Le framework Spark

Spark est un outil permettant de faire du traitement de larges volumes de données, et ce, de manière distribuée, du cluster computing. Le framework offre un modèle de programmation plus simple que celui d'Hadoop et permet des temps d'exécution jusqu'à 100 plus courts en mémoire et 10 fois plus cours sur disque.
Spark est né dans le laboratoire AMPLab de l'université de Berkeley et partant du principe que : d'une part la RAM coûte de moins en moins cher et les serveurs en ont donc de plus en plus à disposition ; d'autre part, beaucoup d'ensemble de données dits BigData ont une taille de l'ordre de 10Go et tiennent donc en RAM.
Le projet a intégré l'incubateur Apache en juin 2013 et est devenu un « Top-Level Project » en février 2014.
L'écosystème Spark comporte donc plusieurs outils :
- Spark pour les traitements en batch
- Spark Streaming pour le traitement en continu de flux de données
- Mllib pour le machine learning
- GraphX pour les calculs de graphes
- Spark SQL, une implémentation SQL-like d'interrogation de données

Spark est capable de s'intégrer parfaitement avec l'écosystème Hadoop, notamment avec HDFS et des intégrations avec Cassandra et ElasticSearch sont prévues.
Ce framework est écrite en Scala et propose un binding Java qui permet de l'utiliser sans problème en Java.

Notions de Base
L'élément de base à manipuler est le RDD : Resilient Distributed Dataset. Il s'agit d'une abstraction de collection sur laquelle les opérations sont réalisées de manière distribuée tout en étant tolérante aux pannes matérielles. Le traitement que l'on écrite semble ainsi s'exécuter au sein de notre JVM mais il sera découpé pour s'exécuter sur plusieurs nœuds. En cas de perte d'un nœud, le sous-traitement sera automatiquement relancé sur un autre nœud par le framework sans impacté sur le résultat.
Les éléments manipulés par le RDD peuvent être des objets simples comme des String ou des Integer, nos propres classes ou plus couramment des tuples. Dans ce dernier cas les opérations offertes par l'API permettront de manipuler la collection comme une map clé-valeur.
L'API exposé par le RDD permet d'effectuer des transformations sur les données :
- map() permet de transformer un élément en un autre
- mapToPair() permet de transformer un élément en un tuple clé-valeur
- filter() permet de filtrer les éléments en ne conservant que ceux qui correspondent à une expression
- flatMap() permet de découper un élément en plusieurs autres éléments
- reduce() et reduceByKey() permet d'agréger des éléments entre eux.
…

Ces transformation ne s'exécuteront que si une opération finale est réalisée en bout de chaine. Les opération finales sont :
- count() pour compter les éléments
- collect() pour récupérer les éléments dans une collection Java dans la JVM de l'exécuteur(dangereux en cluster)
- saveAsTextFile() pour sauver le résultat dans des fichiers texte.
….

Premier exemple :

Pour commencer, il faut d'abord créer un contexte Spark. Puisque nous écrivons du Java, la classe que nous utilisons est JavaSparkContext et nous lui passons un objet de configuration contenant :
- un nom d'application
- la référence vers un cluster Spark à utiliser, en l'occurence « local » pour exécuter les traitements au sein de la JVM courante.
Ex :

```
SparkConf conf = new SparkConf()
                ,setAppName(« arbres-alignement »)
                .setMaster(« local ») ;
JavaSparkContext sc = new JavaSparkContext(conf) ;
```

suite du traitement :

```
long count = sc.textFile(<nomdufichierenentré>)
                .filter(<element> → <traitement>)
                .map(<element> → <traitement>)
                .map(field → <traitement>)
                .filter(height → <traitement>)
                .count() ;
System.out.println(count) ;
```

Le code écrit avec Spark présente l'intérêt d'être à la fois compact et lisible.

MapReduce avec Spark

Avec Spark comme avec Hadoop, une opération de Reduce est une opération qui va permettre d'agréger les valeurs 2 à 2,en procédant par autant d'étapes que nécessaire pour traiter l'ensemble des éléments de la collection. C'est ce qui permet au framework de réaliser des agrégations en parallèle, éventuellement sur plusieurs nœuds.
Le framework va choisir 2 éléments et les passer à une fonction qu'il faut définir au préalable. La fonction doit retourner le nouvel élément qui remplacera les 2 premiers.
Il en découle que le type reçu en entrée doit être le même que celui en sortie de fonction : les valeurs doivent être homogènes. C'est nécessaire pour que les opérations soit répétée jusqu'à ce que l'ensemble des éléments aient été traités.

Nous pouvons calculer la hauteur moyenne des arbres de notre fichier en réalisant :

- une opération d’agrégation (somme) des hauteurs
- un comptage des éléments
- une division de la première valeur par la deuxième.

Puisque l’agrégation et le comptage se basent sur le même fichier et que les premières opérations de traitement sont identiques, nous pouvons réutiliser le même RDD. Nous allons utiliser l’opération `cache()` qui permet de mettre le RDD en cache en mémoire. Les calculs ne seront donc exécutés qu’une seule fois et le résultat intermédiaire sera directement utilisé pour les deux opérations

d’agrégation et de comptage.

```
JavaRDD<float> rdd = sc.textFile(“data/arbresalignementparis2010.csv”)
.filter(line -> !line.startsWith(“geom”))
.map(line -> line.split(“;”))
.map(fields -> Float.parseFloat(fields[3]))
.filter(height -> height > 0)
.cache();
float totalHeight = rdd.reduce((x, y) -> x + y);
long count = rdd.count();
System.out.println(“Total height: ” + totalHeight);
System.out.println(“Count: ” + count);
System.out.println(“Average height ” + totalHeight / count);
```

Spark Streaming permet un traitement de données produites en flux continu, par opposition à Spark qui permet de traiter des données connues à un instant
données