Université de Bordeaux I

Programmation large échelle

Compte rendu du TP 8/9

Intro Spark 1/2
Intro Spark 2/2

Réalisé par :

GAMELIN Antoine
LASSOUANI Sofiane

Université Bordeaux 1 (2016–2017) – Master Génie Logiciel (UE PLE)

TABLE DES MATIÈRES

1 Introduction **1**
1.1 Introduction 1

2 Partie I Job Submit **2**
2.1 Introduction 2
2.1.1 Exercice 1 : test simple Spark 2
2.1.2 Exercice 2 : Lancement manuel d'un jar 2

3 Partie II WorldCities Back **3**
3.1 Introduction 3
3.1.1 Exercice I 3
3.1.2 Exercice II 3
3.1.3 Exercice III 3
3.1.4 Exercice IV 4
3.1.5 Exercice V 4

4 Résultat de sortie **5**

5 Conclusion **6**

Chapitre 1

Introduction

1.1 Introduction

Dans les TPs précédents nous avons travaillé avec MapReduce de hadoop pour pouvoir transférer la charge de travail sur différents processeur, pour pouvoir traiter des données importante.

Dans ce nouveau TP on avons utilisé le framework Spark qui permet de coder du traitement BigData plus simplement. Surtout pour lier les Mapper et Réduceur entre eux

Afin de démontrer la simplicité de ce framework dans ce TP il nous ai proposé d'implémenter de nombreux exercices des anciens TP en Spark.

Chapitre 2

Partie I Job Submit

2.1 Introduction

La partie I de ce TP consiste à se familiariser et comprendre le fonctionnement de Spark, en apprenant a lancer un job avec ce dernier et de lancer manuellement des jars.

2.1.1 Exercice 1 : test simple Spark

Le premier exercice consiste a tester le bon fonctionnement et l'installation de Spark, notamment en lançant notre première commande **run-example SparkPi 100**. Par défaut si on le précise pas le job se lance en local sur notre machine. Si nous souhaitons utiliser le yarn sur le réseau il est important de définir le master (Master=yarn-cluster)

2.1.2 Exercice 2 : Lancement manuel d'un jar

Maintenant qu'on a lancé notre job Spark en local et sur le cluster, nous allons voir dans cet exercice plus d'option, en particulier spark-submit qui nous permet d'exécuter un programme avec différentes options. Par exemple, nous pouvons définir le nombre d'exécuteurs, la mémoire, le nombre de coeur...

La commande qui nous permet de lancer manuellement un job spark avec 4 exécutors utilisant chacun deux 2 coeurs et 512méga de mémoire en utilisant spark-submit est :

spark-submit –deploy-mode cluster –master yarn –num-executors 4 –executor-cores 2 –executor-memory 512m spark-examples.jar 100

- L'option –master yarn permet de lancer le job sur le cluster
- L'option –num-executors permet de spécifier le nombre d'exécuteurs.
- L'option –executor-core permet de spécifier le nombre de coeur par executeur
- L'option –executor-memory permet de spécifier la mémoire de chaque executeur

Chapitre 3

Partie II WorldCities Back

3.1 Introduction

La partie II consiste a refaire quelque exercices des TPs précédent fait en MapReduce afin de constater la différence entre les deux framework.

3.1.1 Exercice I

L'exercice consiste à charger le fichier WorlCitiesPop dans un RDD et d'afficher le nombre de partitions de ce dernier et de l'augmenter aux nombres d'exécuteurs du cluster.

JavaRDD<String> rdd = context.textFile(inputPath) ; permet de charger le fichier WorlCitiesPop passé en paramètre et de le stocker dans un RDD de type chaîne de caractère.

Nous utilisons la méthode repartition, pour spliter le job sur les différents exécuteurs disponible que nous définissons lors de l'exécution de spark-submit : **rdd.repartition(numExecutors)**.

3.1.2 Exercice II

Cette exercice fait référence à l'exercice de filtrage du TP MapREduce 2/2. L'objectif c'est de compter le nombre de ville ou la population est renseigné, à partir de notre rdd qui contient l'intégralité du fichier, nous splittons réalisons un JavaRDD<String[]> de chaque ligne. Nous créons enfin un nouveau RDD contenant un Tuple2(String,double) qui contient la ville et la population. (Si cela n'est pas renseigné nous lui affectons la valeur -1. Et enfin pour finir, nous retournons un RDD avec un nouveau filter qui retire toutes les lignes où la population n'est pas renseignée.

