Rapport Big Data Frameworks : analyse d’une trace

Romaric Vandycke, Paul Dufour, Thomas Faguet, Maryam Diouf

Professeur référent : Juan Angel Lorenzo del Castillo

Année scolaire 2023/2024

Table des matières

1)Présentation de la trace Google

Pour ce projet, on devait travailler sur la trace de données de Google. La trace d'utilisation d'un cluster de Google est un ensemble de données décrivant de manière détaillée l'utilisation des ressources informatiques au sein d'un cluster de serveurs appartenant à Google.

Cette trace comprend des informations sur les tâches et les travaux exécutés sur les machines du cluster, ainsi que des données sur les ressources allouées, les exigences en ressources, et l'utilisation effective de ces ressources. Les données sont collectées à partir des divers composants du cluster, y compris les machines individuelles, les systèmes de gestion de cluster et les applications en cours d'exécution. Ces données comprennent des informations telles que les ressources allouées, les exigences en ressources, les tâches exécutées et les horodatages associés.

Dans notre cas, la trace date du mois de mai 2011. Notre objectif est dans un premier temps, d'identifier les tâches et les travaux les plus prenant en CPU, en valeur maximal et en moyenne. Ensuite, on va faire une identification des travaux et tâches qui prennent le plus de mémoire vive en valeurs maximales et en moyenne lors de la durée de la trace. Puis on va faire une classification des travaux dominants et enfin, pour les travaux dominants, on va essayer de trouver une corrélation entre la consommation de CPU et de la mémoire vive.

2) Le choix de Pyspark

Nous avons décidé d'utiliser Pyspark pour ce projet car Python est un langage de programmation facile à manipuler et performant. Pyspark et le mélange de python et du framework Apache Spark développée par l'université de Berkeley, il permet d'effectuer de l'analyse de gros volumes de données grâce à du calcul distribué.

L'utilisation de ce langage nous permettra de profiter du pyspark sql afin de pouvoir effectuer des requêtes sql entre les différentes tables à notre disposition ainsi que d'utiliser des fonctions d'agrégations.

Partie 1 du projet:

La première partie du projet consiste à identifier les tâches et travaux les plus prenant en CPU, en valeur maximal et en moyenne. On a une grande quantité de données qui prend du temps à compiler. Pour éviter de perdre du temps à exécuter tous les fichiers avec notre code, on a testé notre code uniquement sur un seul document pour voir si cela fonctionnait. Une fois que ça fonctionnait, on a compilé le code sur tous les documents du dossier.
Pour commencer, on a créé le Data Frame comme expliqué dans la partie précédente. Pour cela nous avons dû définir les en-têtes avec les informations récoltées dans le fichier excel schema.csv, et le type de données défini dans le même fichier. Pour faire cela nous avons dû importer différents types de pyspark.sql.types.

```
from pyspark.sql.types import StructType, StructField, LongType, IntegerType, StringType, FloatType, BooleanType
import os

# Créer une liste avec chaque en-tête de colonne
header = [
    "time", "missing info", "job ID", "task index", "machine ID", "event type",
    "user", "scheduling class", "priority", "CPU request",
    "memory request", "disk space request", "different machines restriction"
]

# Définir le schéma pour les éléments de la table
dfTE_Schema = StructType([
    StructField(header[0], LongType(), True),
    StructField(header[1], LongType(), True),
    StructField(header[2], LongType(), True),
    StructField(header[3], IntegerType(), True),
    StructField(header[4], IntegerType(), True),
    StructField(header[5], IntegerType(), True),
    StructField(header[6], StringType(), True),
    StructField(header[7], IntegerType(), True),
    StructField(header[8], IntegerType(), True),
    StructField(header[9], FloatType(), True),
    StructField(header[10], FloatType(), True),
    StructField(header[11], FloatType(), True),
    StructField(header[12], BooleanType(), True),
])
```

Ensuite afin d'être capable d'ouvrir les fichiers qui n'était pas décompressée nous avons utilisé ("compression","gzip")

Ensuite on nettoie le fichier en gardant les lignes où les valeurs des colonnes "mean CPU usage rate", "machine CPU rate" et machine ID" sont différentes de 0 et qui sont non nulle.

```
[1]  ##Nettoyage des données
     df_files_task_usage_1_nettoyer = df_files_task_usage_1.filter(#col("machine ID").isNotNull() &
                                        (col("mean CPU usage rate").isNotNull() & (col("mean CPU usage rate") != 0)) &
                                        (col("maximum CPU rate").isNotNull() & (col("maximum CPU rate") != 0)))
```

Par la suite, on affiche un tableau en affichant les résultats de "max (maximum CPU rate)" qui calcule le maximum de CPU par ligne et "avg_cpu_consumption_per_task" pour le calcule de la consommation moyenne de cpu par tâches.

```
    # Lire le fichier CSV compressé avec le schéma spécifié
df_temp = spark.read.format("csv") \
    .option("mode", "PERMISSIVE") \
    .option("sep", ",") \
    .option("inferSchema", "false") \
    .option("header", "false") \
    .option("nullValue", "") \
    .option("compression", "gzip") \
    .schema(dfTU_Schema) \
    .load(file_path)
```

Identifications des Jobs dominants

Dans un premier temps nous avons cherché à comprendre les données que nous possédions.

