pour que les différences de performances soient perceptibles.

2. Mesure du Temps d'Exécution

Pour chaque script (Python UDF et Scala UDF), nous voulons mesurer le temps d'exécution de la partie critique du code, c'est-à-dire, l'application de l'UDF et toute opération de traitement de données subséquente jusqu'à la génération du résultat final.

Méthodes de Mesure:

- Utilisation de .show() : Nous n'avons vu aucune différence d'affichage entre les deux scripts car show() sur un petit nombre de lignes (par défaut 20) ne force pas Spark à traiter tout le DataFrame. Ce n'est donc pas une bonne mesure de performance.

```
currently activated Python version 3.11.4 is not supported by the project (~3.10.12). in the project (~3.10.12) in the project (~3.10.12) is not supported by the project (~3.10.12).
Using python3.10 (3.10.12)
24/03/05 21:52:56 WARN Utils: Your hostname, LAPTOP-GT5DD50V resolves to a loopback address: 127.0.1.1; using 172.30.165.220 instead (on interface eth0)
24/03/05 21:52:56 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
24/03/05 21:52:56 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address.
24/03/05 21:52:58 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
                       date|category|price|category_name|
 |2500000|2020-07-06|
                                       6 | 91.0 |
13 | 53.0 |
4 | 60.0 |
13 | 36.0 |
 2500001 2015-06-10
                                                            furniture
  2500003 2021-04-03
                                                                    food
                                                            furniture
 2500004 2018-02-16
                                        13 | 36.0 |

14 | 92.0 |

4 | 81.0 |

11 | 79.0 |

18 | 0.0 |

1 | 17.0 |

8 | 52.0 |

0 | 44.0 |
  2500005 2019-07-27
                                                            furniture
  2500006 2020-07-01
                                                                    food
                                                            furniture
 2500008 2018 - 02 - 16
  2500009 2019-10-02
 2500010 2017-12-27
2500011 2014-12-08
                                                            furniture
                                                                    food
                                        0 | 44.0 |

16 | 69.0 |

19 | 77.0 |

0 | 44.0 |

12 | 63.0 |

7 | 27.0 |

9 | 5.0 |

1 | 37.0 |
  2500012 2013-09-27
                                                            furniture
 2500013 2015-08-25
                                                            furniture
  2500014 2017-07-20
 2500015 2019-12-27
                                                            furniture
  2500016 2019-07-21
  2500017 2013-03-02
                                                            furniture
 2500018 2012-03-15
 2500019 2018-07-21
 only showing top 20 rows
```

- Utilisation de .collect() : Bien que .collect() force Spark à traiter toutes les données, on obtient une erreur qui indique un dépassement de la mémoire à cause du trop grand jeu de données. La mémoire en ram est insuffisante.

```
(base) jobordeau@LAPTOP-GTSDD56V:~/spark/spark-handson/src/fr/hymaia/exx4$ poetry run python_udf
The currently activated Python version 3.11.4 is not supported by the project (~3.10.12).
Trying to find and use a compatible version.
Using python3.10 (3.10.12)
24/03/05 20:53:56 WARN Utils: Your hostname, LAPTOP-GTSDD50V resolves to a loopback address: 127.0.1.1; using 172.30.165.220 instead (on interface eth0)
24/03/05 20:53:56 WARN Utils: Set SPARK_LOCAL IP if you need to bind to another address
setting default log level to "WARN".
To adjust logging level use sc.settoglevel(newLevel). For SparkR, use settoglevel(newLevel).
24/03/05 20:53:57 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
24/03/05 20:53:57 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
24/03/05 20:53:57 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
24/03/05 20:56:77 E0POP Evacutor. Evacution in task 15.0 in stage 3.0 (TID 90)0]

java.lang.outOfMemoryError: Java heap space
at java.nio.ByteBuffer.allocate(ByteBuffer.java:335)
at java.nio.ByteBuffer.allocate(ByteBuffer.java:335)
at org.apache.spark.serializer.SerializerHelper$.Sanonfun$serializeToChunkedBuffer$1$adapted(SerializerHelper.scala:40)
at org.apache.spark.serializer.SerializerHelper$.Sanonfun$serializeToChunkedBuffer$1$adapted(SerializerHelper.scala:40)
at org.apache.spark.serializer.SerializerHelper$.Sanonfun$serializeToChunkedBuffer$1$adapted(SerializerHelper.scala:40)
```

- Utilisation de count() après un groupBy() : Une solution qui fonctionne consiste à appliquer un `.groupBy()` suivi d'une opération d'agrégation comme count() sur le résultat de l'UDF. Cela force Spark à traiter toutes les données sans risque de dépassement de mémoire.
- Écriture dans un fichier: écrire le résultat dans un fichier par exemple, en utilisant write() est une excellente manière de mesurer le temps d'exécution, car cela inclut le temps de traitement complet de toutes les données.

