ABD - Rapport de travail

Introduction

L'objectif de ces séances de TP est de nous introduire au framework de calculs distribués Spark. En premier lieu, nous tâcherons de mettre en place notre environnement de développement, pour ensuite commencer par un exercice de base, le calcul d'une intégrale. Pour finir, nous tenterons d'utiliser Spark dans un contexte plus approprié à son utilisation: sous forme de cluster.

Installation de l'environnement

Pour commencer ce TP, la première étape a été la création de la machine virtuelle sur laquelle nous allions travailler. Pour ce faire, nous avons donc créé une machine grâce au système de VM de l'ISTIC. Suite à la mise en place de la VM, nous avons installé les éléments nécessaires au démarrage du TP: Scala, sbt et Git.
Une fois cette étape terminée, nous avons pris un moment pour comprendre, plus en détails, le fonctionnement interne de Spark ainsi que le lancement de notre première application de test. Les éléments fournis dans la documentation de TP nous ont permis de commencer par une fonction simple: le calcul de Pi. Sur Internet, (https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/SparkPi.scala) nous avons utilisé un guide de démarrage pour mettre en place l'arborescence de dossiers et fichiers pour le lancement grâce à la commande "*sbt package*".
Cette étape nous a pris du temps car nous avions fait des erreurs lors de l'installation de l'environnement (arborescence, installation de Spark). Cependant, nous avons eu le temps de comprendre le fonctionnement de ce framework.

Expérimentation sur une unique machine

Integrale

L'objectif est d'approximer la valeur de l'intégrale de 1/10 à l'aide de la méthode des rectangles. Le résultat devrait être ln(10) = 2.30258509299…
Pour la partie développement, nous nous sommes inspirés du code de *SparkPi*, en reprenant les concepts des "slices" et les méthodes propres à Spark tel que "parallelize" qui permet de paralléliser le travail entre plusieurs threads.
Dans notre fonction, nous prenons en compte le nombre de "slices", que l'utilisateur est invité à entrer, ainsi que le nombre de rectangles variant de 10 à 10000.

Ensuite, nous effectuons l'enchaînement d'un map (Transformation) et d'un reduce (Action) qui nous permet dans un premier temps de créer une map et de la remplir avec la valeurs correspondantes, puis d'additionner en cascade les résultats afin d'obtenir un résultat final affichable.

Suite à l'exécution du programme (voir Git pour le code source), nous avons dressé des tableaux comparatifs des résultats obtenus

Résultats :

Pour 10 slices				
Nombre de rectangles	10	100	1000	10000
résultat	1,87021443	2,253752798	2,297641775	2,30209016
temps (en ns)	1131434991	80989079	75956720	82946735

Pour 100 slices				
Nombre de rectangles	10	100	1000	10000
résultat	1,87021443	2,253752798	2,297641775	2,30209016
temps (en ns)	1499846587	356690711	321413074	278316729

D'après les résultats obtenus, nous pouvons constater que le nombre de rectangles joue un rôle majeur dans la précision de notre calcul d'intégrale. Le résultat ne devient correct (précision de 2 chiffres après la virgule) qu'à partir de 1000 rectangles calculés.
Pour ce qui est du temps d'exécution, nous pouvons remarquer que le premier calcul est le plus long. Notre explication à ce phénomène est la mise en mémoire cache des instructions à exécuter : Les quantums de temps alloués par l'ordonnanceur sont trop grands et donc trop peu précis. De plus, il y a un temps de chauffe et des moyens de compilation en natif si le code est utilisé plusieurs fois, le premier tour de boucle est donc plus long que la suite.

Geolocation

La deuxième partie des expérimentations sur une seule machine avait pour but d'étudier un ensemble de données. En l'occurrence, nous partions d'un fichier contenant une liste de 16240 utilisateurs de San Francisco qui utilisent FourSquare. Chaque ligne de ce fichier présentait un utilisateur ainsi que les lieux qu'il a visité sous la forme suivante :

<user_Id> { <location_Id>","<frequency> }*

A partir de ces informations, notre programme Spark devait être en mesure de calculer ces différentes valeurs:

- Nombre total de visite par lieux (lieux le plus visité en premier)
- Nombre total de visiteurs par lieux (lieux le plus visité en premier)
- Moyenne de visite par utilisateur par lieux (moyenne la plus élevée en première)

Pour suivre un raisonnement logique, nous avons commencé par la première instruction.
La première étape de notre développement a été de réaliser les lignes omniprésentes, peu importe l'instruction: l'initialisation.

```
object Geoloc {
        def main(args: Array[String]) {
                val spark = SparkSession
                .builder
                .appName("Geoloc")
                .getOrCreate()


                var textFile = spark.sparkContext.textFile("./src/main/scala/onlyBay.txt")
                textFile = textFile.map(line => line.replaceFirst("[A-Za-z0-9]* ", ""))
                val lineSplitted = textFile.flatMap(line => line.split(" "))
                val byPaire = lineSplitted.map(line => line.split(","))
```

Ces lignes de code permettent de mettre en place, de manière générique, notre ensemble de données. A la fin de ces étapes nous obtenions une "map" de la forme :

{ (<location_Id>,<frequency>)* }

Nous avons perdu du temps à lire le fichier, ainsi qu'à suivre un cheminement correct pour le "découpage" des données. Cependant, des lectures et des exemples nous ont permi de progresser doucement jusqu'à atteindre un état correct.

