

TP Apache Spark

v1.1

Table of Contents

Intro

Connexion en ssh sur la VM depuis votre machine physique

Demarrage du cluster Spark

Configuration de l'environnement

Spark Shell

SparkUI

Spark shell

Exercices scala

Exercices RDDs

SparkSQL

Spark et Cassandra

Ressources

Intro

Pour le TP Spark vous allez utiliser une machine virtuelle qui contient un cluster Cassandra de 3 noeuds ainsi que Apache Spark. Le cluster Spark est composé d'un master et d'un noeud worker avec deux executeurs ([documentation deployment standalone](https://spark.apache.org/docs/2.0.2/spark-standalone.html) (<https://spark.apache.org/docs/2.0.2/spark-standalone.html>)) Vous allez vous connecter a cette machine via **ssh**.



Pour ce TP vous avez besoin d'une machine (idéalement linux) avec 4GB de RAM et 4GB d'espace disque disponible. **VirtualBox** et un client **SSH** doivent être déjà installés. Si vous etes sous Windows vous pouvez utiliser <http://www.putty.org/>

Connexion en ssh sur la VM depuis votre machine physique

Pour pouvoir accéder à l'interface SparkUI nous allons rediriger 4 ports :

- le port de SparkUI pour le Spark shell (4040) ⇒ sera dirige sur votre machine physique localhost:4040
- le port qui contient l'UI du master spark (8080) ⇒ sera dirige sur votre machine physique localhost:9080
- le port qui contient l'UI du premier worker (8081) ⇒ sera dirige sur votre machine physique localhost:8081
- le port qui contient l'UI du deuxiem worker (8082) ⇒ sera dirige sur votre machine physique localhost:8082

Pour **Windows** suivre le tuto suivant: [Configuration Tunnel SSH avec putty](https://intranet.cs.hku.hk/csintranet/contents/technical/howto/putty-portforward.jsp)
(<https://intranet.cs.hku.hk/csintranet/contents/technical/howto/putty-portforward.jsp>)

1. Connectez vous en ssh sur la VM depuis votre machine physique (un terminal linux/ putty sous windows). On profite pour rediriger les ports locaux de votre machine physique vers les ports de la VM :

```
[andrei@desktop ~]$ ssh -L 9080:127.0.0.1:8080 \
-L 8081:127.0.0.1:8081 \
-L 8082:127.0.0.1:8082 \
-L 4040:127.0.0.1:4040 \
bigdata@192.168.56.101
bigdata@192.168.56.101's password:
Last login: Sun Jan  4 14:53:32 2015 from pc12.home
[bigdata@bigdata ~]$
```



