**SPARK**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ce document expliquera ce qu’est SPARK, à quoi il sert et comment mettre cette API en place depuis le système d’exploitation windows via l’IDE Eclipse en JAVA.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Introduction**

Apache Spark est un framework de traitements Big Data open source pour effectuer des analyses sophistiquées et conçu pour la rapidité et la facilité d’utilisation.

Spark présente plusieurs avantages par rapport aux autres technologies big data et MapReduce comme Hadoop et Storm. D’abord, Spark propose un framework complet et unifié pour répondre aux besoins de traitements Big Data pour divers jeux de données, divers par leur nature (texte, graphe, etc.). Spark permet également à des applications sur une grappe de serveurs d’Hadoop d’être exécutées beaucoup plus rapidement.

Spark est à défaut de différents outils utilisé par Hadoop tel que MapReduce beaucoup plus facile à

mettre en place tout en permettant d’effectuer des traitement de données complexes.

Apache Spark contient une bibliothèque « core ». Sur ce framework Core de Spark s'appuient un certain nombre de bibliothèques plus haut niveau, en l'occurrence :

* Spark SQL, un module de manipulation des données sous une forme structurée (comme peut l'être SQL dans le monde des SGBDR)
* Spark Streaming, répondant aux problématiques de flux de données « temps réel »
* MLib, une bibliothèque de Machine Learning
* GraphX pour la manipulation de données orientées graphe.

Cependant, dans notre cas nous utiliserons uniquement le Core d'Apache Spark via **l’API Java Spark-Core**.

**Mise en place**

Nous utiliserons donc spark via son API Java Spark-Core. Étant plus facile à mettre en place nous obtiendrons des résultats beaucoup plus rapidement qu’en installant spark sur notre ordinateurs. Pour réaliser les prochaine étapes, nous aurons besoin d’installer les outils suivants :

* Apache Hadoop (pour se connecter à nos données sous la forme d’un csv)
* Un fichier de données (<https://ressources.data.sncf.com/explore/dataset/validations-sur-le-reseau-ferre-nombre-de-validations-par-jour-1er-semestre-2015/table/>)
* Maven (que nous pouvons facilement installer via utilisation de l’IDE eclipse)

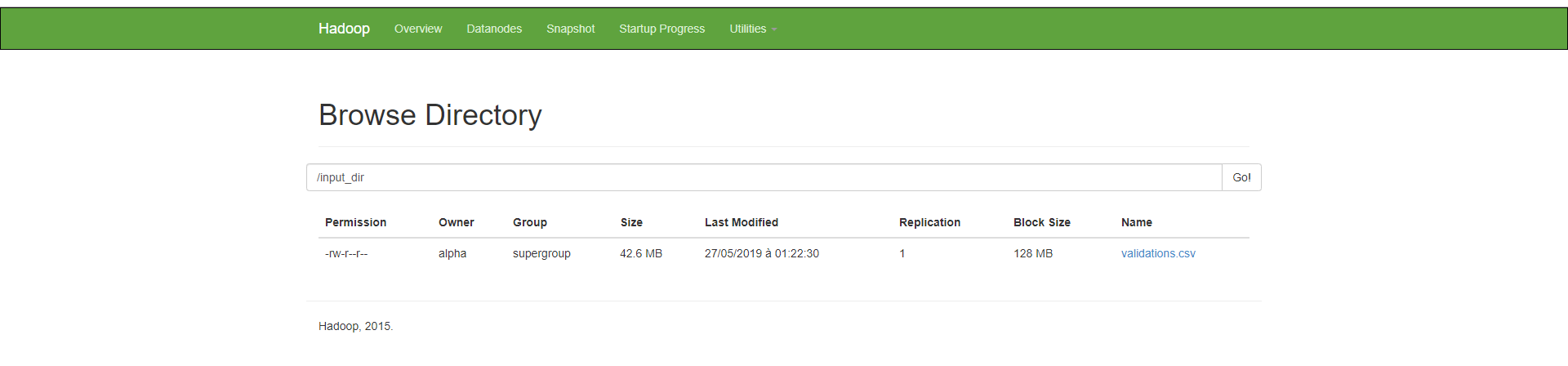
Une fois les outils installer. Nous devons tout d’abord rendre accessible notre fichiers de données .csv ou .txt depuis Hadoop. Pour cela, nous allons tout d’abord créer notre répertoire dans Hadoop via la commande :

*hadoop fs -mkdir /input\_dir*

Puis ajouter notre fichier dans ce répertoire :

hadoop fs -put cheminVersMonFichier /*input\_dir*

Nous pourrons alors voir que notre fichier est visible depuis l’URL **localhost:50070**.



