Initiation à Apache Spark avec Java

Posted on **avril 14, 2015**

En cette édition 2015 de Devoxx France, **Apache Spark** est l'une des technologies qui se démarque, comme le furent Docker et Java 8 en 2014 ou AngularJS en 2013. Connu pour être le digne successeur d'Hadoop, le framework Spark fait partie des outils Big-Data que j'ai découvert lors de la conférence NoSQL Matters 2015.

Présenté par Hayssam Saleh et Olivier Girardot, le Hands-on-Lab « Initiation à Spark avec Java 8 et Scala » était donc l'occasion idéale pour m'initier en pratique aux fonctionnalités proposées par Spark et découvrir l'univers du **Machine Learning**.

Si vous n'avez pas eu la chance de pouvoir assister à ce Lab, toutes les ressources utilisées lors du Lab ont été mises en lignes pour le suivre en offline (ou le terminer à la maison).

1. Un gitbook Initiation à Spark avec Java 8 et Scala. Avec ses 33 pages, ce livre contient à la fois la présentation réalisée en séance par les speakers ainsi que les intitulés des exercices.
2. Les jeu de données au format CSV et JSON nécessaires pour le Lab
3. La configuration maven pour Java ou sbt pour Scala.

Bien qu'ayant particulièrement bien préparés leur présentation, Hayssam et Olivier ont surestimé la vélocité de leur auditoire. Nous n'avons en effet eu le temps que de coder 3 des 9 workshops prévus initialement. Les présentateurs ont donné aux participants le choix d'utiliser Java 8 ou Scala. La grande majorité de l'assistance a choisi Java ; ce fut également mon cas. Dans ce billet, je compte vous restituer ce que j'ai appris au cours de ces 3h de Lab. Je vous relaierai les **bonnes pratiques** dispensées par Hayssam qui a récemment passé la certification Spark. Pour vous aider, j'ai publié le code Java sur le projet github initiation-spark-java.

Hello World avec Spark

Le pré-requis à l'utilisation de Spark est de disposer d'un JDK 8.
La distribution Apache Spark 1.3 est multi-OS. Une fois l'archive dézippée, on peut vérifier son

fonctionnement en utilisant son shell en ligne de commande *bin/spark-shell.sh* puis en exécutant une première commande Scala :
sc.parallelize(1 to 1000).foreach(println)

Cette commande affiche 1000 nombres de manière non ordonnée. La fonction *parallelize* construit une **collection distribuée**. La liste peut être répartie sur plusieurs machines et/ou plusieurs cœurs. Ici, elle est répartie sur un **cluster local**. Sur un Macbook disposant de 8 cœurs, 8 partitions sont créées. Chaque nombre est affiché sur la console. L'ordre d'affichage n'est pas prédictif.

Spark, mais pourquoi faire ?

Apache Spark est un **framework de calcul distribué** s'inscrivant dans la mouvance BigData. Il s'adresse aussi bien aux datascientists qu'aux développeurs.

Apparu en 2010, Spark est le digne successeur du pattern d'architecture Map/Reduce mis en œuvre dès 2002 chez Google.
Map/Reduce est conçu comme un batch distribué : pas d'interaction utilisateur, pas de temps réel … Spark permet quant à lui de réaliser des traitements au **fil de l'eau**. Il permet de réaliser des **micro-batchs.** Quitte à perdre des données, Spark met tout en œuvre pour aller le plus rapidement possible en profitant de leur co-localité.
Spark inclue la librairie d'algorithmes de Machin Learning MMLib.

Ecrit en Scala, Spark est pensé pour être utilisé Scala. Néanmoins, Spark propose une API Java. Par apport à l'API Scala, l'API Java souffre souvent de petits retards.
D'après l'expérience d'Hayssam Saleh, la taille des programmes Spark tourne autour des 700 lignes de code Scala.

