TP Apache Spark

v1.0

Table of Contents

Vous pouvez télécharger ce document en format pdf: ***Version 1.0***

Intro

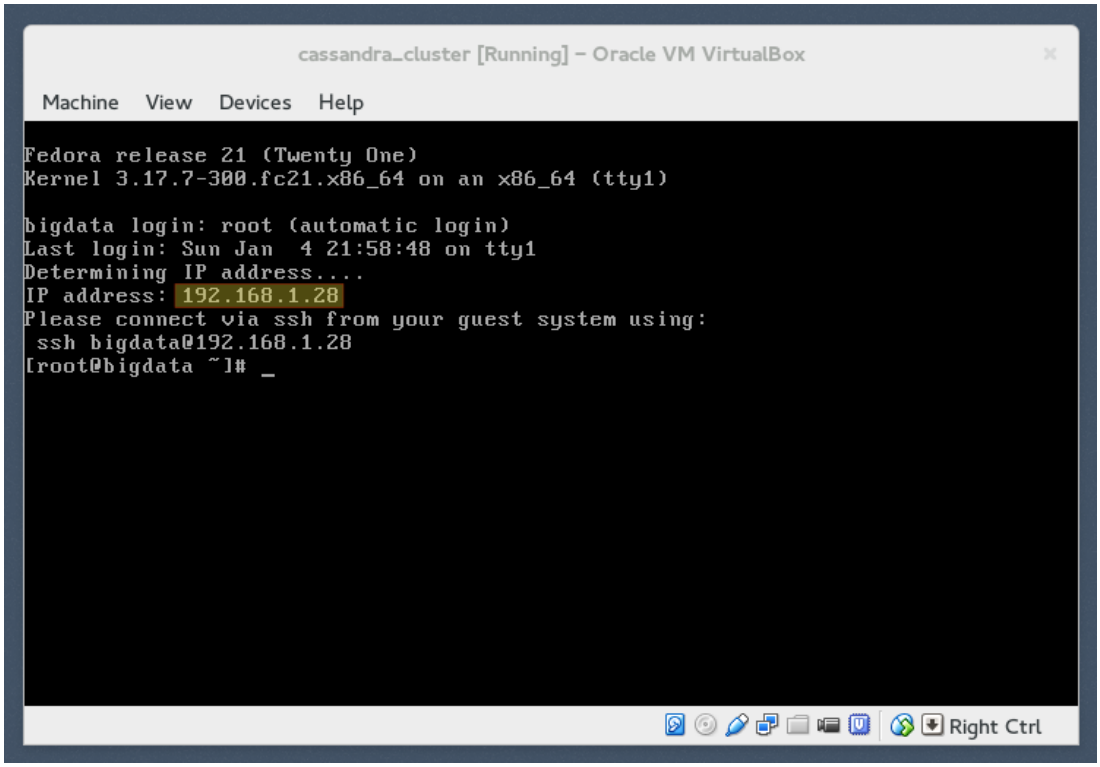
Pour le TP Spark vous allez utiliser une machine virtuelle qui contient un cluster Cassandra de 3 noeuds hébergés sur la même VM ainsi que Spark et Spark Notebooks. Le cluster Spark est composé d'un master et deux workers en mode standalone Vous allez vous connecter a cette machine via ***ssh***.

Pour ce TP vous avez besoin d'un poste (idéalement linux) avec 4GB de RAM et 4GB d'espace disque disponible. ***VirtualBox*** et un client ***SSH*** doivent être déjà installés.

Installation de la machine virtuelle

1. Telecharger ou recuperez-la depuis la cle USB
2. Importer cette VM dans votre VirtualBox
3. Lancer la VM
4. Noter l'adresse IP affichée lors du démarrage

Dans nos exemples l'adresse IP de la VM est ***192.168.1.28***. ***N'oubliez pas de remplacer partout dans les exemples cette adresse par l'adresse affichée dans la console VirtualBox***:



L'adresse IP que vous devez utiliser est affichée dans la fenêtre de la VM

Connexion en ssh sur la VM

1. Connectez vous en ssh sur la VM depuis un terminal linux :

```
[andrei@desktop ~]$ ssh bigdata@192.168.1.28
bigdata@192.168.1.28's password:
Last login: Sun Jan  4 14:53:32 2015 from pc12.home
[bigdata@bigdata ~]$
```

