



Projet Hadoop : Evaluation croisée

LAPLAGNE Chloé
RAZAFIMANANTSOA Nathan

Département Sciences du Numérique - Deuxième Année ASR
2020-2021

Sommaire

1	Partie Technique	3
2	Syntèse	5
3	Annexes	7

. L'objectif de ce rapport est de porter un regard critique sur la partie Hadoop du projet de Données Réparties. Dans un premier temps seront présentés les aspects techniques revus par un script et des tests, et finalement une synthèse sera faite proposant, s'il le faut, des pistes d'amélioration.

1 Partie Technique

. Pour témoigner du bon fonctionnement de cette partie, des programmes de tests ont été créés. Les parties de Hadoop interagissant avec des fichiers sont testées avec 3 exemples :

- `aragon.txt` : il s'agit du fichier donné comme exemple dans les fournitures (1.3 Ko)
- `medium.txt` : un fichier texte arrangé par ligne d'une taille de 4 Mo
- `big.txt` : idem mais avec une taille de 120 Mo.

L'objectif est d'évaluer sommairement les performances de l'application et de vérifier que la parallélisation apporte une amélioration des performances sur des fichiers plus volumineux.

La correction des résultats est évaluée sur la base des résultats de la classe fournie `Count.java`¹.

Test du Callback

. La classe `CallbackTest.java` permet de vérifier le bon fonctionnement de l'objet `CallbackImpl`. Il s'agit d'un test unitaire réalisé avec JUnit.

Son but est de vérifier que le Callback ne réveille le thread en attente que lorsque tous les appels à la méthode réveiller ont été émis. Le principe de fonctionnement du test est le suivant : on lance un nombre connu de Threads qui après un certain temps, feront chacun un appel à la méthode `reveiller()` de `CallbackImpl`. Un autre thread est placé en attente sur ce Callback et doit modifier une variable partagée à son réveil. Le test consiste alors à vérifier que la variable partagée a bien été modifiée (*ie* que le thread a bien repris) après un certain nombre d'appels à réveiller, ou qu'au contraire sa valeur n'est pas modifiée si tous les appels n'ont pas été effectués.

Ce test a été validé.

Test du Worker

. La classe `WorkerTest.java` permet de tester le bon fonctionnement de `WorkerImpl` et son temps d'exécution. Il s'agit d'un test unitaire réalisé avec JUnit. Pour pouvoir utiliser cette classe de tests, `CallbackDummy.java` permet de simuler un Callback (mais n'a aucun effet).

Le test vérifie dans un premier temps l'intégrité des résultats. On se base ici sur les résultats obtenus par un appel à `Count.java`. Les lignes du fichier de référence et de celui obtenu par `WorkerImpl` sont ensuite triées et comparées. Le test vérifie enfin si le temps d'exécution est raisonnable. On a pris ici un temps maximum d'exécution de :

$$t_{worker} \leq 1.1 \times t_{count}$$

Où t_{worker} est le temps d'exécution du worker sur un fichier donné et t_{count} le temps d'exécution de `Count` sur ce m.

Ce test a été validé sur les deux points évalués.

¹La destination du fichier `count-res` de sortie a été modifiée à `<Project.PATH>/count-res` pour pouvoir utiliser cette classe.