Nous avons opté pour utiliser write car il implique une évaluation complète et réelle du DataFrame donc cela semble plus efficace pour mesurer le temps d'exécution. Mais le

désavantage est que nous aurons des résultats différents en fonction de la vitesse d'écriture de nos ordinateurs respectifs.

3. Isoler la Mesure du Temps d'UDF

Pour isoler le temps d'exécution de l'UDF de tout autre traitement, on mesure le temps avant et après l'application de l'UDF et l'écriture dans le fichier.

Implémentation:

```
start_time = time.time()

df_with_category_name = df.withColumn('category_name', category_name_udf(df['category']))

df_with_category_name.write.mode('overwrite').csv('/tmp/python_udf_output')

end_time = time.time()

print(f"Execution time with Python UDF: {end_time - start_time} seconds")
```

- Avant d'appliquer l'UDF : Enregistrez le temps de début.
- Appliquez l'UDF et forcez l'évaluation avec write.
- Immédiatement après l'évaluation: Enregistrez le temps de fin.
- -Calculez la différence entre le temps de fin et le temps de début pour obtenir le temps d'exécution.

4. Comparaison des Temps d'Exécution

Nous pouvons maintenant comparez les temps d'exécution obtenus pour l'UDF Python, l'UDF Scala et Sans UDF. Cette comparaison devrait révéler laquelle des deux approches est la plus performante dans votre cas d'utilisation spécifique.

```
Execution time without UDF: 66.68806886672974 seconds

Execution time with Scala UDF: 72.47169733047485 seconds

Execution time with Python UDF: 93.72714614868164 seconds

72.47 - 66.8 = 5,7 sec

93.72 - 68.8 = 24,7 sec
```

Utiliser UDF sera toujours plus long que sans mais on observe que le temps d'exécution de scala UDF est à peu près quatre fois plus rapide que le python UDF.

Exo4 partie 2

Pour la partie window function

Après le chargement des données, on a ajouté une colonne category_name en utilisant la fonction when de Spark SQL. Cela nous a permis de classifier chaque enregistrement dans des catégories plus larges ('food' ou 'furniture'), facilitant ainsi l'analyse par catégorie.

Ensuite nous avons calculé la somme des prix par catégorie d'article et par jour. Pour cela, on a défini une spécification de fenêtre avec Window.partitionBy('date', 'category_name'), qui nous a permis de regrouper les données par date et par nom de catégorie. On a ensuite utilisé la fonction sum('price').over(windowSpecDay) pour calculer la somme désirée au sein de chaque groupe.

Dans notre premier essai, on a observé que les résultats n'étaient pas conformes à nos attentes. On a identifié que le problème venait de la manière dont on avait partitionné les données. Le résultat affiché montrait un total global plutôt qu'un total quotidien par catégorie, ce qui suggère que soit la spécification de fenêtre n'est pas appliquée correctement, soit il y a une confusion dans l'affichage des résultats

Après révision, on a ajusté la spécification de la fenêtre pour s'assurer que le calcul de la somme se faisait correctement par date et par catégorie.

Test unitaire:

Nous avons aussi réalisé des tests unitaires et d'intégration pour tester toutes les fonctions dans les fichiers no_udf, scala_udf et python_udf

Exo 5:

1:

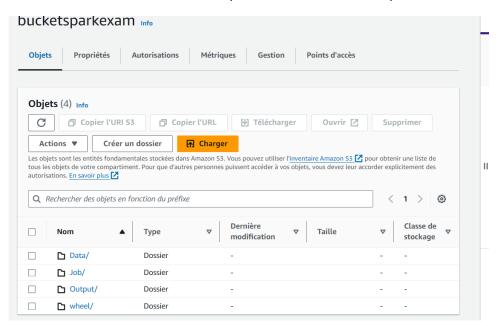
On a commencé par créer un utilisateur IAM , puis nous lui avons attribué les autorisations afin de pouvoir créer des Jobs glue, de pouvoir modifier des autorisations IAM , et de pouvoir ajouter/supprimer des fichiers sur le service S3.