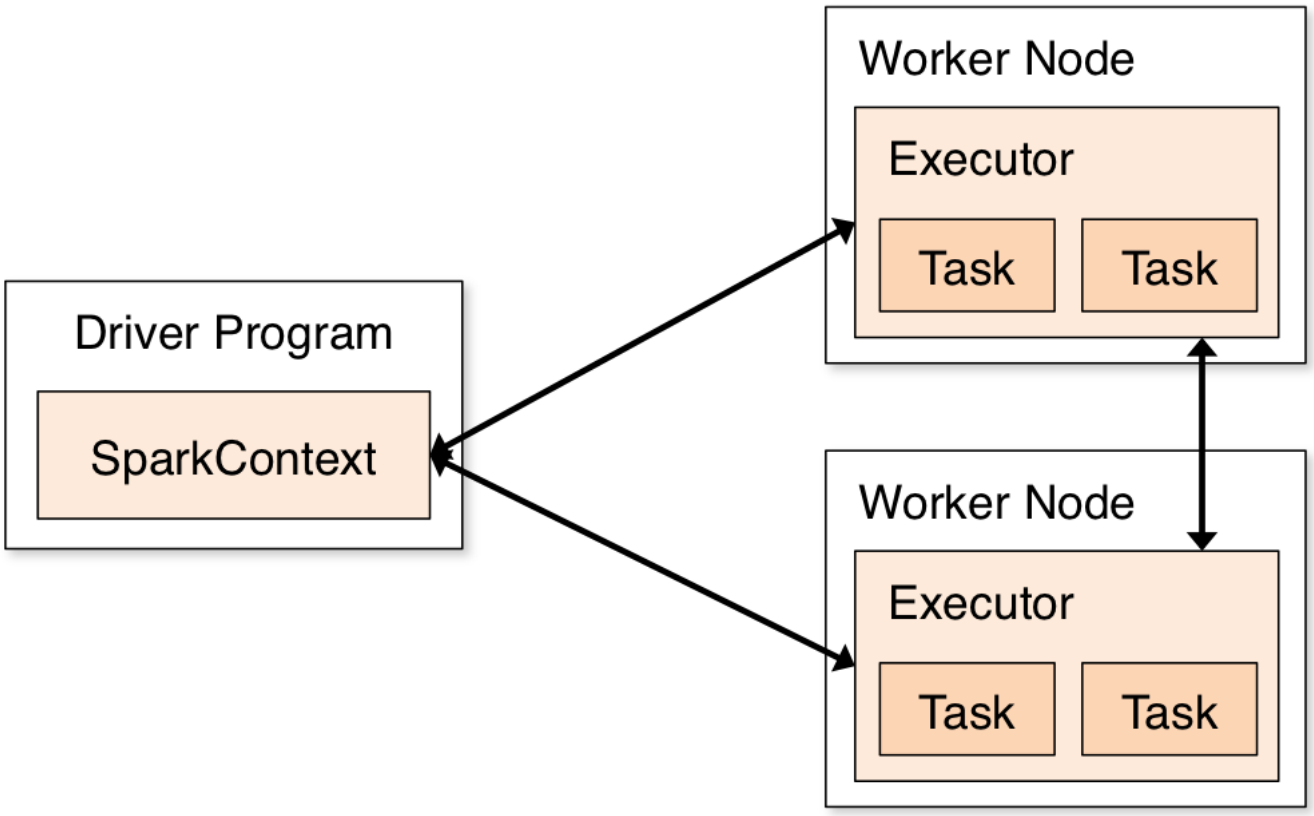
Identifiants de connexion:

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

Si vous êtes sur Windows vous pouvez utiliser putty

Demarrage du cluster Spark

Chaque application Spark est composée d'un programme *driver Spark* qui sert a lancer des calculs distribués sur un cluster. Ce programme définit les structures de données distribuées dans le cluster et lance des opérations sur ces structures de données. Sur chaque noeud *Spark* va lancer un executeur qui va pouvoir traiter plusieurs tâches.