Dans le dossier task usage, nous disposons des variables “CPU Rate”, “Maximum CPU Rate” et “Sampled CPU Rate”. D'après la documentation fournie ces données sont collectées sur des périodes de 5 mn, si la task est terminée durant cette période ou bien mise en attente, l'échantillonnage peut durer encore quelques instants avant de prendre fin. La variables Sampled CPU Rate représente une valeur prise aléatoirement durant cette période d’échantillonnage et ne possède de valeur statistiques que sur des couples (job,task) possédant plusieur itérations au sein de notre table.

Ici nous souhaitons obtenir les valeurs moyennes et maximales de la consommation de CPU par task. Ici, la consommation est mesuré par unité de CPU-core seconde par seconde ce qui signifie que si par exemple une tâche prenait en permanence la puissance de 2 coeur, la valeur mesurait serait de 2.

Il aurait été intéressant de comparer la valeur mesurée dans cette table avec les performances de la machine sur la task à été lancée. Toutefois dans la table machines_attributes, les nom des caractéristiques des machines ont été transformés et ne sont pas trouvables pour nous. Il a fallu faire sans.

Afin d’obtenir les résultats demandés nous allons utiliser des fonctions d'agrégation sur nos dataframes. Nous utiliserons average (avg) sur la colonne de consommation moyenne et max (max) sur celle du pic de consommations.

Finalement nous avons décidé de représenter les résultats par moyenne décroissante puis valeur maximum décroissantes.

Vous trouverez dans le fichier datacpu.csv les 10000 jobs dominant plus prenant en valeur moyenne

```
agg_expr = {
    "CPU rate": "avg",
    "maximum CPU rate": "max"
}

avg_cpu_consumption_per_task = df_files_task_usage_nettoyer.groupBy("job ID", "task index").agg(
    agg_expr
).withColumnRenamed("avg(CPU rate)", "avg_CPU_rate")
```

On trie les résultats par ordre décroissant par la variable “avg_cpu_cosumption_per_task” et si deux ligne ont le même résultat, alors on trie ces lignes par ordre décroissant par la variable “max (maximum CPU rate)”

Il est bon de remarquer que nous avons décidé de supprimer les lignes contenant des valeur null ou vide pour les variables qui nous intéressent.

Nous avons également décidé de supprimer les ligne pour lesquelles la valeur machine Id est null, afin d'être dans le futur capable de retrouver les attribut de la machine pour analyse plus complètes

Identifications des Jobs à haute consommation de mémoire vive

On refait les mêmes opérations que l'on a fait précédemment mais cette fois-ci pour la consommation de mémoire.
On fait d'abord le nettoyage , puis on définit les variables que l'on veut calculer et ressortir et trier le résultat par ordre décroissant en fonction de "avg_total_page_cache_memory" et ensuite par "max (maximum memory usage)" s'il y a des valeurs similaires pour la première variable cité.

Nous utilisons les même fonctions que précédemment mais sur les variables concernées, les 10000 premier couple (job,task) sont disponible dans le fichier csv datafinalcache.csv fournie en annexe

```
[ ] df_with_cache_consumption = df_files_task_usage_1_nettoyer.withColumn("cpu_consumption_per_task", col("mean CPU usage rate"))

    agg_expr = {
        "total page cache": "avg",
        "maximum memory usage": "max"
    }

    avg_cache_consumption_per_task = df_files_task_usage_nettoyer.groupBy("job ID", "task index").agg(
        agg_expr
    ).withColumnRenamed("avg(total page cache)", "avg_total_page_cache_memory")

    cache_conso_final = avg_cpu_consumption_per_task.orderBy(desc("avg_total_page_cache_memory"),desc("max(maximum memory usage)")).show()
```

```
+----------+----------+------------------------+--------------------------+
|    job ID|task index|max(maximum memory usage)|avg_total_page_cache_memory|
+----------+----------+------------------------+--------------------------+
|4476805516|       383|                  0.2261|       0.22796470540411332|
|4476805516|       683|                  0.2251|        0.2275882345788619|
|4476805516|       448|                  0.2256|        0.2273529417374555|
|4476805516|        15|                  0.2261|       0.22715882431058323|
|4476805516|       271|                  0.2253|       0.22673529474174275|
|4476805516|       889|                  0.2253|       0.22607647145495696|
|4476805516|       346|                  0.2258|       0.22599999957224903|
|4476805516|       322|                  0.2275|        0.2259588188984815|
|4476805516|       552|                  0.2266|        0.2259294083889793|
|4476805516|       134|                  0.2273|       0.22581176722750945|
|4476805516|        95|                  0.2256|       0.22532352980445414|
|4476805516|       481|                  0.2295|       0.22515294131110697|
|4476805516|       911|                  0.2378|         0.225117646596011|
|4476805516|         9|                  0.2256|       0.22502353173844955|
|4476805516|       106|                  0.2256|       0.22499411947586956|
|4476805516|        30|                  0.2261|        0.2249529423082576|
|4476805516|       975|                  0.2251|       0.22483529325793772|
|4476805516|       922|                  0.2495|        0.2248294099288828|
|4476805516|       358|                  0.2253|       0.22455882324891932|
|4476805516|       530|                  0.2266|       0.22448235311928919|
+----------+----------+------------------------+--------------------------+
only showing top 20 rows
```

Classifications des jobs dominants par classe de priorité.