**Annexe 1: Fichier .csv contenu dans hadoop**

Dans un second temps, nous devons ajouter notre API qui permettra de faire des manipulations de base sur les données. Pour cela, il nous suffit juste de rajouter notre dépendance dans notre pom.xml de Maven:

*<dependency>*

*<groupId>org.apache.spark</groupId>*

*<artifactId>spark-core\_2.11</artifactId>*

*<version>1.6.0</version>*

*</dependency>*

Une fois cela réalisations, nous pouvons commencer à réaliser notre configuration spark. Pour cela, nous allons ajouter le code suivant dans notre main :

*SparkConf conf = new SparkConf();*

*conf.setAppName("Nom que je veux");*

*conf.setMaster("local");*

*JavaSparkContext sparkContext = new JavaSparkContext(conf);*

*System.setProperty ("hadoop.home.dir", "D:\\hadoop\\hadoop-2.7.1");*

Maintenant que nous avons configuré notre spark, nous allons chercher notre fichier dans notre base spark. Pour cela, il nous suffit d’écrire le code suivant:

*JavaRDD<String> lines = sparkContext.textFile("hdfs://localhost:9000/input\_dir/validations.csv");*

|  |  |
| --- | --- |
|  | *Un RDD (pour Resilient Distributed Dataset) est une structure de données, immuable, itérable et complètement lazy. Cette structure représente un graphe acyclique ordonnée des différentes opérations à appliquer aux données chargées par Spark. Il s’agit en quelque sorte d’un plan d’exécution.* |

Pour tester son bon fonctionnement, nous pouvons effectuer certaines commandes tels que :

*long count = lines.count();*

*System.out.println("Nombre de lignes: "+ count);*

Maintenant que nous avons récupéré le contenu de notre fichier dans un RDD de chaînes de caractères, nous allons transformer chaque ligne en objet Validation.

*public class Validation implements Serializable*

*public LocalDate jour;*

*public String codeStifTrns;*

*public String codeStifRes;*

*public String codeStifArret;*

*public String libelleArret;*

*public String idRefaLda;*

*public String titre;*

*public Long validations;*

*public Validation(String validation) {*

*String split[] = validation.split(";");*

*this.jour = LocalDate.parse(split[0]);*

*this.codeStifTrns = split[1];*

*this.codeStifRes = split[2];*

*this.codeStifArret = split[3];*

*this.libelleArret = split[4];*

*this.idRefaLda = split[5];*

*this.titre = split[6];*

*this.validations = split.length == 8 ? split[7].equals("Moins de 5") ? 0 : Long.valueOf(split[7]) : 0;*

*}*

*// Getters et Setters ...*

Maintenant que nous avons créée notre classe, nous pouvons l’utiliser dans notr main :

*JavaRDD<Validation> validations = lines*

*// On filtre le Header du CSV*

*.filter(line -> !line.equals("JOUR;CODE\_STIF\_TRNS;CODE\_STIF\_RES;CODE\_STIF\_ARRET;LIBELLE\_ARRET;ID\_REFA\_LDA;CATEGORIE\_TITRE;NB\_VALD"))*

*// On transforme chaque ligne en Objet Validation*

*.map(maLigne -> new Validation(maLigne));*

*// On met le RDD en cache mémoire pour une prochaine utilisation*

*validations.cache();*

Nous avons donc créer un Java RDD que l’on a nommé validations. Celui-ci contiendra uniquement des objets de type validations. Chaque ligne de notre fichier .csv est une validation. Nous avons transformé chacune de c’est ligne en objet validation que l’on a ajouté à notre Java RDD validations.

Nous pouvons désormais réaliser diverse action sur Notre RDD tels que :

*long totalValidations = validations.map(v -> v.getValidations()).reduce((v1, v2) -> v1 + v2); //nombre total de voyageurs qui ont validé leur titre de transport dans le réseau ferré en Île-de-France sur le 1er semestre 2015*

***Résultat :***

***totalValidations: 788379173***

long totalValidationsJanvier = validations.filter(v1 -> v1.getJour().getMonth().equals(Month.JANUARY)).map(v -> v.getValidations()).reduce((v1, v2) -> v1 + v2); *//nombre total de voyageurs qui ont validé leur titre de transport dans le réseau ferré en Île-de-France sur le 1er semestre 2015*

***Résultat :***

***totalValidationsJanvier: 152279278***

On peut également afficher les 5 gares avec le plus grand nombre de validations :

*// On groupe par gare*

*validations.groupBy(v -> v.getLibelleArret())*

*// Pour chaque gare on calcule le total des validations*

*.mapToPair(tuple -> {*

*Map<String, Long> result = new HashMap<>();*

*result.put(tuple.\_1(), 0L);*

*tuple.\_2().forEach(validation -> {*

*result.put(tuple.\_1(), result.get(tuple.\_1()) + validation.getValidations());*

*});*

*return new Tuple2<String, Long>(tuple.\_1(), result.get(tuple.\_1()));*

*})*

*// On trie par ordre ascendant sur les validations et on garde le top 5*

*.top(5, MyTupleComparator.INSTANCE)*

*// On affiche*

*.stream().forEach(stringLongTuple2 -> System.out.println(stringLongTuple2.\_1() + " : " + stringLongTuple2.\_2() + " =====> " + ((double) stringLongTuple2.\_2() / totalValidations) \* 100 + " %"));*

*}*

*// Comparator sur les validations par gare*

*static class MyTupleComparator implements Comparator<Tuple2<String, Long>>, Serializable {*

*final static MyTupleComparator INSTANCE = new MyTupleComparator();*

*public int compare(Tuple2<String, Long> t1, Tuple2<String, Long> t2) {*

*return t1.\_2.compareTo(t2.\_2); // sort descending*

*}*

*}*

***Résultat :***

***LA DEFENSE-GRANDE ARCHE : 21930580 =====> 2.7817300039203343 %***

***GARE DE LYON : 12549570 =====> 1.5918190674983748 %***

***GARE DU NORD : 11561849 =====> 1.4665340480779037 %***

***MONTPARNASSE : 11542941 =====> 1.4641357097341787 %***

***SAINT-LAZARE : 11208687 =====> 1.421738090486061 %***