Les concepts de Spark

LE RDD

Le concept fondamental de Spark est le **RDD**, pour **Resilient Distributed Dataset.** Il s'agit d'une **structure de données**, **immuable**, **itérable** et complètement **lazy**.

Cette structure représente un **graphe acyclique ordonnée** (de la même manière que les commits Git) des différentes **opérations à appliquer aux données** chargées par Spark. Il s'agit en quelque sorte d'**un plan d'exécution**.

Tout traitement Spark commence par le chargement d'un RDD. Spark permet de charger les données depuis plusieurs sources : HDFS, un fichier texte, une structure en mémoire, des données sérialisées, des données ou des SequenceFile Hadoop …
Transformations et actions

Les 2 concepts que sont les **transformations** et les **actions** s'appliquent au RDD.
Les transformations sont empilables sur les RDD. Les RDD étant immutables, une transformation crée donc un nouveau RDD. Les transformations sont paresseuses. Cela signifie qu'elles ne sont pas évaluées tout de suite.

Pour véritablement lancer le traitement sur un cluster ou les CPU locaux, on passe par une action. **Les actions sont terminales**. Lorsqu'on lancer une action, on retourne à la JVM (appelant). Une seule action peut être réalisée par RDD.

Dans l'API Spark, le nom des méthodes ne permettent pas de déterminer si l'on a affaire à une opération ou d'une action. Pour cela, il faut regarder le type de retour : s'il s'agit d'un RDD, la méthode est une transformation, sinon c'est une action.

LE SPARK CONTEXT

Le SparkContext est la couche d'abstraction permettant à Spark de savoir où il va exécuter les traitements. Dans le code, il est généralement matérialisé par la variable *sc*.
Une fois les développements terminés, le SparkContext est redéfini lors du déploiement du binaire sur un cluster de 50 machines.
En Java, on privilégie l'utilisation du JavaSparkContext : il retourne des JavaRDD et sait manipuler les collections Java.

Chargement d'un premier RDD en Java

Comme nous l'avons vu, tout programme Spark commence par le chargement d'un RDD. Afin de boostraper notre IDE, nous allons créer un projet Java chargeant un RDD depuis un fichier CSV.
On commence par écrire un pom.xml contenant la dépendance vers **spark-core** :

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-core_2.11</artifactId>
    <version>1.3.1</version>
</dependency>
```

A noter que lors du Lab, la version 1.3.1 de Spark n'était qu'en Release Candidate et qu'il était donc nécessaire d'ajouter un repository pour les early adopters. La version 1.3.1 permet de pleinement utiliser Spark 1.3 en Java.

Un *mvn dependency:tree* permet de lister toutes les librairies sur lesquelles se base Spark : Hadoop, Jackson, Metrics et bien entendu Scala.
Le suffixe 2.11 de l'artefact spark-core correspond à la version de Scala utilisée. Ainsi, pour Scala 2.10, il existe l'artefact spark-core_2.10.

Le scope maven par défaut *compile* est utilisé ici afin de pouvoir exécuter le code Spark depuis un IDE. Afin de pouvoir déployer le code sur un cluster Spark, il sera nécessaire de le positionner à *provided* et de construire un unique JAR à l'aide du plugin maven assembly.

Voici le code source de la classe FirstRDD affichant le nombre de lignes contenues dans le fichier rating.txt.

```
public class FirstRDD {

    public static void main(String[] args) {
```

```
        SparkConf conf = new SparkConf().setAppName("Workshop").setMas
        JavaSparkContext sc = new JavaSparkContext(conf);