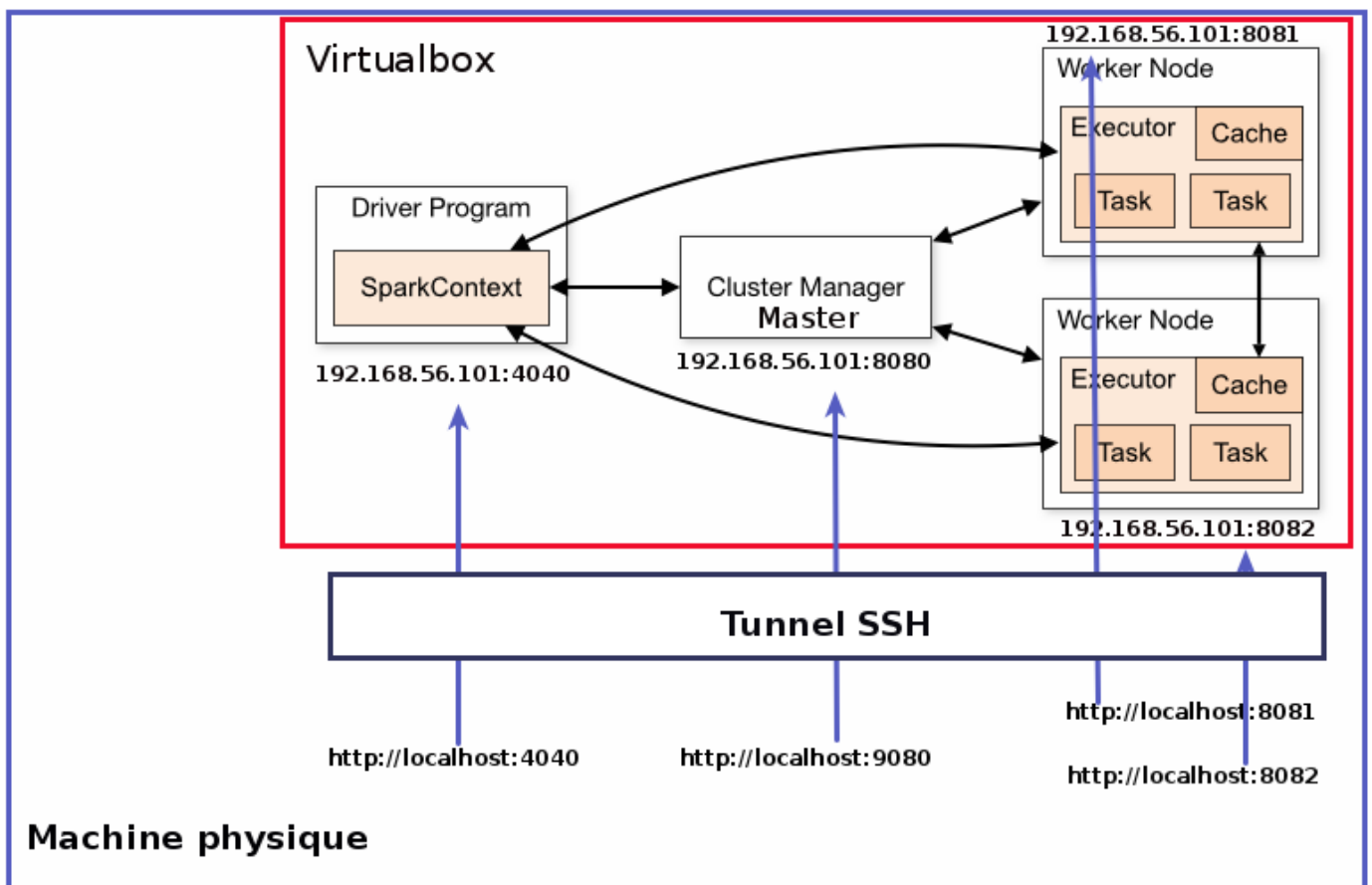
Identifiants de connexion:

- utilisateur: **bigdata**
- password: **bigdatafuret**

Si vous êtes sur Windows vous pouvez utiliser [putty](http://www.putty.org/) (<http://www.putty.org/>)

Demarrage du cluster Spark

Chaque application Spark utilise un programme *driver* qui sert à lancer des calculs distribués sur un cluster. Ce programme définit les structures de données distribuées dans le cluster ainsi que le DAG d'opérations sur ces structures de données. Un noeud coordinateur (appelé aussi master) lancera ces opérations sur deux noeuds Worker qui hébergeront un/plusieurs executeur(s).



Configuration de l'environnement

Rajouter dans le script de configuration de l'environnement spark (conf/spark-env.sh) la ligne *export SPARK_WORKER_INSTANCES=2* puis démarrer le master et les workers via le script *spark-2.0.2-bin-hadoop2.7/sbin/start-all.sh*

```
[bigdata@bigdata ~]${bigdata@bigdata ~}$ cat spark-2.0.2-bin-hadoop2.7/conf/spark-env.sh | grep
SPARK_WORKER_INSTANCES 1
# - SPARK_WORKER_INSTANCES, to set the number of worker processes per node
export SPARK_WORKER_INSTANCES=2
```

