Rapport du projet

Université de Bordeaux, M2 Génie Logiciel
Programmation Large Échelle
2019-2020

Guillaume NEDELEC et Dorian ROPERT

Infos :

- **Les résultats sont les résultats sur les 1 premiers Go du fichier phases.csv (**`/user/gnedelec001/resources/phasesHead1Go.csv`**). Il nous a été impossible de lancer nos programmes sur le fichier de 69Go dus à la sollicitation du cluster par l'ensemble des groupes et les problèmes de cluster récurrents (machine éteinte ect…)**

- **Pour générer les histogrammes, on utilise des intervalles prédéfinies (différents entre la durée, le nombre de patterns, ect…) dans le code. En utilisant les tranches, les résultats de l'histogramme ne sont pas très exploitables.**

1.a)

Pour obtenir la distribution des phases qui ne sont pas idle, on commence tout d'abord à lire le fichier ligne et à filtrer pour garder seulement les lignes non idle (on regarde si la colonnes "pattern" contient une valeur différentes de -1 et de "pattern" (l'en-tête).
Une fois ces données récupérer, on récupère seulement les durées (colonne "duration") de chacune des lignes que l'on stocke dans un `JavaDoubleRDD`.
Enfin il suffit seulement de récupérer les statistiques (méthode `.stats()`) pour obtenir le minimum, le maximum, la moyenne ainsi que l'histogramme des durées.
Pour calculer la médiane et les quartiles, on transforme et on trie le RDD de Double dans un `JavaPairRDD<Long, Double>` pour ensuite calculer les index de la médiane, du premier et du 3ème quartile.
Une fois les index récupérés, il suffit de récupérer les valeurs en appliquant la fonction `.lookup(index).get(0)` sur le `JavaPairRDD` triés pour obtenir la valeur de l'index voulu.
Le tout est ensuite écrit dans un fichier de sortie.

RÉSULTAT 1a:

```
Nombre de plages horaires correspondantes: 10370012 sur 10410625
Minimum:        1.0
Maximum:        1.8003404697E10
Moyenne:        75160.47431420643
Mediane:        371.0
1er Quartile:   102.0
3ème Quartile:  1644.0
Histogramme:
Entre 0E0 et 1E0 :            0
Entre 1E0 et 1E2 :            2533693
Entre 1E2 et 1E3 :            4505995
Entre 1E3 et 5E3 :            2063456
Entre 5E3 et 1E4 :            493604
Entre 1E4 et 2,5E4 :          391908
Entre 2,5E4 et 5E4 :          229252
Entre 5E4 et 7,5E4 :          61183
Entre 7,5E4 et 1E5 :          24109
Entre 1E5 et 5E5 :            39859
Entre 5E5 et 1E6 :            5207
Entre 1E6 et 1E7 :            12897
Entre 1E7 et 1E8 :            7512
Entre 1E8 et 2E9 :            1325
Entre 2E9 et 2E10 :           12
```

1.b)

On effectue le même cheminement que dans la question 1.a) en appliquant cette fois un filtre gardant seulement les lignes qui ont pour valeur -1 dans la colonne “patterns”.

RÉSULTAT 1b :

```
Nombre de plages horaires correspondantes: 40613 sur 10410625
Minimum:        1.0
Maximum:        7.6206603192E10
Moyenne:        1.605288615627993E7
Mediane:        51386.0
1er Quartile:   36297.0
3ème Quartile:  77726.0
Histogramme:
Entre 0E0 et 1E0 :            0
```

```
Entre 1E0 et 1E2 :            192
Entre 1E2 et 1E3 :            2212
Entre 1E3 et 5E3 :            2414
Entre 5E3 et 1E4 :            570
Entre 1E4 et 2,5E4 :          1237
Entre 2,5E4 et 5E4 :          13006
Entre 5E4 et 7,5E4 :          9560
Entre 7,5E4 et 1E5 :          5603
Entre 1E5 et 5E5 :            3815
Entre 5E5 et 1E6 :            146
Entre 1E6 et 1E7 :            618
Entre 1E7 et 1E8 :            650
Entre 1E8 et 2E9 :            536
Entre 2E9 et 2E10 :           52
```

1.c)

Ici, on filtre les données pour garder seulement les lignes qui ne contiennent qu'un seul pattern (et qui n'est pas idle). On effectue ensuite un mapping pour associer chaque pattern à ses durées. Enfin, on parcourt cet ensemble clé par clé et on effectue la distribution sur la liste de valeurs associées (récupérée grâce à `lookup(index`)), et on effectue les même calculs que dans la question 1.a) pour obtenir les distributions. Cependant, nous n'avons pas réussi à itérer sur les clés (charger les clés du map dans une liste ne fonctionnait pas). Nous n'avons donc pas de résultat à présenter pour cette question.