Démarrer le master et les workers

```
[bigdata@bigdata ~]$ cat spark-1.2.0-bin-hadoop2.4/conf/spark-env.sh | grep
WORKER_INSTANCE ①
# - SPARK_WORKER_INSTANCES, to set the number of worker processes per node
export SPARK_WORKER_INSTANCES=2 //
[bigdata@bigdata ~]$ spark-1.2.0-bin-hadoop2.4/sbin/start-all.sh ②
starting org.apache.spark.deploy.master.Master, logging to /home/bigdata/spark-1.2.0-
bin-hadoop2.4/sbin/../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-1.2.0-bin-hadoop2.4/sbin/../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-1.2.0-bin-hadoop2.4/sbin/../logs/spark-bigdata-
org.apache.spark.deploy.worker.Worker-2-bigdata.out
```

① vérifier la configuration de l'environnement

② démarrer le master

③ démarrer deux noeuds worker

Spark Shell

- Une fois connectés en ssh sur la machine virtuelle lancer le shell Spark:

```
[bigdata@bigdata ~]$ spark-1.2.0-bin-hadoop2.4/bin/spark-shell --master
spark://127.0.0.1:7077
```

```
[bigdata@bigdata ~]$ spark-1.2.0-bin-hadoop2.4/bin/spark-shell
Spark assembly has been built with Hive, including Datanucleus jars on classpath
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
15/01/10 14:11:58 INFO SecurityManager: Changing view acls to: bigdata
15/01/10 14:11:58 INFO SecurityManager: Changing modify acls to: bigdata
15/01/10 14:11:58 INFO SecurityManager: SecurityManager: authentication disabled
; ui acls disabled; users with view permissions: Set(bigdata); users with modify
 permissions: Set(bigdata)
15/01/10 14:11:58 INFO HttpServer: Starting HTTP Server
15/01/10 14:11:58 INFO Utils: Successfully started service 'HTTP class server' o
n port 33029.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.2.0
      /_/

Using Scala version 2.10.4 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_72)
Type in expressions to have them evaluated.
Type :help for more information.
15/01/10 14:12:03 INFO SecurityManager: Changing view acls to: bigdata
15/01/10 14:12:03 INFO SecurityManager: Changing modify acls to: bigdata
```

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

```
ocal-20150110141204-138e
15/01/10 14:12:04 INFO MemoryStore: MemoryStore started with capacity 267.3 MB
15/01/10 14:12:04 WARN NativeCodeLoader: Unable to load native-hadoop library fo
r your platform... using builtin-java classes where applicable
15/01/10 14:12:05 INFO HttpFileServer: HTTP File server directory is /tmp/spark-
ed77bd95-b012-4dae-938d-af7a68978f3d
15/01/10 14:12:05 INFO HttpServer: Starting HTTP Server
15/01/10 14:12:05 INFO Utils: Successfully started service 'HTTP file server' on
 port 34840.
15/01/10 14:12:05 INFO Utils: Successfully started service 'SparkUI' on port 404
0.
15/01/10 14:12:05 INFO SparkUI: Started SparkUI at http://pc34.home:4040
15/01/10 14:12:05 INFO Executor: Using REPL class URI: http://192.168.1.28:33029
15/01/10 14:12:05 INFO AkkaUtils: Connecting to HeartbeatReceiver: akka.tcp://sp
arkDriver@pc34.home:60020/user/HeartbeatReceiver
15/01/10 14:12:05 INFO NettyBlockTransferService: Server created on 60293
15/01/10 14:12:05 INFO BlockManagerMaster: Trying to register BlockManager
15/01/10 14:12:05 INFO BlockManagerMasterActor: Registering block manager localh
ost:60293 with 267.3 MB RAM, BlockManagerId(<driver>, localhost, 60293)
15/01/10 14:12:05 INFO BlockManagerMaster: Registered BlockManager
15/01/10 14:12:06 INFO SparkILoop: Created spark context..
Spark context available as sc.

scala>
```

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

Pour se connecter au cluster le *shell Spark* nous a crée automatiquement un objet "SparkContext" qui est notre connexion au cluster qu'on peut utiliser directement:

```
15/01/10 14:12:05 INFO BlockManagerMaster: Registered BlockManager
15/01/10 14:12:06 INFO SparkILoop: Created spark context..
Spark context available as sc.
scala> sc ①
res1: org.apache.spark.SparkContext = org.apache.spark.SparkContext@3b4d1da7
scala> sc.getConf.toDebugString ①
res2: String =
spark.app.id=app-20150111011726-0000
spark.app.name=Spark shell
spark.driver.host=localhost
spark.driver.port=57327
spark.executor.id=driver
spark.fileserver.uri=http://127.0.0.1:51125
spark.jars=
spark.master=spark://127.0.0.1:7077
spark.repl.class.uri=http://127.0.0.1:39996
spark.tachyonStore.folderName=spark-6553813e-3e57-4deb-9bc4-43b295f62221
```

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.