BASH

```
[bigdata@bigdata ~]$ spark-2.0.2-bin-hadoop2.7/sbin/start-all.sh 2
starting org.apache.spark.deploy.master.Master, logging to /home/bigdata/spark-2.0.2-bin-hadoop2.7/logs/spark-
bigdata-org.apache.spark.deploy.master.Master-1-bigdata.out
localhost: starting org.apache.spark.deploy.worker.Worker, logging to /home/bigdata/spark-2.0.2-bin-
hadoop2.7/logs/spark-bigdata-org.apache.spark.deploy.worker.Worker-1-bigdata.out
localhost: starting org.apache.spark.deploy.worker.Worker, logging to /home/bigdata/spark-2.0.2-bin-
hadoop2.7/logs/spark-bigdata-org.apache.spark.deploy.worker.Worker-2-bigdata.out
```

- 1 vérifier la configuration de l'environnement
- 2 on utilise le script start-all.sh pour démarrer le master et les workers

Spark Shell

- Une fois démarrées le master Spark et les Workers on se connecte via le shell Spark:

```
[bigdata@bigdata ~]$ spark-2.0.2-bin-hadoop2.7/bin/spark-shell \
    --master spark://bigdata:7077
```

BASH

```
[bigdata@bigdata ~]$ spark-2.0.2-bin-hadoop2.7/bin/spark-shell --master spark://bigdata:7077
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel).
16/12/04 08:18:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin
16/12/04 08:18:41 WARN SparkContext: Use an existing SparkContext, some configuration may not take effect.
Spark context Web UI available at http://192.168.56.101:4040
Spark context available as 'sc' (master = spark://bigdata:7077, app id = app-20161204081840-0000).
Spark session available as 'spark'.
Welcome to

  ____      __
 / ___ |    /  \
| |  \|    /    \
| |___|    /  ___| |
|  _  |   /  |___|
| | \ |  /  |___|
|_| \_| /  |___|

version 2.0.2

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_72)
Type in expressions to have them evaluated.
Type :help for more information.

scala> █
```

- Notez le port utilisé par SparkUI dans les messages de démarrage:

Dans ce qui suit on va utiliser le shell Spark comme driver.

Pour se connecter au cluster le *shell Spark* nous a créé automatiquement un objet `sc` de type *SparkContext* :

```
scala> sc 1
res0: org.apache.spark.SparkContext = org.apache.spark.SparkContext@3f10439f

scala> sc.getConf.toDebugString 2
res1: String =
hive.metastore.warehouse.dir=file:/home/bigdata/spark-warehouse
spark.app.id=app-20161204081840-0000
spark.app.name=Spark shell
spark.driver.host=192.168.56.101
spark.driver.port=33541
spark.executor.id=driver
spark.home=/home/bigdata/spark-2.0.2-bin-hadoop2.7
spark.jars=
spark.master=spark://bigdata:7077
spark.repl.class.outputDir=/tmp/spark-d120dc7d-9f40-4b5d-99ec-91cb46425e2d/repl-e18a9b14-f2d4-4f23-b916-bd61e3dd983b
spark.repl.class.uri=spark://192.168.56.101:33541/classes
spark.sql.catalogImplementation=hive
spark.submit.deployMode=client
```

- 1 afficher le type de l'objet
- 2 `sc.getConf.toDebugString` permet d'afficher la configuration du cluster

SparkUI

L'interface SparkUI permet de voir la configuration et l'état du cluster Spark.