Test d'intégration

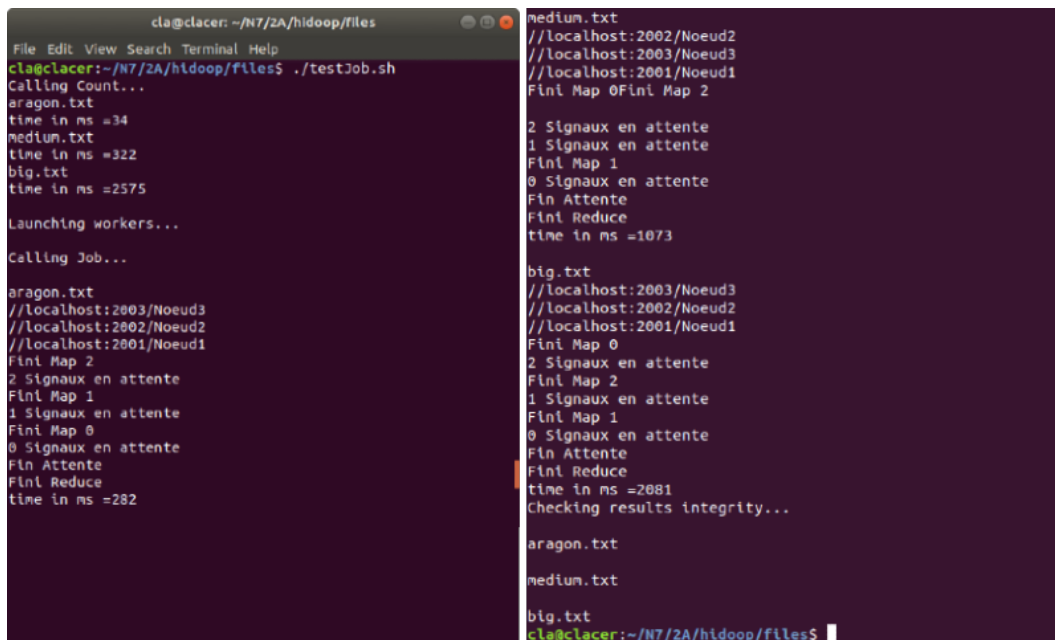
. La classe `Job` fait le lien entre les parties Map et Reduce de Hadoop. Il nous a donc semblé plus pertinent de la tester par le biais des tests de fonctionnement de cette version de Hadoop.

Le test est cette fois-ci réalisé sous forme d'un script bash. Il ne s'agit cependant pas d'un test en boîte noire. En effet l'application n'étant pas encore liée avec les fonctionnalités d'HDFS (notamment celle de lecture permettant de recombinaison des résultats du Map), il nous a fallu générer ces fichiers pour pouvoir utiliser l'application.

C'est le script `genJobChunks.sh` (voir en annexe) qui génère à partir d'un fichier donné 3 chunks de taille approximativement égale (en respectant les lignes). Le nom de chaque chunk respecte celui demandé par l'implémentation de `Job`. Dans un deuxième temps, `Job` nécessite aussi un fichier contenant les résultats des Maps concaténés (nommé "`<fname>-tot`"). Ce fichier est créé en joignant les résultats de `Count.java` sur ces chunks.

Après l'exécution de ce script, tout est prêt pour tester cette version de Hadoop. Le script `testJob.sh` permet de vérifier que l'exécution se déroule sans erreur, et que le résultat du Reduce est bien similaire à celui donné par un appel à `Count`. On ne revérifie pas ici les résultats individuels sur chaque chunk puisque cela a été fait dans le test unitaire `WorkerTest`.

A ce stade, l'application ne peut pas être lancée autrement qu'en local, et le fichier "`<fname>-tot`" est créé par avance. Nous n'avons donc pas fait d'assertion sur des contraintes temporelles d'exécution, ces données sont seulement indicatives.



```
cla@clacer: ~/N7/2A/hadoop/files
File Edit View Search Terminal Help
cla@clacer:~/N7/2A/hadoop/files$ ./testJob.sh
Calling Count...
aragon.txt
time in ms =34
medium.txt
time in ms =322
big.txt
time in ms =2575

Launching workers...

Calling Job...

aragon.txt
//localhost:2003/Noeud3
//localhost:2002/Noeud2
//localhost:2001/Noeud1
Fini Map 2
2 Signaux en attente
Fini Map 1
1 Signaux en attente
Fini Map 0
0 Signaux en attente
Fin Attente
Fini Reduce
time in ms =282

medium.txt
//localhost:2002/Noeud2
//localhost:2003/Noeud3
//localhost:2001/Noeud1
Fini Map 0
2 Signaux en attente
Fini Map 2
1 Signaux en attente
Fini Map 1
0 Signaux en attente
Fin Attente
Fini Reduce
time in ms =2081
Checking results integrity...

aragon.txt
medium.txt
big.txt
cla@clacer:~/N7/2A/hadoop/files$
```

Figure 1: Résultats de `testJob.sh`

Cependant, nous pouvons déjà faire quelques observations. Comme on s'y attendait, sur des petits fichiers comme `aragon.txt` et `medium.txt`, le temps de traitement est plus important qu'avec un simple appel à `Count`. Mais on remarque un gain de 16% sur le fichier `big.txt` (sur une dizaine d'exécutions, on a en moyenne 2,4s avec `Count` contre 2,0s avec `MyMapReduce`). Les résultats obtenus nous ont donc semblé pertinents sur ce test, et on note bien une amélioration du point de vue des performances en temps sur des fichiers 'volumineux'.