Nous avons également permis à l'utilisateur d'accéder à AWS Cli , et nous avons récupérer les clés permettant de s'identifier sur AWS Cli.



2:

On a ensuite crée un bucket S3 qu'on a nommé bucketsparkexam vi l'interface Web.



3:

Nous avons créé un dossier Infra, et initialisé le répertoire et nous nous sommes aussi loggés en ligne de commande avec aws configure.

```
PS C:\Users\erwan\Desktop\Rendu_Spark\terraform_1.7.4_windows_amd64\Infra> terraform init

Initializing the backend...

Initializing provider plugins...
- Finding latest version of hashicorp/aws...
- Installing hashicorp/aws v5.39.1...
- Installed hashicorp/aws v5.39.1...

Terraform has created a lock file .terraform.lock.hcl to record the provider selections it made above. Include this file in your version control repository so that Terraform can guarantee to make the same selections by default when you run "terraform init" in the future.

Terraform has been successfully initialized!

You may now begin working with Terraform. Try running "terraform plan" to see any changes that are required for your infrastructure. All Terraform commands should now work.

If you ever set or change modules or backend configuration for Terraform, rerun this command to reinitialize your working directory. If you forget, other commands will detect it and remind you to do so if necessary.
```

4:

Nous avons utilisé la commande poetry build afin de récuperer le fichier .whl

		1 **	
spark_handson-0.1.0.tar.gz	06/03/2024 00:21	Dossier d'archive c	4 Ko
spark_handson-0.1.0-py3-none-any.whl	06/03/2024 00:21	Fichier WHL	7 Ko

5:

Nous avons upload les fichiers dont nous aurons besoin pour la configuration des fichiers tf

```
PS C:\Users\erwan\Desktop\Rendu_Spark\terraform_1.7.4_windows_amd64\Infra> aws s3 cp C:\Users\erwan\Desktop\Rendu_Spark\terraform_1.7.4_windows_amd64\spark_handson-0.1.0-py3-none-any.whl s 3://bucketsparkexam/wheel/spark_handson-0.1.0-py3-none-any.whl to s3://bucketsparkexam/wheel/spark_handson-0.1.0-py3-none-any.whl to s3://bucketsparkexam/wheel/spark_handson-0.1.0-py3-none-any.whl pS C:\Users\erwan\Desktop\Rendu_Spark\terraform_1.7.4_windows_amd64\city_zipcode.csv s3://bucketsparkexam/Data/city_zipcode.csv to s3://bucketsparkexam/Data/city_zipcode.csv upload: ..\city_zipcode.csv to s3://bucketsparkexam/Data/city_zipcode.csv to s3://bucketsparkexam/Data/city_zipcode.csv to s3://bucketsparkexam/Data/city_zipcode.csv to s3://bucketsparkexam/Data/city_zipcode.csv to s3://bucketsparkexam/Data/city_zipcode.csv to s3://bucketsparkexam/Data/city_zipcode.csv to s3://bucketsparkexam/Data/clients_bdd.csv to s3://bucketsparkexam/Data/clients_bdd.csv to s3://bucketsparkexam/Data/clients_bdd.csv to s3://bucketsparkexam/Data/clients_bdd.csv to s3://bucketsparkexam/Data/clients_bdd.csv to s3://bucketsparkexam/Job/exo2_glue_job.py s3://bucketsparkexam/Job/exo2_glue_job.py s3://bucketsparkexam/Job/exo2_glue_job.py s3://bucketsparkexam/Job/exo2_glue_job.py to s3://bucketsparkexam/Job/exo2_glue_job.py ps s3://bucketsparkexam/Job/exo2_glue_job.py ps c:\Users\erwan\Desktop\Rendu_Spark\terraform_1.7.4_windows_amd64\Infra> terraform apply
```

Etant donné que nous avions crée le bucket à l'aide de l'interface web d'Aws, on a ajusté le fichier bucket.tf :