RÉSULTAT 1c:

2)

Cette fois, on filtre les données pour garder seulement les lignes ou les phases sont non idle. Sur ces données, on stocke dans un `JavaDoubleRDD`, le nombre de patterns, plutôt que la durée.

On applique la distribution sur ce `JavaDoubleRDD` comme expliqué dans la question 1.a

RÉSULTAT 2 :

```
Minimum:       1.0
Maximum:       9.0
Moyenne:       4.786853573554274
Mediane:       5.0
1er Quartile:   4.0
```

```
3ème Quartile:  5.0
Histogramme:
Entre 0E0 et 1E0 :            0
Entre 1E0 et 2E0 :            76642
Entre 2E0 et 3E0 :            316379
Entre 3E0 et 4E0 :            673340
Entre 4E0 et 5E0 :            1927344
Entre 5E0 et 6E0 :            5531511
Entre 6E0 et 7E0 :            1441799
Entre 7E0 et 8E0 :            338582
Entre 8E0 et 9E0 :            57225
Entre 9E0 et 1E1 :            7190
Entre 1E1 et 1,1E1 :          0
Entre 1,1E1 et 1,2E1 :        0
Entre 1,2E1 et 1,3E1 :        0
Entre 1,3E1 et 1,4E1 :        0
Entre 1,4E1 et 1,5E1 :        0
Entre 1,5E1 et 1,6E1 :        0
Entre 1,6E1 et 1,7E1 :        0
Entre 1,7E1 et 1,8E1 :        0
Entre 1,8E1 et 1,9E1 :        0
Entre 1,9E1 et 2E1 :          0
Entre 2E1 et 2,1E1 :          0
Entre 2,1E1 et 2,2E1 :        0
```

3)

Cette fois, on filtre les données pour garder seulement les lignes ou les phases sont non idle. Sur ces données, on stocke dans un `JavaDoubleRDD`, le nombre de jobs. On applique la distribution sur ce `JavaDoubleRDD` comme expliqué dans la question 1.a

RÉSULTAT 3 :

```
Minimum:  1.0
Maximum:  71.0
Moyenne:  9.780269492455943
Mediane:  5.0
1er Quartile:  3.0
3ème Quartile: 11.0
Histogramme:
Entre 0E0 et 1E1 :       7560653
Entre 1E1 et 2E1 :       1523844
```

```
Entre 2E1 et 3E1 :       457399
Entre 3E1 et 4E1 :       512979
Entre 4E1 et 5E1 :       16711
Entre 5E1 et 6E1 :       218
Entre 6E1 et 7E1 :       288953
Entre 7E1 et 8E1 :       9255
Entre 8E1 et 9E1 :       0
Entre 9E1 et 1E2 :       0
Entre 1E2 et 1E3 :       0
Entre 1E3 et 1E5 :       0
```

4a et 4b)

La première étape consiste à filtrer les données pour ne garder que les phases qui ne sont pas idle.

Ensuite, on effectue un mapping pour associer chaque job à ses durées, et on additionne ces durées par jobs. (`reduceByKey()`).

Pour la distribution du temps total d'accès au PFS par job, on effectue le même traitement qu'à la question 1.c) sur ces données regroupées par job.

Pour le top 10 des jobs en temps total d'accès au PFS, on extrait des données regroupées par job les 10 jobs ayant la plus grande durée totale avec la méthode (`top(10)`), en fournissant une implémentation d'un comparateur sérialisable.