Une fois la mise en place des 4 tunnels ssh (4040,8080,8081,8082) vous pouvez vous connecter à:

- l'interface SparkUI qui tourne sur le master : <http://localhost:9080> (permettra de connaître l'état du master, voir les jobs et logs)

Spark Master at spark://bigdata:7077

URL: spark://bigdata:7077
REST URL: spark://bigdata:6066 (*cluster mode*)
Alive Workers: 2
Cores in use: 2 Total, 2 Used
Memory in use: 2.0 GB Total, 2.0 GB Used
Applications: 1 [Running](#), 0 [Completed](#)
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Worker Id	Address	State	Cores	Memory
worker-20161204080815-192.168.56.101-53196	192.168.56.101:53196	ALIVE	1 (1 Used)	1024.0 MB (1024.0 MB Used)
worker-20161204081815-192.168.56.101-34877	192.168.56.101:34877	ALIVE	1 (1 Used)	1024.0 MB (1024.0 MB Used)

Running Applications

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20161204081840-0000 (kill)	Spark shell	2	1024.0 MB	2016/12/04 08:18:40	bigdata	RUNNING	32 min

Completed Applications

- l'interface du Worker 1 : <http://localhost:8081> (permettra de connaitre l'état du worker, voir les jobs et logs)



Spark Worker at 192.168.56.101:34877

ID: worker-20161204081815-192.168.56.101-34877

Master URL: spark://bigdata:7077

Cores: 1 (1 Used)

Memory: 1024.0 MB (1024.0 MB Used)

[Back to Master](#)

Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
1	1	RUNNING	1024.0 MB	ID: app-20161204081840-0000 Name: Spark shell User: bigdata	stdout stderr

- l'interface du Worker 2 : <http://localhost:8082> (permettra de connaitre l'état du worker, voir les jobs et logs)



Spark Worker at 192.168.56.101:53196

ID: worker-20161204080815-192.168.56.101-53196

Master URL: spark://bigdata:7077

Cores: 1 (1 Used)

Memory: 1024.0 MB (1024.0 MB Used)

[Back to Master](#)

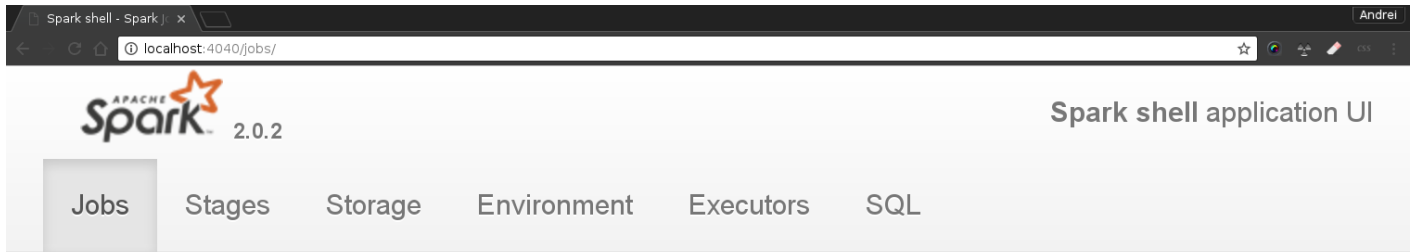
Running Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	1	RUNNING	1024.0 MB	ID: app-20161204081840-0000 Name: Spark shell User: bigdata	stdout stderr

Finished Executors (1)

ExecutorID	Cores	State	Memory	Job Details	Logs
0	1	KILLED	1024.0 MB	ID: app-20161204080839-0000 Name: Spark shell User: bigdata	stdout stderr

- l'interface du driver <http://localhost:4040> (permettra de suivre l'exécution de nos programmes Spark)



Spark Jobs (?)