Pour pouvoir accéder à l'interface SparkUI il faut faire 4 tunnels SSH (dans 4 terminaux differents) pour exposer:

- le port de SparkUI pour le Spark shell (4040)
- le port qui contient l'UI du master spark (8080)
- le port qui contient l'UI du premier worker (8081)
- le port qui contient l'UI du deuxiem worker (8082)

```
[andrei@desktop ~]$ ssh -nNT -L 4040:127.0.0.1:4040
bigdata@192.168.1.28

bigdata@192.168.1.28's password: ①
```

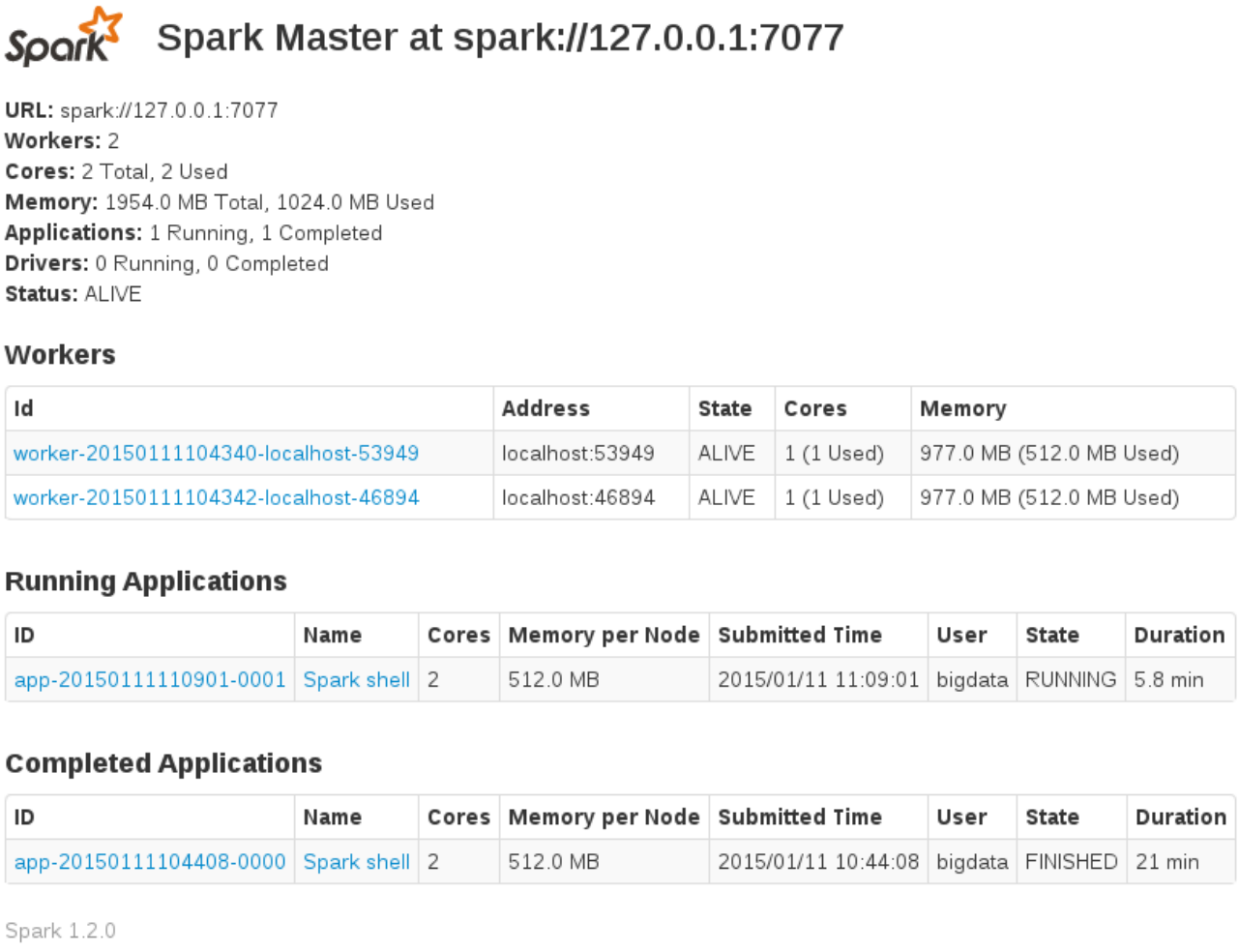
① et saisir le mot de pase ssh (***bigdatafuret***)

