Rapport
Données Massives

Guillaume Zimol - Hugo Mathieu Steinbach

La mine d'Alibaba

2023 - 2024

Table des matières

Exploration des données

Dans un premier temps nous avons examiné les données dans le fichier "selectioncourt.csv" voici la première ligne :

ins_1076700187,M1,j_3691533,Terminated,129332,129359,m_2401,95,222,0.13,0.16

Nous avons ainsi pu à l'aide du sujet déterminé à quoi correspondent chacune des informations présentes.

- **Nom de l'instance (ins_1076700187)** : Il s'agit d'un identifiant unique pour chaque instance de tâche. Dans cet exemple, l'instance est nommée ins_1076700187.
- **Nom de la tâche (M1)** : C'est le nom de la tâche à laquelle appartient l'instance. Dans ce cas, la tâche est appelée M1. Elle peutêtre sous différente forme, par exemple M3-2-1, ce qui indique que c'est la tâche 3 et qu'elle dépend de la tâche 2 et de la tâche 1. Parfois elle est également sous la forme task_XXXX, ce qui correspond à une tâche unique et indépendante.
- **Nom du job (j_3691533)** : C'est l'identifiant du job auquel appartient l'instance. Ici, le job est identifié par j_3691533.
- **Statut de l'instance (Terminated)** : Cela indique l'état de l'instance. Ici, Terminated signifie que l'instance s'est terminée correctement.
- **Date de départ (129332)** : Représente le moment de début de l'instance, donné en secondes depuis un point de départ arbitraire.
- **Date de fin (129359)** : Représente le moment de fin de l'instance, également en secondes depuis le même point de départ.
- **Identifiant de la machine exécutant l'instance (m_2401):** C'est l'identifiant de la machine sur laquelle l'instance a été exécutée. Ici, c'est la machine m_2401.
- **Utilisation moyenne des CPUs (95)** : Un chiffre indiquant l'utilisation moyenne des CPU par l'instance, où 100 représenterait l'utilisation complète d'un cœur de CPU. Ici, la valeur est de 95, ce qui signifie qu'un cœur a été utilisé à 95%.
- **Utilisation maximale des CPUs (222)** : Un chiffre indiquant l'utilisation maximale des CPU par l'instance. Ici, elle est de 222.
- **Utilisation moyenne de la mémoire (0.13)** : Indique l'utilisation moyenne de la mémoire par l'instance, normalisée entre 0 et 100. Ici, elle est de 0.13.
- **Utilisation maximale de la mémoire (0.16)** : Indique l'utilisation maximale de la mémoire par l'instance, également normalisée entre 0 et 100. Ici, elle est de 0.16.

Organisation des fichiers, structure du code et utilisation de l'application

Le projet est structuré en différents fichiers répartis dans différents fichiers.
En voici l'arborescence :

```
└──Projet
   │  build_jar.sh
   │  etape1.sh
   │  etape2.sh
   │  etape3.sh
   │  etape4.sh
   │  etape5.sh
   │  hadoop-common-3.3.6.jar
   │  hadoop-mapreduce-client-core-3.3.6.jar
   │  MyHadoopApps.jar
   │  runAll.sh
   │  Sujet.pdf
   ├──corpus
   │      selectionComplete.csv
   │      selectionCourt.csv
   ├──output
   │  ├──etape1_2024-03-02_11-47
   │  │      jobs_complete.csv
   │  │      jobs_court.csv
   │  │      tasks_complete.csv
   │  │      tasks_court.csv
   │  ├──etape2_2024-03-02_11-50
   │  │      etape2_complete.csv
   │  │      etape2_court.csv
   │  ├──etape3_2024-03-02_12-18
   │  │      etape3_complete.csv
   │  │      etape3_court.csv
   │  ├──etape4_2024-03-04_14-37
   │  │      etape4_court.csv
   │  ├──etape5_2024-03-02_12-50
   │  │      jobs_complete.csv
   │  │      jobs_complete_0-10.csv
   │  │      jobs_complete_10-15.csv
   │  │      jobs_complete_15-1000.csv
   │  │      jobs_court.csv
   │  │      jobs_court_0-10.csv
   │  │      jobs_court_10-15.csv
   │  │      jobs_court_15-1000.csv
   │  │      tasks_complete.csv
   │  │      tasks_complete_0-500.csv
   │  │      tasks_complete_500-99999999.csv
   │  │      tasks_court.csv
   │  │      tasks_court_0-500.csv
   │  │      tasks_court_500-99999999.csv
   └──src
      ├──drivers
      │      CoreEstimationDriver.java
      │      CpuUsageAnalysisDriver.java
      │      JobAnalysisDriver.java
      │      MachineMemoryWritable.java
      │      MemoryRatioDriver.java
      │      TaskAnalysisDriver.java
      ├──mappers
      │      CoreEstimationMapper.java
      │      CoreUsageMapper.java
      │      JobInstanceMapper.java
      │      MemoryUsageMapper.java
      │      TaskInstanceMapper.java
      └──reducers
             CoreEstimationReducer.java
             CoreUsageReducer.java
             JobReducer.java
             MemoryUsageReducer.java
             TaskReducer.java
```

Dans le dossier src, les trois dossiers drivers, mappers et reducers, Les fichiers :

- Task et Job correspondent aux fichiers de l'étape 1 et 5.
- CpuUsage correspondent à ceux de l'étape 2.
- Core Estimation correspondent à ceux de l'étape 3.
- Memory à ceux de l'étape 4.