        String path = Paths.get(FirstRDD.class.getResource("/ratings.t
        JavaRDD<String> lines = sc.textFile(path.toString());
        System.out.println("Lines count: " + lines.count());
    }
}
```

Le paramètre *« local[*] »* précise à Spark d'exécuter les traitements sur un cluster local et en profitant de tous les cœurs disponibles.

La lecture du fichier texte à la mode Java 7 renvoie un RDD de String.
L'appel à la méthode *count()* déclenche une action. 100 000 lignes comptabilisées.

Les fichiers de type CSV sont particulièrement bien adaptés à Spark : chaque ligne correspond à un élément. Néanmoins, Sparl offre la possibilité de charger du JSON à l'aide du parseur Jackson. On utilise alors un RDD de tuple RDD[(String, String)] : le path est la clé, la valeur est le contenu entier du fichier JSON.

Sparl permet également de charger le RDD à partir d'une source de données JDBC. 3 éléments sont nécessaires : une connexion JDBC, la requête JDBC et ses paramètres et une classe chargée de lire une ligne du *ResultSet*.

A des fins de test, comme nous l'avons fait dans l'exemple du Hello World, on peut utiliser une collection via la fonction *parallelize*.
Exemple en Scala :
val rdd1 : RDD[1] = sc.parallelize(1 until 100000)

Utilisation des RDD

A présent que nous savons comment charger un RDD, apprenons à l'utiliser.

Plusieurs transformations sont disponibles sur un RDD. En voici un extrait :

1. **Map** : change le type des objets contenus dans une collection
2. **Filter** : sélectionne un sous-ensemble d'une liste à partir d'un prédicat
3. **Union** : regroupe plusieurs RDD

Pour rappel, Spark ne réalise aucun traitement tant qu'on n'exécute pas l'action terminale. Il reste sur le **driver**. Spark construit en mémoire une structure de données reliant les transformations les unes et aux autres. Il prépare le **graphe acyclique dirigé**.

Les actions s'exécutent de manière distribuée sur des **workers**. Certaines actions ne produisent aucun résultat (ex : *println*), d'autres renvoient un objet ou une collection.
Les données issues du calcul sont retournées au drivers.
Par exemple, lors d'une action de réduction (qui consiste à réduire une liste, par exemple en sommant ses éléments), un objet Java ou Scala est retourné sur le driver.

Les speakers mettent en garde l'auditoire sur le fait que, lorsque l'action renvoie trop de données Java, un *OutOfMemoryException* a des chances d'être levé par la JVM. Attention donc aux volumes de données importants.

En fonction des opérations de transformation appliquées au RDD, Spark va optimiser les traitements. Le réseau est l'ennemi du distribué car très lent. Spark limite le **shuffling** et privilégie la **colocalisation**.

Workshop 1 : première action

Le premier workshop de ce Lab consiste à calculer la moyenne, le min, le max et le nombre de votes de l'utilisateur ayant l'identifiant 200.

Pour source de données, nous repartons du fichier ratings.txt dont voici les 3 premières lignes :

```
196 242    3  881250949
186    302    3  891717742
22 377    1  878887116
```

L'initiation du *JavaSparkContext* et la récupération du chemin vers le fichier ratings.txt s'effectuent de la même manière que dans la classe *FirstRDD*.
3 transformations sont ensuite enchaînées :

```
JavaRDD<Rating> ratings = sc.textFile(ratingsPath)
        .map(line -> line.split("\\t"))
        .map(row -> new Rating(
                Long.parseLong(row[0]),
                Long.parseLong(row[1]),
                Integer.parseInt(row[2]),
                LocalDateTime.ofInstant(Instant.ofEpochSecond(Long.pa

        ));
```

La 1ière transformation consiste à lire le fichier texte dans un tableau de String
La 2nde transformation consiste à séparer chaque ligne en tokens séparés par le caractère
La 3ième transformation permet de mapper les 4 tokens dans le POJO Rating.
Le résultat de ses transformations est l'obtention d'un RDD de Rating.
A noter que Spark ne permet malheureusement pas d'utiliser les Streams de Java 8 et que les méthodes map font ici parties de l'API Spark.

Le calcul de la moyenne des votes repart du RDD *ratings* et lui applique 2 transformations et une action :