2 Syntaxe

. Dans cette partie, seront indiquées nos remarques selon différents critères pour la version actuelle du schéma Hidoop.

Correction

. Après exécution des tests, les classes `CallBackImpl.java` et `WorkerImpl.java` donnent des résultats conformes à ceux attendus (performance et résultats attendus). Le test d'intégration est également correct sur les aspects évalués et les performances nous ont semblé conformes à nos attentes.

Complétude

. Tous les points ont été abordés :

- La classe `Job.java` implante bien l'interface `JobInterface` et permet bien d'une part de lancer sur chaque noeud un thread réalisant le map sur chaque chunk. Après récupération du fichier (qui aurait un suffixe "-tot") grâce à HDFS, le reduce est ensuite fait localement et donne un fichier résultat avec un suffixe "-res".
- La classe `WorkerImpl.java` sera bien le thread qui permettra le map sur chaque chunk, et son fonctionnement a été validé par le test.
- La classe `CallBackImpl` permet bien de réveiller le processus lançant le map (`textbfstartJob` une fois que le map a été fait sur tous les chunks (par chaque Thread `WorkerImpl`).

Pertinence

. Le parallélisme du `CallBack` a été géré comme suit :

- Un objet `ReentrantLock` et des variables conditions permettent une communications entre threads :
- La méthode `startJob` permettra dans un premier temps de lancer les threads qui lanceront le map, et elle signalera à chaque chunk qu'elle attend leur résultat en appelant un `signal()` sur la condition `NoeudAttente`. Une fois le map réalisé par un thread, il signalera le processus dormant sur la condition `JobAttente`, qui sera reveillé lorsque tous les map auront été fini.
- Après signalisation de tous les thread, il y a possibilité de procéder au reduce.

Cette méthode de parallélisme semble cohérente dans le sens où elle permet un au reduce de se lancer uniquement lors de la réception de tous les map.

Cohérence

L'architecture est claire et les fonctions ne se recoupent pas. Cependant deux points nous poussent à la réflexion dans la classe `Job.java` :

- Pour les map, si le format des fichiers à map est kv, alors le format des fichiers résultant serait des fichiers LINE, nous ne voyons pas à quel cadre applicatif cela correspond.
- De même, une erreur est présente dans la conditionnelle pour déterminer les types de fichiers lors du reduce.
- Enfin, la variable `Project.PATH` ne semble pas être utilisée pour avoir un dossier data qui comprend le résultat des MapReduce.

Conclusion et pistes d'amélioration

. Après réalisations de tests (unitaires ou scripts), ou encore par étude du code en lui-même, l'application Hidoop réalisée semble remplir son cahier des charges, que ce soit en termes de performances ou de résultats.

Il ne nous reste plus qu'à voir ce que donnera l'intégration avec la partie HDFS.

Une piste d'amélioration que nous suggérons (mais pas obligatoirement, au vu du fait que l'on ne gère pas les pannes de noeuds), serait de définir un temps d'attente maximum pour le reveil du Callback, de sorte à ce que l'on attende pas indéfiniment si un problème survient au niveau d'un des Worker ou de la connexion.