Puis dans d'autres terminaux refaire la redirection pour les ports 8080, 8081 et 8082.

Pour ***Windows*** suivre le tuto suivant: Configuration Tunnel SSH avec putty

Une fois la mise en place des 4 tunnels ssh (4040,8080,8081,8082) vous pouvez vous connecter à SparkUI qui tourne sur le master : http://localhost:8080

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



Spark Master at spark://127.0.0.1:7077

URL: spark://127.0.0.1:7077
Workers: 2
Cores: 2 Total, 2 Used
Memory: 1954.0 MB Total, 1024.0 MB Used
Applications: 1 Running, 1 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers

Id	Address	State	Cores	Memory
worker-20150111104340-localhost-53949	localhost:53949	ALIVE	1 (1 Used)	977.0 MB (512.0 MB Used)
worker-20150111104342-localhost-46894	localhost:46894	ALIVE	1 (1 Used)	977.0 MB (512.0 MB Used)

Running Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150111110901-0001	Spark shell	2	512.0 MB	2015/01/11 11:09:01	bigdata	RUNNING	5.8 min

Completed Applications

ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20150111104408-0000	Spark shell	2	512.0 MB	2015/01/11 10:44:08	bigdata	FINISHED	21 min

Spark 1.2.0

Combien de noeuds sont dans votre cluster?

Lancer des traitements via *Spark shell*

Lancer quelques commandes Scala dans le Spark shell

Le *Spark shell* est un fork du 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 function foldLeft de scala. Elle prend deux listes d'arguments ⇒ un élèment neutre de type B et une fonction f: (B, A) ⇒ B

⑩ utilisation de foldLeft pour calculer la somme des élèments de la liste

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

Version non parallelisée (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 génerer 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 précedent on n'a pas utilisé le context Spark (l'objet sc). Les objets qu'on a manipulé sont les objets des collections standard Scala.

Modifier votre code pour utiliser Spark pour distribuer vos données et le filtrage sur le cluster.

```
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 géneration des données (! dans la JVM du driver/shell Spark)

- Création d'un graph de traitement:

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)

- Execution du graph de traitement:

④ lancer l'action (*collect*) qui va déclencher le traitement parallèle et va retourner le résultat

Par default le niveau de log dans la console ***Spark*** est à INFO. Lors du lancement de l'action (*collect*) toutes les informations sur l'exécution de l'action sur le cluster sont affichées dans la console.