À la racine les différents package .jar (utilisés essentiellement pour la vérification syntaxique du code) compatibles avec la version installée du cours.

Le sujet ainsi que notre Archive .jar et des script shell, permettant l'exécution automatisé et sur n'importe quelle machine des différentes étapes. La totalité des commandes que nous avons utilisées y sont présentées et commentées en plus d'être facilement récupérables et donc réutilisables.

Il est également possible d'exécuter le script runAll.sh pour lancer toutes les étapes d'un seul coup. Notons que certaines commandes qui prennent du temps à s'exécuter (sur le corpus complet) ont été mises en commentaire.

Le dernier dossier, output contient l'intégralité des résultats des exécutions, que nous avons effectués dans la dernière version du programme (l'étape 4 n'est pas incluse notre fichier en sortie dépassant les 200Mo). Et correspondent strictement à la sortie de notre programme sur notre machine. Ils peuvent donc être interprétés comme nos résultats, bien qu'il soit tout à fait possible de compiler et d'exécuter notre programme à partir des fichiers présents dans src (grâce au script build_jar.sh fournit en TP).

Concernant les commandes utilisés, outre les get, put et autre commandes de bases, nous avons utilisé getmerge qui permet de récupérer les fichiers de plusieurs reducers et de les fusionner en un seul lors de la réception, cette fusion crée un fichier .crc qui sert "de signature" pour en vérifier l'intégrité, nous supprimons ce fichier lors de chaque étape dans nos différents scripts.

Notons également, que dans chacun des scripts nous avons une première étape de chargement des différents corpus dans HDFS. Et que nous utilisons la date actuelle (avec les minutes) pour pouvoir générer des résultats dans des dossiers de sortie différents.

Concernant Hadoop, nous avons retiré la limite de bloc mise en place lors des TPs. Nous forçons également la présence de deux reducers dans chaque étape (excepté la 2ème, voire explication ci-dessous). Mais cela semble sans impact sur l'étape 5 puisque le fichier en sortie est unique malgré la spécification avec la commande donnée dans le sujet.

Choix effectués lors des différentes étapes

Étape 1

Pour la première liste, nous avons remarqué que certaines tâches ne sont pas terminées, elles ont donc un “Date de fin” qui est par défaut à 0, notre mapper ne récupère donc pas ces lignes, pour éviter un calcul de durée d’une tâche et un calcul de la durée moyenne absurde.

Pour la seconde liste, nous avons décidé de calculer les tâches en suivant cette logique, pour les tâches dans le format “X3_2_1”, nous récupérons la plus grande valeur, la première, ici 3. Nous conservons cette valeur et la mettons à jour en fonctions des différentes tâches au sein d’un même job, en plus de cela nous comptons individuellement chacune des tâches sous le format “task_XXX”, enfin nous additionnons la plus grandes tâches (par exemple, ici 3) avec le nombre de “task_XXX” trouvées, ce qui nous donne le nombre de tâches d’un seul job.

Étape 2

Pour l’étape 2, nous avons décidé de définir un seuil pour la détection de pics.
On définit un pic comme un instant où :

- Le nombre de cœurs utilisés est strictement plus grand que la seconde d’avant.
- Le nombre de cœurs utilisés diminue strictement à la fin du pic.

Notre interprétation du strictement est “100 unités de cores supérieures”, sachant que 100 unités de cores sont équivalentes à un cœur sur la machine.

Ainsi 0, 1, 2, 1 n’est pas détecté comme un pic.
Tandis que 0, 100, 101, 101, 150, 200, 50, détecte un pic de 0 à 200, d’une durée de 6 secondes.

Pour le tri, nous avons utilisé la fonction compare des Integer, dans la fonction cleanup de Hadoop, ce qui nous permet à partir de la liste des “machines - cores” produite lors de l’étape reduce, de produire cette liste de pics.

Pour cette étape l’utilisation de plusieurs reducers pose problème, en effet si nous avons deux reducers, cela produit deux fichiers en sortie, ce qui est un premier problème solvable grâce à -getmerge de Hadoop.
Le réel obstacle à l’utilisation de plusieurs reducers est le fait que la répartition se fasse une fois dans le premier puis une fois dans le second, ainsi un reducer aura l’ensemble des actions mais chaque 2 secondes.
Pour résoudre cela nous avons utilisé un partitionner qui répartit la moitié des actions dans le premier reducer et l’autre dans le second, ainsi chaque reducer obtient en entrée des données sur chaque seconde.
Cependant l’usage de plusieurs reducers a continué de poser problème, car la détection de pic entre la fin du premier reducer et le début du second devenait impossible. Et c’est donc pour cette raison, qu’il est au final impossible d’utiliser plusieurs reducers lors de cette étape.