3 Annexes

Tests

Classe CallbackTest permettant de vérifier le bon fonctionnement du Callback

```
1 import ordo.CallBackImpl;
2 import org.junit.Assert;
3 import org.junit.Before;
4 import org.junit.Test;
5
6 import java.rmi.RemoteException;
7 import java.util.concurrent.Executors;
8 import java.util.concurrent.ScheduledExecutorService;
9 import java.util.concurrent.TimeUnit;
10 import java.util.concurrent.atomic.AtomicInteger;
11
12 public class CallbackTest {
13     private CallBackImpl cb;
14     private AtomicInteger res;
15
16     /**
17      * Reinitialise le Callback et res avant chaque test
18      */
19     @Before
20     public void init() throws RemoteException {
21         cb = new CallBackImpl(5);
22         res = new AtomicInteger(0);
23     }
24
25     /**
26      * Verifier que le thread en attente est bien reveille au bout de nbNoeud
27      * appels a
28      * reveiller()
29      */
30     @Test
31     public void testCallback() throws InterruptedException {
32         ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(5);
33
34         Runnable run = () -> {
35             try {
36                 cb.reveiller();
37             } catch (RemoteException e) {
38                 e.printStackTrace();
39             }
40         };
41
42         // Programmer des appels a reveiller dans les 3 prochaines secondes
43         scheduler.schedule(run, 2, TimeUnit.SECONDS);
44         scheduler.schedule(run, 1, TimeUnit.SECONDS);
45         scheduler.schedule(run, 2, TimeUnit.SECONDS);
46         scheduler.schedule(run, 1, TimeUnit.SECONDS);
47         scheduler.schedule(run, 3, TimeUnit.SECONDS);
48
49         Thread verif = new Thread(() -> {
50             try {
51                 cb.attente();
52                 res.set(4);
53             } catch (InterruptedException e) {
54                 e.printStackTrace();
55             }
56         });
57
58         // Lancer le thread qui attend
59         verif.start();
60         // Attendre le temps que les appels soient effectues
61         Thread.sleep(4000);
62     }
63 }
```

```

61
62     // Verifier que la valeur de res a change
63     Assert.assertEquals(res.get(), 4);
64
65 }
66
67 /**
68  * Verifier que le thread reste en attente si nbNoeuds appels a reveiller() n'
69  * ont pas
70  * ete effectues
71  */
72 @Test
73 public void testCallbackAttente() throws InterruptedException {
74     ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(5);
75
76     Runnable run = () -> {
77         try {
78             cb.reveiller();
79         } catch (RemoteException e) {
80             e.printStackTrace();
81         }
82     };
83
84     // Programmer des appels a reveiller dans les 3 prochaines secondes
85     scheduler.schedule(run, 2, TimeUnit.SECONDS);
86     scheduler.schedule(run, 1, TimeUnit.SECONDS);
87     scheduler.schedule(run, 2, TimeUnit.SECONDS);
88     scheduler.schedule(run, 1, TimeUnit.SECONDS);
89
90     Thread verif = new Thread(() -> {
91         try {
92             cb.attente();
93             res.set(4);
94         } catch (InterruptedException e) {
95             e.printStackTrace();
96         }
97     });
98
99     // Lancer le thread qui attend
100    verif.start();
101    // Attendre le temps que les appels soient effectues
102    Thread.sleep(4000);
103
104    // Verifier que la valeur de res n'a pas change (le thread ne s'est pas
105    reveille)
106    Assert.assertEquals(res.get(), 0);
107
108 }
109
110 }

```

Classe CallBackDummy permettant de simuler le Callback lors des tests

```

1 import ordo.Callback;
2 import java.rmi.RemoteException;
3
4 /** Callback vide pour les tests */
5 public class CallbackDummy implements Callback {
6     @Override
7     public void attente() throws RemoteException {}
8
9     @Override
10    public void reveiller() throws RemoteException {}
11 }

```


Classe WorkerTest permettant de vérifier le bon fonctionnement d'un worker

```
1 import application.Count;
2 import application.MyMapReduce;
3 import config.Project;
4 import formats.Format;
5 import formats.KVFormat;
6 import formats.LineFormat;
7 import ordo.WorkerImpl;
8 import org.junit.Assert;
9 import org.junit.BeforeClass;
10 import org.junit.Test;
11
12 import java.io.File;
13 import java.io.IOException;
14
15
16 public class WorkerTest {
17
18     // Dossier ou sont les fichiers
19     final static String path = Project.PATH;
20     // Fichiers utilises comme chunks pour les tests
21     final static String[] files = {"aragon.txt", "medium.txt", "big.txt"};
22     // Durees d'execution par Count (sequentiel)
23     final static long[] durations = new long[files.length];
24
25     // Worker
26     WorkerImpl wi = new WorkerImpl();
27     // Implementation de Mapper utilisee
28     MyMapReduce mmr = new MyMapReduce();
29     // Callback sans effet pour le test
30     CallbackDummy dummy = new CallbackDummy();
31
32     public WorkerTest() throws IOException { }
33
34     /**
35      * Appel de Count pour chaque fichier
36      * On garde le temps d'execution et les fichiers de resultat comme reference
37      */
38     @BeforeClass
39     public static void initRefFiles(){
40         File fcount;
41         long s;
42         for (int i=0;i<files.length;i++) {
43             s = System.currentTimeMillis();
44             Count.main(new String[]{path + files[i]});
45             durations[i] = System.currentTimeMillis() - s;
46             fcount = new File(path + "count-res");
47             fcount.renameTo(new File(path+ files[i] + "-count-res"));
48         }
49         System.out.println("End init.");
50     }
51
52     /**
53      * Test d'un des fichiers en utilisant WorkerImpl
54      * On verifie si les resultats sont corrects et si la duree est acceptable
55      * (ie si elle est a 10% pres inferieure a celle mise par Count)
56      * @param index l'index dans files
57      */
58     private void testIndex(int index) throws IOException {
59
60         String output = path+files[index]+"-wk-res";
61         Format fwMap = new KVFormat(output);
62         Format frMap = new LineFormat(path+files[index]);
63
64         long t = System.currentTimeMillis();
65         // Execution du Map
66         wi.runMap(mmr, frMap, fwMap, dummy);
```

```

67 // Calcul de la duree d'exec
68 t = System.currentTimeMillis() - t;
69 System.out.println("Duree en ms : "+t);
70
71
72 // Comparer les fichiers (commande bash)
73 Process p = Runtime.getRuntime().exec("diff <(sort "+output+" ) <(sort "+
path+files[index]+"-count-res)");
74 String diff = new String(p.getInputStream().readAllBytes());
75
76 // Verifier que les 2 fichiers dont les lignes sont trieés sont les mÃªmes
77 Assert.assertTrue(diff.isBlank());
78
79 // Verification de la duree d'execution : ok si au temps d'execution de
Count
// a 10% pres
80 Assert.assertTrue(t <= (long)(durations[index]*1.1f));
81
82 }
83
84 @Test
85 public void testWorkerAragon() throws IOException {
86     testIndex(0);
87 }
88
89 @Test
90 public void testWorkerMedium() throws IOException {
91     testIndex(1);
92 }
93
94 @Test
95 public void testWorkerBig() throws IOException {
96     testIndex(2);
97 }
98 }