RÉSULTAT 4a:

```
--- DISTRIBUTION DU TEMPS TOTAL D'ACCES AU PFS PAR JOB ---
Minimum:  2191.0
Maximum:  4.2177643615E10
Moyenne:  2.650477344731547E9
Mediane:  9.8439605E7
1er Quartile:  4.11719E7
3ème Quartile: 2.299096586E9
Histogramme:
Entre 1E0 et 1E2 :       0
Entre 1E2 et 1E3 :       0
Entre 1E3 et 5E3 :       7
Entre 5E3 et 1E4 :       0
Entre 1E4 et 2,5E4 :          1
Entre 2,5E4 et 5E4 :          16
Entre 5E4 et 7,5E4 :          6
Entre 7,5E4 et 1E5 :          1
```

```
Entre 1E5 et 5E5 :       54
Entre 5E5 et 1E6 :       39
Entre 1E6 et 1E7 :       243
Entre 1E7 et 1E8 :       734
Entre 1E8 et 2E9 :       342
Entre 2E9 et 2E10 :      612
```

RÉSULTAT 4b:

```
--- TOP 10 JOBS EN TEMPS TOTAL D'ACCES AU PFS ---
Numero 1 : Job 1042, duree : 4.2177643615E10
Numero 2 : Job 1785, duree : 3.9050398536E10
Numero 3 : Job 1651, duree : 3.7837636729E10
Numero 4 : Job 1676, duree : 3.7829062844E10
Numero 5 : Job 1691, duree : 3.7819742364E10
Numero 6 : Job 1728, duree : 3.7760075696E10
Numero 7 : Job 1662, duree : 3.7738047747E10
Numero 8 : Job 1724, duree : 3.7735870176E10
Numero 9 : Job 1637, duree : 3.7733002887E10
Numero 10 : Job 1647, duree : 3.7732909749E10
```

5)

Pour cette question, il nous suffit de filtrer les données pour garder seulement garder les lignes ou les phases sont idle (colonne pattern = -1).
Ensuite il suffit de garder les durées dans un `JavaDoubleRDD` (comme la question 1.b)
Enfin, on applique la fonction `.sum()` sur ce `JavaDoubleRDD` pour obtenir la somme des durées des phases IDLE, soit la durée totale IDLE du système.

RÉSULTAT 5 :

```
Resultat en microsecondes:    6.519558654649968E11
```

6a et 6 b)

Pour calculer les pourcentage, on filtre d'abord les données pour ne garder que les non IDLE. On stocke ensuite la somme des durées de toutes ces données filtrées.

Ensuite, pour chaque pattern (les 22), on effectue 2 filtrages sur les données IDLE, un pour garder les données ou le pattern apparaît seul, un autre pour garder les données où le pattern apparaît en concurrence avec d'autre.

Pour chacune de ces 2 sets de données, on calcule le pourcentage de leur durée par rapport à la durée totale (calculée au tout début). Pour ce faire on transforme notre `JavaPairRDD` en `JavaDoubleRDD` grâce à la fonction `mapToDouble()` (dans laquelle on retourne la durée).
Ensuite on calcule leur représentativité en additionnant le pourcentage où le pattern est seul et le pourcentage où il est en concurrence.

Enfin on sauvegarde dans une TreeMap (pour trier les données) avec en clé, la représentativité, et en valeur le pattern.
Pour avoir le top 10 il suffit de restreindre la taille de la map à 10 et de la parcourir pour avoir le top 10 (de 10 à 1).