```
double mean = ratings
        .filter(rating -> rating.user == 200)
        .mapToDouble(rating -> rating.rating)
        .mean();
```

Les transformations de filtre et de mapping utilisent les lambdas de Java 8.
L'action *mean()* est terminale. Elle déclenche la distribution du calcul sur les workers.

En interne, des acteurs Akka discutent ensemble. Les données sont échangées via de la sérialisation Java. Pour gagner en performance, il est possible d'utiliser Kryo pour sérialiser les données.
L'obtention de la note maximale attribuée par l'utilisateur n°200 ressemble au calcul de la moyenne. Sauf l'action finale diffère :

```
double max = ratings
        .filter(rating -> rating.user == 200)
        .mapToDouble(rating -> rating.rating)
        .max(Comparator.<Double>naturalOrder());
```

L'implémentation en Scala de ces quelques lignes aurait gagnée en concision. En effet, Scala aurait réussi à déterminer le comparateur par défaut des POJO et l'utilisation du *Comparator.<Double>naturalOrder()* aurait été superflue. Qui plus est, il aurait été inutile d'appeler la transformation *mapToDouble*. Son appel aurait été explicite.

Je ne rentrais pas ici dans le détail du calcul du min et du count. Le code source complet de la classe Workshop1 est disponible sur GitHub.

Workshop 2 : le cache

Dans le Workshop 1, le fichier est lu 4 fois. De même, le filtrage sur l'utilisateur n°200 est opéré 4 fois. Vous vous en doutez, 4 aller/retours entre le driver et le cluster a un coût. C'est pourquoi, lorsque des transformations sont communes à plusieurs opérations, Spark propose un mécanisme de cache. Et l'objectif du workshop n°2 est précisémment d'utiliser le cache.

Pour se faire, il faut indiquer à Spark que le RDD ne doit pas être déchargé suite à une action. Spark laisse alors les données sur le cluster. Libre au développeur de décharger le cache lorsqu'il n'en a plus besoin. Spark propose 5 stratégies de caching (exemple : StorageLevel.MEMORY_AND_DISK).

Dans la classe Workshop2, nous repartons du RDD ratings sur lequel nous appliquons une transformation de filtrage sur l'utilisateur 200. Ces transformations sont mises en cache :

```
JavaRDD<Rating> cachedRatingsForUser = ratings
        .filter(rating -> rating.user == 200)
        .cache();

double max = cachedRatingsForUser
        .mapToDouble(rating -> rating.user)
        .max(Comparator.<Double>naturalOrder());

double count = cachedRatingsForUser
```

```
            .count();

    cachedRatingsForUser.unpersist(false);
```

Lors de l'appel à l'action *max*, les logs montent que les 2 partitions rdd_4_0 et rdd_4_1 sont mises en cache :

```
15/04/12 14:03:43 INFO MemoryStore: ensureFreeSpace(4192) called with
15/04/12 14:03:43 INFO MemoryStore: Block rdd_4_1 stored as values in
15/04/12 14:03:43 INFO BlockManagerInfo: Added rdd_4_1 in memory on lo
15/04/12 14:03:43 INFO BlockManagerMaster: Updated info of block rdd_4
15/04/12 14:03:44 INFO MemoryStore: ensureFreeSpace(20704) called with
15/04/12 14:03:44 INFO MemoryStore: Block rdd_4_0 stored as values in
15/04/12 14:03:44 INFO BlockManagerInfo: Added rdd_4_0 in memory on lo
15/04/12 14:03:44 INFO BlockManagerMaster: Updated info of block rdd_4
```

L'appel à *unpersist* libère la mémoire :