```

Scripts

Script genJobChunks permettant de créer des chunks pour tester cette version

```

1 #!/bin/bash
2
3 # Dossier de fichiers hidoop (egal a la variable Project.PATH)
4 folder=$1
5 # Chemin vers le repertoire contenant les .class
6 classes=$2
7 # Nombre de workers / chunks du fichier
8 N=3
9
10 # Fichiers de test
11 files=(aragon.txt medium.txt big.txt)
12
13 echo 'Making chunks...'
14 cd $classes
15 for f in ${files[@]}; do
16     # Decouper le fichier en N en conservant les lignes
17     split --number=1/$N -a 1 --numeric-suffixes=1 ${folder}$f ${folder}$f-
18
19     # Pour chaque chunks appeler Count puis concatener les resultats pour generer
20     # le fichier -tot qui sera transmis au Reducer (normalement resultat de hdfs.Read
    )
21     for i in $(seq 1 $N); do
22         java application.Count $folder$f-$i
23         mv ${folder}count-res ${folder}$f-${i}res
24     done
25     cat ${folder}$f-?res > ${folder}$f-tot
26 done

```

Script testJob permettant de tester Hidoop

```
1 #!/bin/bash
2
3 # Dossier de fichiers hidoop (egal a la variable Project.PATH)
4 folder=$1
5 # Chemin vers le repertoire contenant les .class
6 classes=$2
7 # Fichiers de test
8 files=(aragon.txt medium.txt big.txt)
9
10 cd $classes
11
12 # Pour chaque fichier compter les mots de mani re sequentielle avec Count
13 # Les resultats seront utilises comme references
14 echo 'Calling Count...'
15 for f in ${files[@]}; do
16     echo $f
17     java application.Count $folder$f
18     mv ${folder}count-res ${folder}$f-ref
19 done
20 echo ''
21
22 # Lancement des workers (qui effectuent le Map)
23 echo 'Launching workers...'
24 java ordo.WorkerImpl 1 & p1=$!
25 java ordo.WorkerImpl 2 & p2=$!
26 java ordo.WorkerImpl 3 & p3=$!
27 sleep 1
28 echo ''
29
30 # Appel a l'application pour chaque fichier
31 echo 'Calling Job...'
32 for f in ${files[@]}; do
33     echo ''
34     echo $f
35     java application.MyMapReduce $folder$f
36 done
37
38 # Tuer les workers
39 kill $p1
40 kill $p2
41 kill $p3
42
43 # Verification des resultats : les lignes du fichier genere par Hidoop et
44 # du fichier de reference sont trie es puis les fichiers sont compares
45 echo 'Checking results integrity...'
46 for f in ${files[@]}; do
47     echo ''
48     echo $f
49     diff <(sort $folder$f-ref) <(sort $folder$f-res)
50
51 done
```