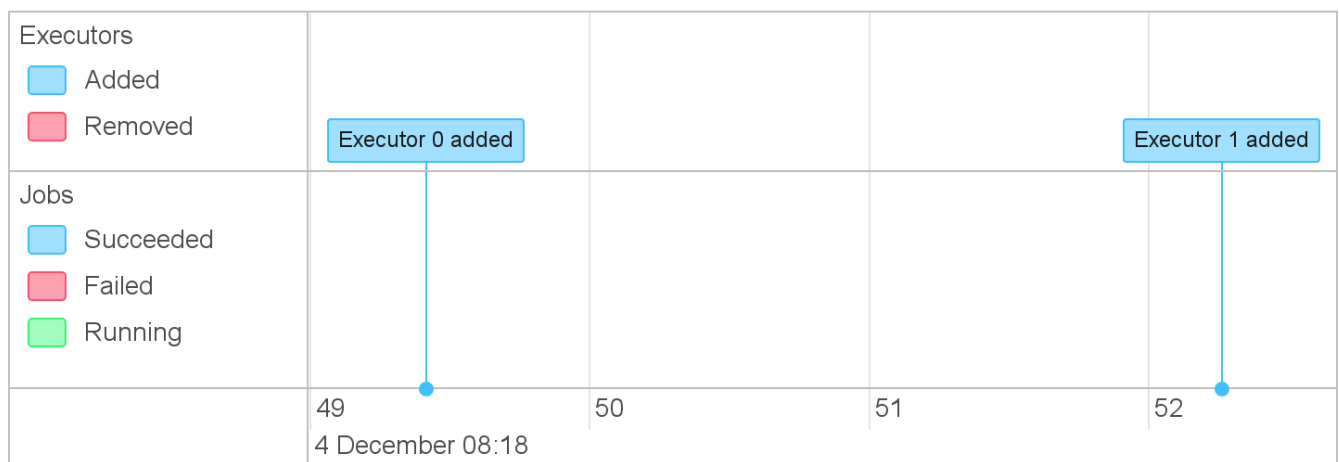
User: bigdata

Total Uptime: 32 min

Scheduling Mode: FIFO

▼ **Event Timeline**

☐ Enable zooming



Vérifier dans SparkUI que tous les noeuds worker sont connectés. Quelles sont les informations disponibles sur le cluster? Combien de noeuds sont dans votre cluster?

Spark shell

Exercices scala

Le *Spark shell* est un shell Scala dans lequel des classes supplémentaires ont été pré-importées et qui construit automatiquement un contexte Spark pour gérer l'échange avec un cluster. Dans *Spark shell* on peut donc écrire n'importe quelle instruction Scala.


```
scala> val myNumbers = List(1, 2, 5, 4, 7, 3) 1
myNumbers: List[Int] = List(1, 2, 5, 4, 7, 3)

scala> def cube(a: Int): Int = a * a * a 2
cube: (a: Int)Int

scala> myNumbers.map(x => cube(x)) 3
res2: List[Int] = List(1, 8, 125, 64, 343, 27)

scala> myNumbers.map{x => x * x * x} 4
res3: List[Int] = List(1, 8, 125, 64, 343, 27)

scala> def even(a: Int): Boolean = { a%2 == 0 } 5
even: (a: Int)Boolean

scala> myNumbers.map(x=>even(x)) 6
res4: List[Boolean] = List(false, true, false, true, false, false)

scala> myNumbers.map(even(_)) 7
res5: List[Boolean] = List(false, true, false, true, false, false)

scala> myNumbers.filter(even(_)) 8
res6: List[Int] = List(2, 4)

scala> myNumbers.foldLeft 9

def foldLeft[B](z: B)(f: (B, A) => B): B

scala> myNumbers.foldLeft(0)(_+_ ) 10
res10: Int = 22
```

- ¹ définir une variable immutable(*val*) de type List[Int]
- ² définir une fonction qui prend en entrée un entier et retourne le cube
- ³ utiliser la fonction map pour appliquer un traitement (ici la fonction cube est appliquée à chaque element de la liste)
- ⁴ utilisation d'une fonction anonyme pour faire le même traitement
- ⁵ definition d'une fonction Int→Boolean qui retourne si un nombre est pair
- ⁶ on transforme la liste des entiers dans une liste de booleans
- ⁷ la même chose qu'avant mais en utilisant une écriture plus compacte
- ⁸ filtrage des numéros paires de la liste
- ⁹ dans le shell on peut utiliser l'aide via la touche `TAB`. Ecrire myNumbers.foldLeft puis appuyer sur `TAB` permet d'afficher la signature de la fonction foldLeft de scala (<http://www.arolla.fr/blog/2011/10/listes-scala-methodes-foldleft-et-foldright/>). Elle prend deux listes d'arguments ⇒ un élément neutre de type B et une fonction f: (B, A) ⇒ B
- ¹⁰ utilisation de foldLeft pour calculer la somme des éléments de la liste