Grâce aux les jobs dominant que nous avons réussi à obtenir nous allons faire une jointure avec la table task event dans laquelle se trouve la valeur priority,
Toutes les tâches ont une valeur de priorité représenté par un nombre entier, le nombre 0 représentant les tâches avec la plus faible priorité.
Plus une tâches à une priorité élevée, plus elle sera prioritaire pour accéder aux ressources.

```
Job_Dominant_Priority = cpu_conso_final.join(df_files_task_events.select("job ID", "task index", "priority"), ["job ID", "task index"], "left_outer")

Job_Dominant_Priority.orderBy(desc("priority")).show(10)

Job_Dominant_Priority.printSchema()
```

On remarquera ici l'utilisation d'une jointure externe à gauche, cela va nous permettre de garder les lignes on nous n'avons pas de correspondance sur les couples (job ID, task)

Etude de la corrélation entre la consommation de CPU et de la mémoire vive pour les jobs dominants

La dernière partie du projet consiste à étudier la corrélation entre la consommation de CPU et de la mémoire vive pour les jobs dominants. On a une grande quantité de données qui prend du temps à compiler. Pour éviter de perdre du temps à exécuter tous les fichiers avec notre code, on a testé notre code uniquement sur un seul document pour voir si cela fonctionnait. Une fois que ça fonctionnait, on a compilé le code sur tous les documents du dossier.

Nous avons réussi à obtenir dans la première partie les jobs dominant , maintenant nous allons réaliser une jointure avec le résultat obtenu en deuxième partie, nous permettant ainsi d'avoir les valeurs moyenne de consommation de CPU et de consommation de mémoire vive.

La jointure sera une simple jointure interne, car seul les jobs dominant nous intéresse, ici les 10000 premiers.

```
joined_df = cache_conso_final.join(cpu_conso_final.select("job ID", "task index", "avg_CPU_rate"),
                                   ["job ID", "task index"], "inner")

# On prend uniquement les collone qui nous intéressent pour ce résultats
result_df = joined_df.select("job ID", "task index", "avg_CPU_rate", "avg_total_page_cache_memory")
```

On ne peut pas utiliser des bibliothèques graphiques comme matplotlib pour afficher des nuages de points pour étudier la corrélation entre ces deux variables, car elle nécessite de collecter les données et les mettre en ligne avec la fonction collect(), ou de convertir le dataframe spark en dataframe pandas avec toPandas(). Pour des grandes quantités de données, ces fonctions ne sont pas envisageables avec de faibles ressources car elles prennent trop de temps. On utilise à la place la fonction corr() de pyspark.sql qui étudie directement la corrélation entre les deux variables dans le dataframe spark.

Nous allons donc tout simplement utiliser la fonction cor de pyspark sql afin d'obtenir le résultat.

```
col1 = "avg_CPU_rate"
col2 ="avg_total_page_cache_memory"

# Calculer la corrélation entre les deux colonnes choisies
correlation = df_files_task_usage_nettoyer.select(corr(col1, col2)).collect()[0][0]
print(f"La corrélation entre {col1} et {col2} parmie les 100000 premier jobs dominant est : {correlation}")
```

Nous avons également décidé d'étudier la corrélation entre les sampled CPU rate et les "memory access per instructions" cette valeur a pour but de compter le nombre de caches miss répertoriés lors de l'exécution de la task.

Nous obtenons une valeur de -0.209 ce qui nous semble cohérent car un nombre de cœur plus grand augmente la taille de la mémoire cache de manière mécanique. Malgré que nous n'ayons pas accès au nombre de cœur avec nos données, un grand échantillon peut laisser envisager une quantité plus importante de cœur.

Toutefois, nous devons faire preuve de prudence car nous ne connaissons pas en détail les tâches effectuées et un nombre de cache miss important peut également être dû à une mauvaise implémentation.

Conclusion

Pour conclure, ce projet nous a permis de mettre en application le cours que nous avons vu sur un très gros ensemble de données. Nous avons ainsi pu être témoin du caractère chronophage de l'étude de ces données. Nous étions limités dans l'analyse de nos résultats par une analyse partielle. En effet, afin d'avoir une représentation visuelle de nos résultats, la compilation pouvait prendre plusieurs heures.

Ce projet nous a donc permis d'instaurer une nouvelle dynamique de travail afin d'éviter de longues heures de compilation. Ce projet à également nécessité une étude profonde de la documentation afin d'être capable d'obtenir des informations de qualité. Toutefois, les caractéristiques des machines étant encodées par google au sein de la trace, nous nous retrouvions avec finalement assez peu d'informations.