3.1.3 Exercice III

Cette exercice consiste à faire des statistique sur notre fichier WorlCitiesPop, et cela en utilisant la fonction stats() du type StatCounter. La première etape consiste de transformer le **JavaRDD< Tuple2<String, Double> >** en JavaPairRDD. en utilisant la fonction **mapToDouble**. Par la suite nous récupérons que les valeur de ce JavaPair pour avoir un JavaDoubleRDD. (En utilisant la fonction values() ;)

Grâce au StatCounter, nous pouvons accéder aux minimum,maximum,la somme,la variance...

3.1.4 Exercice IV

Cet exercice consiste à générer l'histogramme de fréquences des population des villes ainsi que les statistique de chaque classe d'équivalence. Dans un premier temps nous étions parti sur l'idée du GroupBy mais nous avons fini par utiliser la fonction aggregateByKey() ;

Dans un premier temps, nous utilisons la fonction keyBy, qui permet de générer la key, qui nous donne la valeur de l'histogramme (**Math.floor(Math.log10(x))**)

Ensuite nous faisons un aggregateByKey, cela consiste à faire un GroupBy des clefs et de transformer la sortie en StatCounter. Le premier paramètre concerne l'initialisation, ensuite nous devons passer une fonction qui va s'exécuter localement sur l'exécuteur. Cette fonction de type Function2, reçoit en entrée des double (valeur des populations) et des StatCounter et elle fait un merge pour ajouter la valeur dans le statCounter.

Quand à la deuxième fonction elle récupère les StatsCounter de chaque exécuteurs pour les fusionner avec la fonction merge quand ils ont la même key.

Dans le cas d'utilisation d'un MapReduce de Hadoop, la première fonction correspond au combineur et la seconde correspond au réduceur et le keyBy serait le job du mapper.

3.1.5 Exercice V

Ce dernier exercice nous demande de réaliser une jointure (inner join) entre le fichier "WorlCitiesPop" et le fichier de "regions.txt", afin d'obtenir un fichier qui contient le nom de la ville et la région correspondante .

De la même méthode que l'exercice 1 nous chargeons le fichiers regions_codes.csv ; ensuite nous créons un rdd contenant la concaténation du code_region et num_region que nous prenons soins de mettre en lowercase pour pouvoir la même key dans les deux fichiers.

Pour réaliser le Join il faut que les rdd soit de type : JavaPairRDD , nous utilisons donc la fonction JavaPairRDD.fromJavaRDD utilisé dans l'exercice 3.

Puis enfin nous faisons la jointure de ces deux RDD. Le collect de ce RDD étant très coûteuse, nous avons préférés enregistrer le resultat sur le cluster en utilisant la fonction saveAsTextFile.

Chapitre 4

Résultat de sortie

===Exercice 1===
Num executors : 4
Num partitions before repartition : 2
Num partitions after repartition : 4
===Exercice 2===
Nombre de lignes valide :47980
===Exercice 3===
Min : 7.0
Max : 3.1480498E7
Sum : 2.289584998999999E9
Mean : 47719.570633597315
Stdev (Standard deviation) : 302885.559204037
Variance : 9.17396619743422E10
===Exercice 4===
(0,(count : 5, mean : 7,200000, stdev : 0,400000, max : 8,000000, min : 7,000000)),
(1,(count : 174, mean : 55,385057, stdev : 26,299688, max : 99,000000, min : 10,000000)),
(2,(count : 2187, mean : 570,802469, stdev : 260,682724, max : 999,000000, min : 100,000000)),
(3,(count : 20537, mean : 4498,947266, stdev : 2421,290672, max : 9998,000000, min : 1000,000000)),
(4,(count : 21550, mean : 30600,956659, stdev : 21259,650752, max : 99922,000000, min : 10001,000000)),
(5,(count : 3248, mean : 249305,112993, stdev : 179813,589177, max : 997545,000000, min : 100023,000000)),
(6,(count : 269, mean : 2205586,553903, stdev : 1614945,466745, max : 9797536,000000, min : 1001553,000000)),
(7,(count : 10, mean : 13343569,500000, stdev : 6188097,825951, max : 31480498,000000, min : 10021437,000000))
===Exercice 5===

Chapitre 5

Conclusion

Ce TP nous a initié au framework Spark, ce qui nous a permis de voir qu'il est plus rapide de mettre en place des maps/reduce cependant il faut faire très attention aux choix des fonctions pour optimiser le temps d'execution.

Mise a part quelque difficultés rencontrés lors de ce TP Spark reste un framework qui nous a permis de gagner du temps. Notamment le join qui est très simple d'implémentation et qui est très compacte par rapport à ce qu'on a fait sur Hadoop.