Ecrire une courte séquence Scala qui affiche les nombres inférieurs à 1000 à la fois divisibles par 2 et par 13 .

A. Version non parallélisée (Scala uniquement)

```
scala> val data = 1 to 1000 1
data: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121,
122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165,
166, 167, 168, 169, 170...)
scala> val filteredData= data.filter(n=> (n%2==0) && (n%13==0) ) 2
filteredData: scala.collection.immutable.IndexedSeq[Int] = Vector(26, 52, 78, 104, 130, 156, 182, 208, 234,
260, 286, 312, 338, 364, 390, 416, 442, 468, 494, 520, 546, 572, 598, 624, 650, 676, 702, 728, 754, 780, 806,
832, 858, 884, 910, 936, 962, 988)
```

- 1 générer une séquence (du type Scala `scala.collection.immutable.Range.Inclusive`) de tous les nombres de 1 à 1000. Cette séquence sera stockée dans la variable immutable `data`
- 2 garder les nombres qui sont divisible par 2 et 13



Dans l'exemple précédent on n'a pas utilisé le context Spark (l'objet `sc`). Les objets qu'on a manipulé sont les objets des collections standard Scala

(<http://www.scala-lang.org/api/current/#scala.collection.package>).

Exercices RDDs

1.1 Utiliser Spark pour distribuer les données et le filtrage de l'exercice précédent sur le cluster Spark.

SCALA

```
scala> val data = 1 to 1000 1
data: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44,
45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72,
73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121,
122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143,
144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165,
166, 167, 168, 169, 170...)
scala> val paralelizedData= sc.parallelize(data) 2
paralelizedData: org.apache.spark.rdd.RDD[Int] = ParallelCollectionRDD[0] at parallelize at <console>:14

scala> val filteredData=A_COMPLETER 3
filteredData: org.apache.spark.rdd.RDD[Int] = FilteredRDD[1] at filter at <console>:16

scala> filteredData.collect 4
...
```

- 1 génération des données (! dans la JVM du driver/shell Spark)
- 2 créer un **RDD** (`paralelizedData`) qui va contenir les données (distribuées sur les noeuds du cluster)
- 3 créer un **RDD** (`filteredData`) qui va contenir les données filtrées (distribuées sur les noeuds du cluster)
- 4 lancer l'action (`collect`) qui va déclencher le traitement parallèle et va retourner le résultat Suivre dans le UI l'exécution de votre script Spark <http://localhost:4040>.

1.2 Ecrire une courte séquence Scala/Spark qui affiche la somme des nombres paires de 1 à 1000 (utilisez la fonction `fold` disponible sur un RDD qui est similaire à `foldLeft` de scala (<http://www.arolla.fr/blog/2011/10/listes-scala-methodes-foldleft-et-foldright/>)).

1.3 Ecrire une courte séquence Scala/Spark qui affiche le produit des nombres paires de 1 à 1000.

1.4 Combiner les deux derniers programmes en un seul. Mettre en cache les données au niveau des noeuds et vérifier la répartition des ces données via le *Spark UI*


```
scala> val parDataCached = sc.parallelize(1 to 1000).filter(_%2==0).cache 1  
parDataCached: org.apache.spark.rdd.RDD[Int] = FilteredRDD[6] at filter at <console>:13
```

```
scala> parDataCached.fold(0)(_+_) 2  
res7: Int = 250500
```

```
scala> parDataCached.TODO 2
```