Étape 3

L'étape 3 est une implémentation classique d'un MapReduce en Hadoop, l'usage de plusieurs reducers ne pose pas de soucis, du moment que nous utilisons -getmerge pour récupérer les fichiers produits sur HDFS. Ainsi nous obtenons en sortie la liste des machines avec leur nombre de cœurs estimés.

Étape 4

Pour l'étape 4, nous avons utilisé un mapper, qui donne en sortie, un identifiant de tâche lié à un Writable customisé, "MachineMemoryWritable" qui contient l'identifiant de la machine et l'utilisation en mémoire sur cette tâche pour cette même machine.

Notre reducer, qui vient donc à la suite de ce Mapper, récupère la liste des machines pour chaque tâches, effectue un calcul de ratio entre chacunes des machines, si ce ratio est de 1 alors la même mémoire est utilisée pour effectuer cette tâche, nous ignorons donc ce cas dans notre résultat final, cependant s'il est différent alors nous affichons en sortie le couple de machine avec le ratio correspondant.
Cela permet de se faire une idée des performances des machines en les classant suivant leur ratio, cependant cela ne se fait qu'au sein d'une tâche, et les résultats étant différents d'une tâche à une autre, parfois la machine A utilise davantage de mémoire pour effectuer la tâche A que la machine B, mais dans la tâche B, c'est l'inverse la machine B utilisant davantage de mémoire que la machine A. Nous ne savons donc pas comment fusionner ces données. Le résultat présente donc actuellement les différents ratios entre les machines, et permet la création de graphe au sein d'une même tâche uniquement. Les données sont sous la forme :

m_3069;m_2861;0.9767441
m_3069;m_1433;0.7241379
m_3069;m_334;0.76363635
m_3069;m_2828;0.9767441
m_2861;m_1764;1.0238096
m_2861;m_396;1.0238096
m_2861;m_1433;0.7413793
m_2861;m_334;0.78181815
m_2861;m_1725;1.0238096
m_2861;m_2042;1.0238096
m_1764;m_1433;0.7241379
m_1764;m_334;0.76363635
m_1764;m_2828;0.9767441
m_396;m_1433;0.7241379
m_396;m_334;0.76363635
m_396;m_2828;0.9767441
m_1433;m_334;1.0545454
m_1433;m_1725;1.3809524
m_1433;m_2042;1.3809524
m_1433;m_2828;1.3488371

Et elles signifient, pour la première ligne, que la machine "m_3069" utilise 97.6% de la mémoire utilisée par la machine "m_2861".

Étape 5

Pour l'étape 5, nous avons repris nos deux mappers et nos deux reducers ainsi que les deux drivers de l'étape 1 que nous avons modifié, ainsi lancer l'archive sans paramètre comme dans l'étape 1 permet toujours d'exécuter le code sans les paramètres de l'étape 5, cependant nous avons différentes possibilités concernant les paramètres que je vais énoncé ci-dessous :

- Ne considérant pas certaines instances selon leur statut :

S'effectue en ajoutant -ignore <Statuts>, ainsi si l'on exécute le programme de cette façon : hadoop jar MyHadoopApps.jar [...] -ignore Running, alors toutes les tâches dans l'état "Running" seront ignorés lors du Mapper, pour faire cela nous récupérons le paramètre dans le driver, et nous l'envoyons à l'objet de Configuration, et récupérons donc ce paramètre dans le mapper ce qui nous permet d'ignorer les lignes concernés dans le csv.
On peut séparer les différents Statuts à ignorer avec des virgules exemple : "Running,Terminated"

- Classant les tâches et les jobs selon la colonne de son choix :

S'effectue en ajoutant -sort <Colonne>, et permet de trier les données en sortie selon la colonne donnée en paramètre.
Nous utilisons la aussi un Comparator dans une fonction cleanup, pour le post-traitement, pour trier les données, ce qui nous évite encore une fois de passer par un reducer supplémentaire.
Si le numéro de colonne est inexact, aucun tri n'est effectué.

- Séparant les tâches en plusieurs fichiers finaux :

S'effectue en ajoutant -separateFiles <Colonne> <NombreDeFichiers> <Sections>, et permet de générer plusieurs fichiers en sortie, par exemple pour l'utilisation de "-separateFiles 1 2 0-500,500-99999999" cela va permettre de créer deux fichiers en sortie, l'un avec les données de 0 à 500 sur la colonne 1 et l'autre avec les données de 500 à 9999999 sur cette même colonne.

Pour cela nous avons utilisé la classe MultipleOutput présente dans Hadoop.

Nous pouvons bien entendu cumuler les différents paramètres. Comme pour les autres tâches des exemples d'utilisations et de résultats ont été fournis avec l'archive déposée.