```
15/04/12 14:03:44 INFO MapPartitionsRDD: Removing RDD 4 from persister
15/04/12 14:03:44 INFO BlockManager: Removing RDD 4
15/04/12 14:03:44 INFO BlockManager: Removing block rdd_4_0
15/04/12 14:03:44 INFO MemoryStore: Block rdd_4_0 of size 20704 droppe
15/04/12 14:03:44 INFO BlockManager: Removing block rdd_4_1
15/04/12 14:03:44 INFO MemoryStore: Block rdd_4_1 of size 4192 dropped
```

Workshop 3 : Spark SQL

Spark SQL permet d'exécuter de bonnes vielles requêtes SQL 92 sur un RDD structuré. Via JDBC, il est ainsi possible de brancher des outils de BI tels Business Object ou Crystal Report sur un RDD.

Depuis la version 1.3, la notion de SchemaRDD a été remplacée par celle de **DataFrame**.

Le 3ième et dernier Workshop réalisé au cours de ce Lab consiste à charger un fichier JSON contenant un tableau de produits dans un DataFrame puis à l'interroger en SQL.

La méthode *sql()* est une simple transformation que l'on peut chainer avec tout autre transformation, et en particulier d'autres requêtes SQL.

Avant de pouvoir d'utiliser le SQL il faut ajouter dans le *pom.xml* la dépendance vers le **module spark-sql** :

```
<dependency>
    <groupId>org.apache.spark</groupId>
    <artifactId>spark-sql_2.11</artifactId>
```

```
    <version>1.3.1</version>
</dependency>
```

La classe Workshop3 commence par créer un ***SQLContext*** à partir du *SparkContext*. Une liste de produits est ensuite chargée sous forme de *DataFrame* depuis un fichier JSON. Le premier produit de la liste est affiché :

```
SQLContext sqlContext = new SQLContext(sc);
String path = Workshop3.class.getResource("/products.json").getPath();
DataFrame products = sqlContext.load(path, "json");
System.out.println(products.first());
```

Le *DataFrame* est enregistré en tant que table temporaire portant le nom de *products*. Une requête SQL peut ensuite être exécutée sur cette table :

```
sqlContext.registerDataFrameAsTable(products, "products");
DataFrame frame = sqlContext.sql("SELECT count(*) FROM products where
System.out.println(frame.first());
```

Conclusion

En complément des 3 workshops réalisés en séance, les speakers auront eu l'occasion de nous initier aux aspects avancés de Spark, tels les pairs, les accumulateurs, le broadcast, Spark Streaming ou bien encore le repartionnement.

J'ai particulièrement apprécié la prestation d'Hayssam Saleh qui est très pédagogique. J'en ai eu de nouveau la confirmation en assistant à sa session *Machine Learning avec Spark, MMLIB et D3.js*. Je vous invite à regarder sa présentation sur Parleys lorsqu'elle sera disponible d'ici 4 à 8 semaines et, en attendant, à consulter son compte-rendu que je mettrais prochainement en ligne. Stay tuned.

Tweet

Ce contenu a été publié dans **Conférence** par **Antoine**, et marqué avec **BigData**, **Devoxx**, **Hadoop**, **Java**, **Scala**, **Spark**. Mettez-le en favori avec son **permalien [https://javaetmoi.com/2015/04/initiation-apache-spark-en-java-devoxx/]** .

4 THOUGHTS ON "INITIATION À APACHE SPARK AVEC JAVA"

Le **juin 19, 2015 à 11 h 01 min**,
Jeff MAURY
a dit :

Le lien vers le gitbook n'est plus valide

> Le **juin 19, 2015 à 21 h 27 min**,
> **Antoine**
> a dit :
>
> Merci Jeff pour le signalement. N'ayant pas retrouvé le gitbook sur Internet, je viens de publié la version PDF que j'avais archivée.

Le **janvier 12, 2016 à 10 h 46 min**,
Zak
a dit :

Bonjour,

Les liens pour :

Les jeu de données au format CSV et JSON nécessaires pour le Lab@
La configuration maven pour Java ou sbt pour Scala.

ne sont pas disponibles non plus … 😕

> Le **janvier 12, 2016 à 21 h 55 min**,
> **Antoine**
> a dit :
>
> Bonjour Zak,
> Merci de m'avoir signalé les liens morts. Par chance, j'avais conservé les 2 zips.
> Je les ai republiés.
> Bonne initiation,
> Antoine