```
scala> filteredData.collect ①
15/01/10 16:49:30 INFO SparkContext: Starting job: collect at <console>:19
15/01/10 16:49:30 INFO DAGScheduler: Got job 0 (collect at <console>:19) with 1 output
partitions (allowLocal=false)
15/01/10 16:49:30 INFO DAGScheduler: Final stage: Stage 0(collect at <console>:19)
15/01/10 16:49:30 INFO DAGScheduler: Parents of final stage: List()
15/01/10 16:49:30 INFO DAGScheduler: Missing parents: List()
15/01/10 16:49:30 INFO DAGScheduler: Submitting Stage 0 (FilteredRDD[1] at filter at
<console>:16), which has no missing parents
15/01/10 16:49:30 INFO MemoryStore: ensureFreeSpace(1680) called with curMem=0,
maxMem=280248975
15/01/10 16:49:30 INFO MemoryStore: Block broadcast_0 stored as values in memory
(estimated size 1680.0 B, free 267.3 MB)
15/01/10 16:49:30 INFO MemoryStore: ensureFreeSpace(1226) called with curMem=1680,
maxMem=280248975
15/01/10 16:49:30 INFO MemoryStore: Block broadcast_0_piece0 stored as bytes in memory
(estimated size 1226.0 B, free 267.3 MB)
15/01/10 16:49:30 INFO BlockManagerInfo: Added broadcast_0_piece0 in memory on
localhost:48812 (size: 1226.0 B, free: 267.3 MB)
15/01/10 16:49:30 INFO BlockManagerMaster: Updated info of block broadcast_0_piece0
15/01/10 16:49:30 INFO SparkContext: Created broadcast 0 from broadcast at
DAGScheduler.scala:838
15/01/10 16:49:30 INFO DAGScheduler: Submitting 1 missing tasks from Stage 0
(FilteredRDD[1] at filter at <console>:16)
15/01/10 16:49:30 INFO TaskSchedulerImpl: Adding task set 0.0 with 1 tasks
15/01/10 16:49:30 INFO TaskSetManager: Starting task 0.0 in stage 0.0 (TID 0,
localhost, PROCESS_LOCAL, 1260 bytes)
15/01/10 16:49:30 INFO Executor: Running task 0.0 in stage 0.0 (TID 0)
15/01/10 16:49:30 INFO Executor: Finished task 0.0 in stage 0.0 (TID 0). 746 bytes
result sent to driver
15/01/10 16:49:30 INFO DAGScheduler: Stage 0 (collect at <console>:19) finished in
0.039 s
15/01/10 16:49:30 INFO DAGScheduler: Job 0 finished: collect at <console>:19, took
0.203414 s
15/01/10 16:49:30 INFO TaskSetManager: Finished task 0.0 in stage 0.0 (TID 0) in 36 ms
on localhost (1/1)
res0: Array[Int] = Array(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) ②
scala> 15/01/10 16:49:30 INFO TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have
all completed, from pool
```

① Le déclenchement de l'action *collect* lance la distribution des données et l'exécution du calcul sur les noeuds

② le résultat du traitement est rappatrié au niveau du driver et affiché dans la console

Si vous voulez, vous pouvez changer le niveau du log par défaut dans le shell Spark:

```
scala> import org.apache.log4j.{Level, Logger} ①
import org.apache.log4j.{Level, Logger}

scala> val level = Level.WARN ②
level: org.apache.log4j.Level = WARN

scala> Logger.getLogger("org").setLevel(level) ③

scala> Logger.getLogger("akka").setLevel(level) ③

scala>
```

① Importer les classes Level et Logger

② Créer une valeur pour le niveau WARN

③ Configurer le nouveau niveau de logs pour le spark Shell

Si vous voulez que cette modification soit persistante vous pouvez copier le fichier *log4j.properties.example* qu'on vous fournit

```
[bigdata@bigdata ~]$ cp spark-1.2.0-bin-
hadoop2.4/conf/log4j.properties.example  spark-1.2.0-bin-
hadoop2.4/conf/log4j.properties
```

Ecrire une courte séquence Scala/Spark qui affiche la somme des nombres paires de 1 a 1000 (utilisez la fonction *fold* disponible sur un RDD qui est similaire a foldLeft de scala).

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

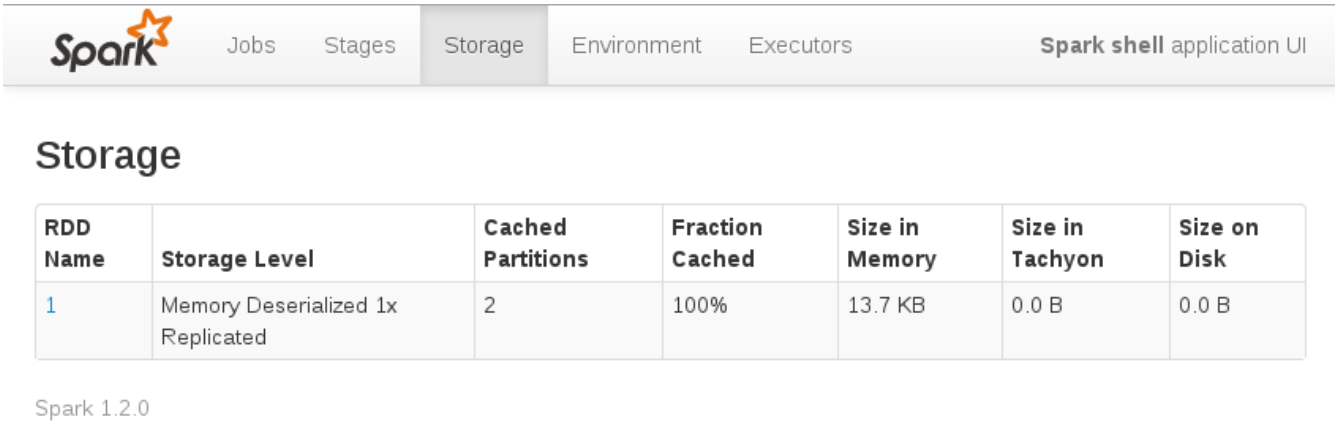
Combiner les deux derniers programmes en un seul. Mettre en cache les données au niveau des noeuds et vérifier la répartition des ces données via le *Spark UI*