- 1 définition d'un RDD qui va stocker tous les nombres paires sur les noeuds
- 2 ré-utilisation du RDD pour calculer la somme et le produit de ces nombres (A FAIRE)

Vous pouvez vérifier sur les noeuds le contenu du RDD mis en cache via le SparkUI (<http://localhost:4040/storage/> puis cliquer sur le lien dans la colonne RDD Name)


 Jobs Stages **Storage** Environment Executors **Spark shell** application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in Tachyon	Size on Disk
1	Memory Deserialized 1x Replicated	2	100%	13.7 KB	0.0 B	0.0 B

Spark 1.2.0

Liste des RDD mise en cache.



[Jobs](#)
[Stages](#)
[Storage](#)
[Environment](#)
[Executors](#)

Spark shell application UI

RDD Storage Info for 1

Storage Level: Memory Deserialized 1x Replicated
Cached Partitions: 2
Total Partitions: 2
Memory Size: 13.7 KB
Disk Size: 0.0 B

Data Distribution on 3 Executors

Host	Memory Usage	Disk Usage
localhost:53089	6.9 KB (267.3 MB Remaining)	0.0 B
localhost:34974	6.9 KB (267.3 MB Remaining)	0.0 B
localhost:36256	0.0 B (267.3 MB Remaining)	0.0 B

2 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_1_0	Memory Deserialized 1x Replicated	6.9 KB	0.0 B	localhost:53089
rdd_1_1	Memory Deserialized 1x Replicated	6.9 KB	0.0 B	localhost:34974

Spark 1.2.0

Details sur la mise en cache.

1.5 WordCount: écrire le programme pour compter l'occurrence des mots du fichier `/home/bigdata/spark-2.0.2-bin-hadoop2.7/README.md`.

1.6 Ecrire le programme qui donne le mot le plus souvent utilisé du fichier (vous pouvez utiliser `sortByKey` après avoir inversé la liste des paires (mot,nb_occurrences)). Suivez via le SparkUI les stage d'executions de votre traitement (<http://localhost:4040/>)

SparkSQL

3.1 Utiliser SparkSQL pour trouver dans le fichier `/home/bigdata/spark-2.0.2-bin-hadoop2.7/examples/src/main/resources/people.txt` les noms des personnes qui ont moins de 19 ans

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc) 1
```

SCALA

```
case class Person(name: String, age: Int) 2
```

```
val people = sc.textFile("/home/bigdata/spark-2.0.2-bin-hadoop2.7/examples/src/main/resources/people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt)).toDF 3
```

```
people.createOrReplaceTempView("people") 4
```

```
val teenagers = sql("TODO") 5
```

```
teenagers.show 6
```


¹ création d'un contexte *SQLContext*

- 2 définition du modèle de données en utilisant une *case classe* Scala
- 3 création d'un dataframe Spark a partir d'un RDD
- 4 enregistrement du dataframe dans une table temporaire
- 5 requêtage sur la table → A COMPLETER !
- 6 le résultat d'une requête *SparkSQL* est un DataFrame, nous utilisons la methode show to afficher son contenu

Spark et Cassandra

Démarrer votre cluster *Cassandra* et vérifier qu'il est démarré:

BASH



```
[bigdata@bigdata ~]$ ccm status; ccm start; ccm status
Cluster: 'cassandra-2.1.16'
-----
node1: DOWN
node3: DOWN
node2: DOWN
Cluster: 'cassandra-2.1.16'
-----
node1: UP
node3: UP
node2: UP
```

Pour se connecter a Cassandra depuis le shell *Spark* il faut arrêter le shell (`Ctrl + C` or `Ctrl + D`) puis le redémarrer en ajoutant en paramètre l'adresse d'un noeud Cassandra et la dependance vers le connecteur spark-cassandra:

BASH