Voici le fichier iam.tf

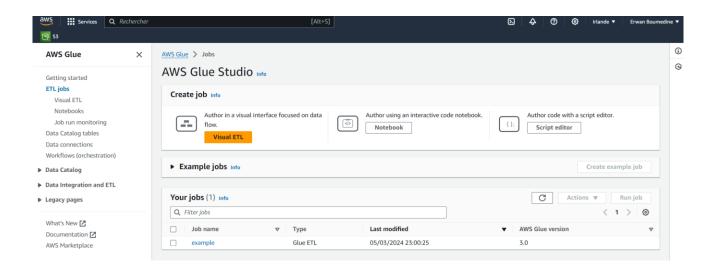
Ainsi que glue_job.tf

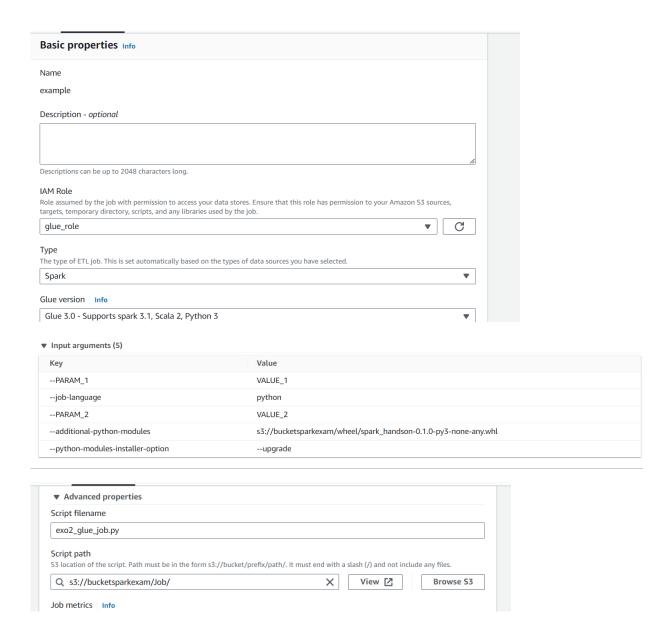
Voilà ce que ça donne :

```
### Second Common Second Commo
```

7:

Tout est prêt, le job a été créé:





8:

Nous avons ajusté le fichier exo2_glue_job.py avant de le réupload :

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, when
from awsglue.context import GlueContext
from awsglue.job import dob
from awsglue.utils import getResolvedOptions
from src.fr.hymaia.exo2.spark_aggregate_job import calculate_population_by_department
lfrom src.fr.hymaia.exo2.spark_clean_job import read_data, filter_adults, join_clients_cities, add_departmen

if __name__ == ' __main__':
    spark = SparkSession.builder.getOrCreate()
    glueContext = GlueContext(spark.sparkContext)
    job = Job(glueContext)
    args = getResolvedOptions(sys.argv, ["JOB_NAME"])
    job.init(args['JOB_NAME'], args)

    input_clients_path = "s3://bucketsparkexam/Data/city_zipcode.csv"
    input_cities_path = "s3://bucketsparkexam/Data/clients_bdd.csv"
    output_clean_path = "s3://bucketsparkexam/Output/clean/"
    output_aggregate_path = "s3://bucketsparkexam/Output/aggregate/"

    df_clients = read_data(spark, input_clients_path)
    df_cities = read_data(spark, input_clients_path)
    df_clients_adults = filten_adults(df_clients)
    df_joined = join_clients_cities(df_clients_adults, df_cities)
    df_with_department = add_department_column(df_joined)
    df_with_department.write.parquet(output_clean_path)
```

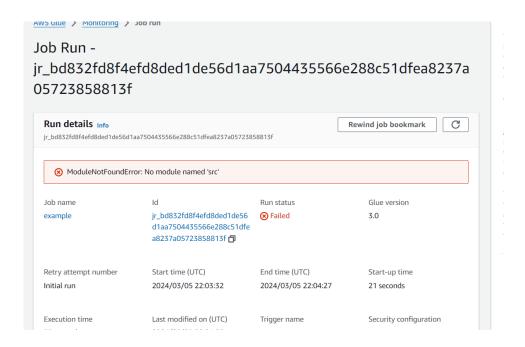
```
df_clients = read_data(spark, input_clients_path)
df_cities = read_data(spark, input_cities_path)
df_clients_adults = filter_adults(df_clients)
df_joined = join_clients_cities(df_clients_adults, df_cities)
df_with_department = add_department_column(df_joined)
df_with_department.write.parquet(output_clean_path)