Par la suite, nous somme partis de l'état générique pour répondre aux différentes questions.

Question 1 :

Nous avons mappé chaque ligne afin de lui donner le format suivant : {(<location_Id>,<frequency>)*} avec frequency un entier

Puis nous avons réduit la map grâce à reduceByKey afin d'additionner les visites de chaque personne par location. Ensuite nous avons trié les résultats par ordre décroissant. Puis nous avons écrit dans un fichier de sortie

Question 2 :

Nous avons mappé chaque ligne afin de lui donner le format suivant : {(<location_Id>, 1)*}

Cela nous a permis de compter le nombre de visiteur d'un certain lieu en simulant une seule visite par lieu par visiteur. Nous avons ensuite réduit par clé et triés les résultats par ordre décroissant comme dans la question 1.

Question 3 :

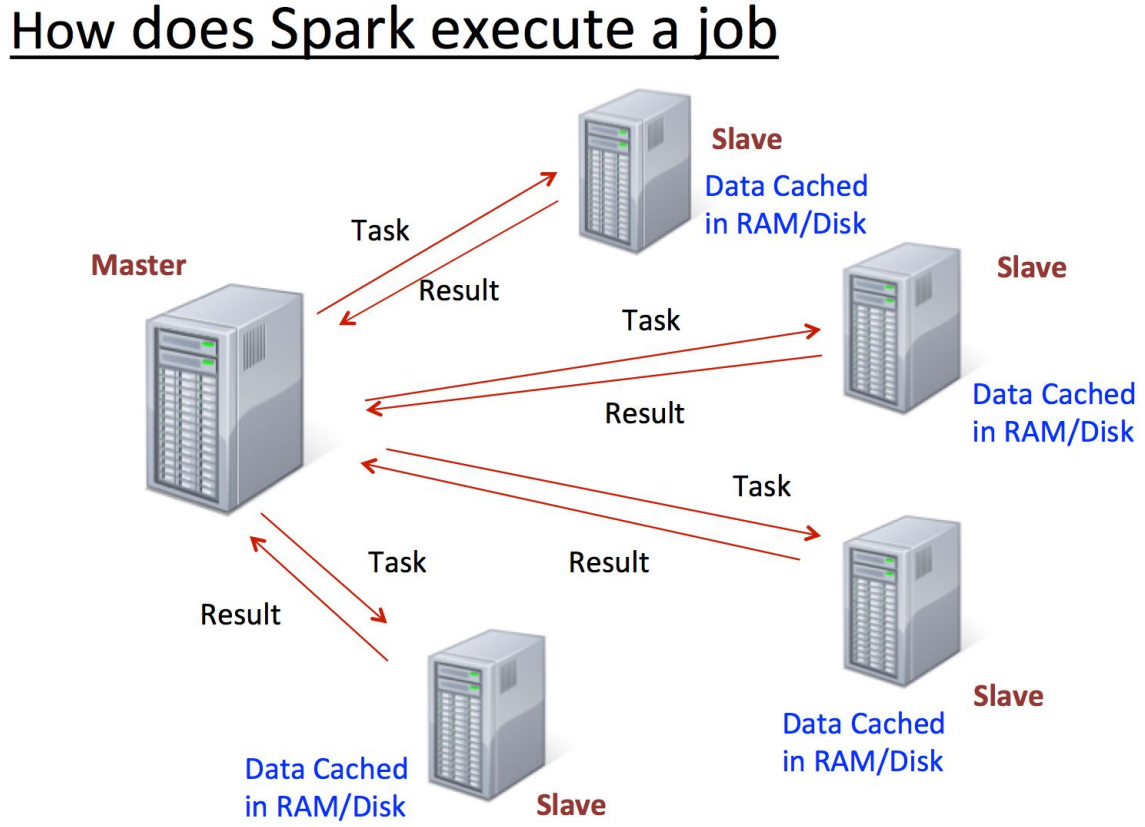
Pour cette dernière question, nous avons mappé chaque ligne afin de lui donner le format suivant : {(<location_Id>, {<frequency>})*} avec frequency une liste de float (possédant un seul élément).

Nous avons ensuite réduit la map en utilisant reduceByKey et en concaténant les listes. Nous obtenons donc le format suivant : {(<location_Id>, {<list of frequencies>})*}

La dernière étape consiste simplement à utiliser mapValues et à lui faire calculer la somme des visites / la taille de la liste (qui correspond au nombre de visiteurs). Cela nous donne la moyenne des visites par lieu par visiteur.

Cluster de machines

L'objectif final de ce TP est d'utiliser toutes les capacités de Spark. Or Spark permet d'utiliser des clusters de machines, c'est à dire un noeud central appelé "Master" et des noeuds secondaires appelés "Slaves".



Pour utiliser cette fonctionnalité, il faut plusieurs machines et installer Spark.
Le noeud principal se définit comme master à l'aide de la commande suivante :
./sbin/start-master.sh

Les autres noeuds se définissent comme slaves du master grâce à la commande suivante :
./sbin/start-slave.sh spark://ESIR-ABD-GUIHAIRE-PERRET.istic.univ-rennes1.fr:7077

Une fois les rôles attribués, le cluster devient transparent pour l'utilisateur. Le programme est lancé sur le master qui va ensuite répartir et envoyer le travail à effectuer sur les différentes machines slaves, puis récupérer les résultats.

Le cluster est donc un très bon moyen de traiter un grand nombre de données en parallèle.