(Fonctionne en mode local mais pas en mode cluster)
RÉSULTAT 6a:

```
SEUL 0 en millisecondes: 1.180763646E10 sur 7.79415020564E11. Soit 1.514935707994922% du temps total
EN CONCURRENCE 0 en millisecondes: 1.21278608572E11 sur 7.79415020564E11. Soit 15.560209307262312% du temps total
SEUL 1 en millisecondes: 2.1002879263E10 sur 7.79415020564E11. Soit 2.694697780882117% du temps total
EN CONCURRENCE 1 en millisecondes: 6.93631614608E11 sur 7.79415020564E11. Soit 88.99387313655754% du temps total des phases
SEUL 2 en millisecondes: 2.151650077E9 sur 7.79415020564E11. Soit 0.2760596114048487% du temps total
EN CONCURRENCE 2 en millisecondes: 6.3894634204E11 sur 7.79415020564E11. Soit 81.97767879526442% du temps total
SEUL 3 en millisecondes: 1.4529076959E10 sur 7.79415020564E11. Soit 1.864100200235617% du temps total
EN CONCURRENCE 3 en millisecondes: 6.27394121901E11 sur 7.79415020564E11. Soit 80.49551334627928% du temps total
SEUL 4 en millisecondes: 8.970488052E9 sur 7.79415020564E11. Soit 1.150925734727152% du temps total
EN CONCURRENCE 4 en millisecondes: 1.42506730927E11 sur 7.79415020564E11. Soit 18.28380608111444% du temps total
SEUL 5 en millisecondes: 9442927.0 sur 7.79415020564E11. Soit 0.0012115402899429515% du temps total
EN CONCURRENCE 5 en millisecondes: 6.28489054463E11 sur 7.79415020564E11. Soit 80.63599467305788% du temps total
SEUL 6 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 6 en millisecondes: 1.258587443E9 sur 7.79415020564E11. Soit 0.1614784690817559% du temps total
SEUL 7 en millisecondes: 1.16047992E8 sur 7.79415020564E11. Soit 0.01488911413537109% du temps total
EN CONCURRENCE 7 en millisecondes: 6.02112855E8 sur 7.79415020564E11. Soit 0.07725189265204299% du temps total
SEUL 8 en millisecondes: 2.100631995E9 sur 7.79415020564E11. Soit 0.2695139225672019% du temps total
EN CONCURRENCE 8 en millisecondes: 5.827679856E10 sur 7.79415020564E11. Soit 7.476991977628268% du temps total
SEUL 9 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 9 en millisecondes: 1.539216011E10 sur 7.79415020564E11. Soit 1.974834934392454% du temps total
```

```
SEUL 10 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 10 en millisecondes: 5.744740233E9 sur 7.79415020564E11. Soit 0.7370579320941227% du temps total
SEUL 11 en millisecondes: 1.71481002E8 sur 7.79415020564E11. Soit 0.022001244199260233% du temps total
EN CONCURRENCE 11 en millisecondes: 5.6247791032E10 sur 7.79415020564E11. Soit 7.216667570930054% du temps total
SEUL 12 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 12 en millisecondes: 3.227833814E9 sur 7.79415020564E11. Soit 0.4141354386093658% du temps total
SEUL 13 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 13 en millisecondes: 3.418019169E10 sur 7.79415020564E11. Soit 4.385364765650339% du temps total
SEUL 14 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 14 en millisecondes: 10052.0 sur 7.79415020564E11. Soit 1.2896851786005067E-6% du temps total
SEUL 15 en millisecondes: 7.0512498E7 sur 7.79415020564E11. Soit 0.009046848744199948% du temps total
EN CONCURRENCE 15 en millisecondes: 4.2268603E7 sur 7.79415020564E11. Soit 0.005423118862838134% du temps total
SEUL 16 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 16 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
SEUL 17 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 17 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
SEUL 18 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 18 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
SEUL 19 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 19 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
SEUL 20 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 20 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
SEUL 21 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
EN CONCURRENCE 21 en millisecondes: 0.0 sur 7.79415020564E11. Soit 0.0% du temps total
```

RÉSULTAT 6b:

```
10 : Pattern 9 avec 1.974834934392454% du temps total.
9 : Pattern 13 avec 4.385364765650339% du temps total.
8 : Pattern 11 avec 7.238668815129314% du temps total.
7 : Pattern 8 avec 7.74650590019547% du temps total.
6 : Pattern 0 avec 17.075145015257235% du temps total.
5 : Pattern 4 avec 19.434731815841594% du temps total.
4 : Pattern 5 avec 80.63720621334782% du temps total.
3 : Pattern 2 avec 82.25373840666927% du temps total.
2 : Pattern 3 avec 82.3596135465149% du temps total.
1 : Pattern 1 avec 91.68857091743966% du temps total.
```

7)

Pour cette question, nous avons effectué un filtrage permettant de ne garder que les données où tous les patterns en paramètres sont présents. Notre solution prends actuellement un temps linéaire en fonction du nombre d'éléments de la base.

Pour prendre un temps linéaire en fonction du nombre de résultats, un traitement préalable est nécessaire. Dans notre problème, nous recevons un jeu de données tous les jours (qui est formé des données du jour concaténées concaténées aux précédentes).
Le traitement préalable consiste à calculer pour chaque pattern toutes ses plages horaires, et les enregistrer dans un fichier (un fichier par pattern -> index inversé). On aura ainsi pour chaque pattern un fichier rassemblant toutes ses plages horaires.

Ainsi, lorsque les plages horaires de 4 patterns sont demandés, il suffit de lire les 4 fichiers associés à ces patterns, et écrire la concaténation de leur contenu dans un fichier résultat en appliquant le pattern “distinct” sur le timestamp de début et de fin de chaque phases pour supprimer les doublons.

8)
Etant donné que chaque jour nous recevons de nouvelles données, il faut, pour avoir les distributions à jour, recalculer tout. (ancienne + nouvelles données). Ainsi, plus les jours passent, plus le temps de calcul de chacune des questions augmentera.