df_clean = spark.read.parquet(output_clean_path)
df_population = calculate_population_by_department(df_clean)
df_population.coalesce(1).write.option("header", "true").csv(output_aggregate_path)

job.commit()
```

PS C:\Users\erwan\Desktop\Rendu_Spark\terraform_1.7.4_windows_amd64\Infra> aws s3 cp C:\Users\erwan\Desktop\Rendu_Spark\terraform_1.7.4_windows_amd64\exo2_glue_job.py s3://bucketsparkexam/Job/exo2_glue_job.py upload: ..\exo2_glue_job.py to s3://bucketsparkexam/Job/exo2_glue_job.py

Nous avons rencontré cette erreur :



Nous avons regénerer le fichier .whl , vérifier son contenu en l'extrayant et sa structure était tout à fait normale.

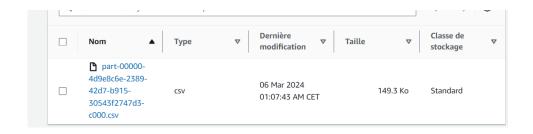
Aprés plusieurs tentatives vaines , on a voulu se convaincre que le job fonctionnait et qu'il s'agissait d'une erreur de dépendances , ducoup nous avons upload un nouveau script :

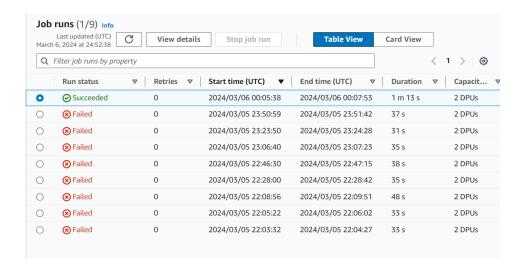
```
Script Job details
                                   Data quality - updated
                                                             Schedules
                                                                           Version Control
                       Runs
Script Info
1 import sys
     from awsglue.context import GlueContext
3 from awsglue.job import Job
     from awsglue.utils import getResolvedOptions
 from pyspark.sql import SparkSession
from pyspark.sql.functions import col
 8 ▼ def process_clients_data(input_path, output_path):
10
        spark = SparkSession.builder.getOrCreate()
11
         # Lecture du dataset
        df = spark.read.csv(input_path, header=True)
13
14
15
         df_filtered_sorted = df.filter(col("age") > 30).orderBy(col("age"))
16
        df_filtered_sorted.coalesce(1).write.option("header", "true").csv(output_path)
17
19
         spark.stop()
20
```

```
16
          df_filtered_sorted.coalesce(1).write.option("header", "true").csv(output_path)
17
18
19
           spark.stop()
20
21 v if __name__ == '__main__':
22
          spark = SparkSession.builder.getOrCreate()
glueContext = GlueContext(spark.sparkContext)
23
24
25
          job = Job(glueContext)
26
          args = getResolvedOptions(sys.argv, ["JOB_NAME"])
27
          job.init(args['JOB_NAME'], args)
28
          input_path = "s3://bucketsparkexam/Data/clients_bdd.csv"
output_path = "s3://bucketsparkexam/Output/clients_bdd_output"
29
30
31
          process_clients_data(input_path, output_path)
32
33
```

Et là tout fonctionne trés bien :

□ □ Data/ Dossier - - - □ □ Job/ Dossier - - -	Nom 🔺	Туре	▼ Dernière modificatio	n ▼ Taille	▽	Classe de stockage	▽
☐ ☐ Job/ Dossier	Data/	Dossier	-		-	-	
	☐ Job/	Dossier	-		-	-	
Output/ Dossier	☐ Output/	Dossier	-		-	-	
□ 🗅 wheel/ Dossier		Dossier	-		-	-	





Conclusion:

Ces exercices nous ont beaucoup aidé à comprendre Spark, la différence entre Python et Scala et la gestion de pipelines de données. On a pu voir comment tout se connecte, de Spark à AWS Glue, en passant par les dépendances. La pratique nous a rendu tout ça bien plus clair et accessible. C'était super enrichissant. Merci !