```
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
```

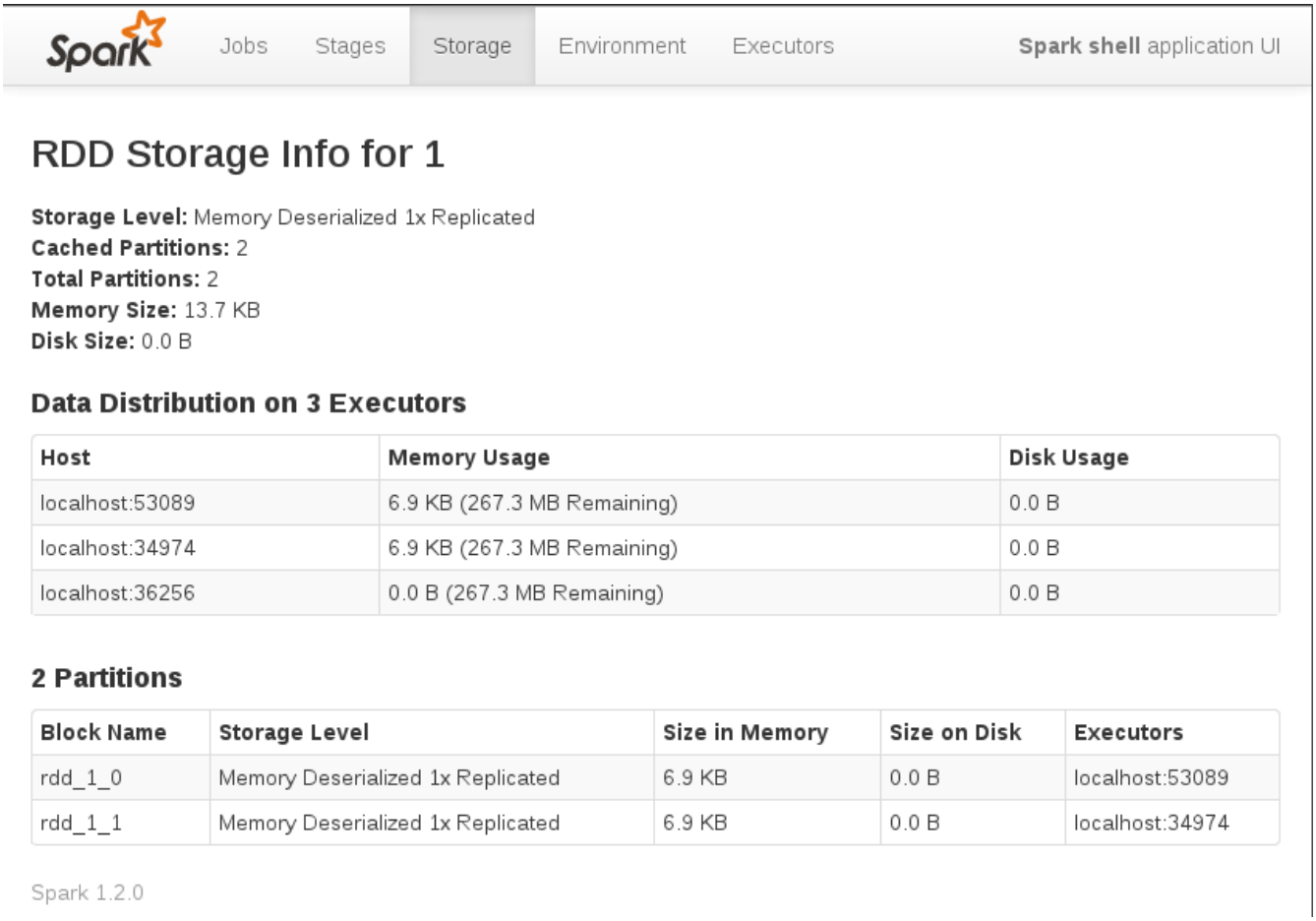
① définition d'un RDD qui va stocquer tous les nombres paires sur les noeuds

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)



Liste des RDD mise en cache.



Details sur la mise en cache.

WordCount: écrire le programme pour compter l'occurence des mots du fichier */home/bigdata/spark-1.2.0-bin-hadoop2.4/README.md*.

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_occurences))

SparkSQL

Utiliser SparkSQL pour trouver dans le fichier */home/bigdata/spark-1.2.0-bin-hadoop2.4/examples/src/main/resources/people.txt* les noms des personnes qui ont le

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc) ①
import sqlContext._ ②


case class Person(name: String, age: Int) ③


val people = sc.textFile("/home/bigdata/spark-1.2.0-bin-
hadoop2.4/examples/src/main/resources/people.txt").map(_.split(",")).map(p =>
Person(p(0), p(1).trim.toInt)) ④

people.registerAsTable("people") ④


val teenagers = sql("TODO") ⑤


teenagers.map(t => "Name: " + t(0)).collect ⑥
```