```
[bigdata@bigdata ~]$ spark-2.0.2-bin-hadoop2.7/bin/spark-shell --master spark://bigdata:7077 --conf
spark.cassandra.connection.host=127.0.0.1 --packages datastax:spark-cassandra-connector:2.0.0-M2-s_2.11
```

3.1 WordCount avec *Spark* et *Cassandra*

```
scala> import com.datastax.spark.connector._ 1
import com.datastax.spark.connector._

scala> val rdd = sc.cassandraTable("temperature", "temp1") 2
rdd: com.datastax.spark.connector.rdd.CassandraTableScanRDD[com.datastax.spark.connector.CassandraRow] =
CassandraTableScanRDD[0] at RDD at CassandraRDD.scala:18

scala> rdd.collect 3
res0: Array[com.datastax.spark.connector.CassandraRow] = Array(CassandraRow{ville: Paris, date: 2016-12-01
00:00:00+0100, temperature: 4}, CassandraRow{ville: Rennes, date: 2016-12-01 00:02:00+0100, temperature: 8},
CassandraRow{ville: Rennes, date: 2016-12-01 00:01:00+0100, temperature: 7}, CassandraRow{ville: Paris, date:
2016-12-01 00:02:00+0100, temperature: 6}, CassandraRow{ville: Rennes, date: 2016-12-01 00:00:00+0100,
temperature: 6}, CassandraRow{ville: Paris, date: 2016-12-01 00:01:00+0100, temperature: 5})

scala> import com.datastax.spark.connector.cql.CassandraConnector 4

scala> CassandraConnector(sc.getConf).withSessionDo { session =>
  session.execute(s"CREATE KEYSPACE IF NOT EXISTS demo WITH REPLICATION = {'class': 'SimpleStrategy',
'replication_factor': 1 }")
  session.execute(s"CREATE TABLE IF NOT EXISTS demo.wordcount (word TEXT PRIMARY KEY, count COUNTER)")
  session.execute(s"TRUNCATE demo.wordcount")
} 5

scala> val rdd = sc.textFile("/home/bigdata/spark-2.0.2-bin-hadoop2.7/README.md").flatMap(_.split("[^a-zA-Z0-9']+")).map(word => (word.toLowerCase, 1)).reduceByKey(_ + _).filter(x => x._1 != "") 6

scala> rdd.collect.foreach(println(_))

scala> rdd.saveToCassandra("demo", "wordcount") < 7

scala> sc.cassandraTable("demo", "wordcount").collect 8
```

- 1 importer toutes les classes du package `com.datastax.spark.connector`.
- 2 créer un RDD à partir de la table de températures.
- 3 collecter le RDD et afficher les températures
- 4 import de la classe `CassandraConnector` du package `cql`
- 5 créer un nouveau keyspace et une nouvelle table
- 6 créer un RDD avec les occurrences des mots du fichier `README.md`
- 7 sauver ce RDD dans la table Cassandra `wordcount`
- 8 afficher le contenu de la table `wordcount`

Ressources

[Spark Programming Guide](https://spark.apache.org/docs/latest/sql-programming-guide.html) (https://spark.apache.org/docs/latest/sql-programming-guide.html)

[SQL Dataframes Tutorial](http://spark.apache.org/docs/latest/sql-programming-guide.html) (http://spark.apache.org/docs/latest/sql-programming-guide.html)

[Scala API](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package) (http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.package)

[Python API](http://spark.apache.org/docs/latest/api/python/index.html) (http://spark.apache.org/docs/latest/api/python/index.html)

[Spark Cassandra Connector](https://github.com/datastax/spark-cassandra-connector) (https://github.com/datastax/spark-cassandra-connector)

[Scala Cheat-Sheet](http://homepage.cs.uiowa.edu/~tinelli/classes/022/Fall13/Notes/scala-quick-reference.pdf) (http://homepage.cs.uiowa.edu/~tinelli/classes/022/Fall13/Notes/scala-quick-reference.pdf)

Last updated 2018-12-03 22:35:39 +0100