① création d'un contexte *SQLContext*

② importation des méthodes utilitaires (ex: *sql()*)

③ définition du modèle de données en utilisant une *case classe* Scala

④ création d'un RDD et l'enregistrement dans une table

⑤ requêtage sur la table → A COMPLETER !

⑥ le résultat d'une requête *SparkSQL* est un RDD

Cassandra and Spark [Optionel]

Le connecteur *Spark* pour *Cassandra* a ete copié dans le répertoire */home/bigdata/spark-cassandra-connector* de la VM.

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

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

Pour l'utiliser dans le shell *Spark* il faut arrêter le shell (`Ctrl`+`C` or `Ctrl`+`D`) puis le redémarrer en ajoutant comme paramètre le jar du connector:

```
[bigdata@bigdata ~]$ spark-1.2.0-bin-hadoop2.4/bin/spark-shell --master
spark://127.0.0.1:7077 --jars /home/bigdata/spark-cassandra-connector/spark-cassandra-
connector-assembly-1.1.2-SNAPSHOT.jar
```

WordCount avec *Spark* et *Cassandra*

```
scala> sc.stop ①

scala> import com.datastax.spark.connector._
scala> import org.apache.spark.SparkContext
scala> import org.apache.spark.SparkContext._
scala> import org.apache.spark.SparkConf
scala> import com.datastax.spark.connector.cql.CassandraConnector

scala> val conf = new SparkConf(true).set("spark.cassandra.connection.host",
"127.0.0.1") ②

scala> val sc = new SparkContext("spark://127.0.0.1:7077", "test", conf) ③

scala> :paste ④

scala> CassandraConnector(conf).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> sc.textFile("/home/bigdata/spark-1.2.0-bin-
hadoop2.4/README.md").flatMap(_.split("\\s+")).map(word => (word.toLowerCase,
1)).reduceByKey(_ + _).saveToCassandra("demo", "wordcount")⑥

scala> sc.cassandraTable("demo", "wordcount").collect.foreach(println) ⑦
```

① arrêter le contexte Spark démarré par le shell Spark

② création d'une configuration pour la connexion à *Cassandra*.

③ créer un nouveau contexte Spark avec la configuration du cluster *Cassandra*

④ activer le mode *paste* pour pouvoir copier une commande sur plusieurs lignes

⑤ créer le keyspace *demo* et la table *wordcount* (copier puis finir par `Ctrl`+`D`)

⑥ compter les mots et sauvegardez le résultat dans la table *wordcount*

⑦ afficher le contenu de la table *wordcount*

Documentation:

Spark Programming Guide

SQL Dataframes Tutorial

Scala API

Python API