

# INTRODUCCIÓN A APACHE SPARK CON PYTHON

Introducción a Apache Spark

Jorilles.com  
info@jorilles.com

## Índice de contenido

1.Descripción.....	3
2.Cual es su ventaja competitiva.....	3
3.Instalación.....	3
4.Conceptos básicos.....	4
5. Rendimiento.....	7
6. Caso de uso: Cluster en Spark.....	8

## 1. Descripción

Apache Spark es un framework de computación distribuida open source, con el motor de procesamiento con mayor velocidad gracias a su tecnología de procesamiento en memoria de grandes volúmenes de datos.

## 2. Cual es su ventaja competitiva

Spark ofrece múltiples ventajas respecto a MapReduce-Hadoop

- Procesamiento en memoria. Con el fin de mejorar el rendimiento entre operaciones, se permite la persistencia o el almacenamiento en caché de un RDD entre operaciones.
- Las aplicaciones son ejecutadas en Cluster por los nodos y están gestionadas por un maestro
- Escalable y tolerante a fallos

## 3. Instalación

Spark esta escrito en Scala y se ejecuta bajo una maquina virtual de Java. Por ello para ejecutar Apache Spark, es necesario tener instalado Java 6 o superior.

Como en esta introducción queremos trabajar con Python, necesitaremos un intérprete, si no lo tenemos podemos instalar Python 2.7.

Descargamos Apache Spark:

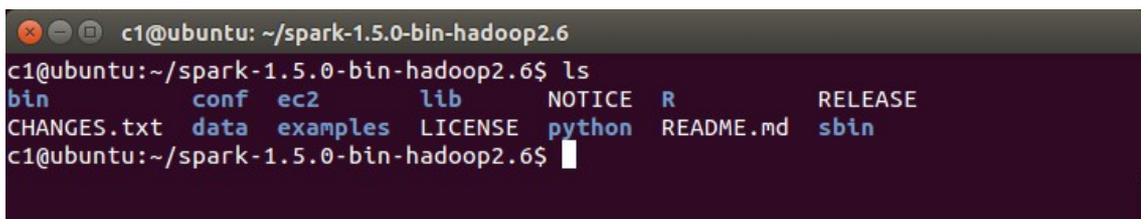
```
wget http://archive.apache.org/dist/spark/spark-1.5.0/spark-1.5.0-bin-hadoop2.6.tgz
```

```
tar -xf spark-1.5.1-bin-hadoop2.6.tgz
```

Como muchas de las herramientas del ecosistema Hadoop, Spark dispone de su propia Shell para poder interactuar, ejecutar sentencias y procesar datos.

Hay dos Shell disponibles en Spark, en la carpeta de *bin*, podemos encontrar ambas

- **Scala:** `spark-shell` --> Para poder utilizar la Shell d'Spark, es necesario instalar Scala
- **Python** → ejecuta directamente `./pyspark`. En esta introducción utilizaremos PySpark



```
c1@ubuntu: ~/spark-1.5.0-bin-hadoop2.6
c1@ubuntu:~/spark-1.5.0-bin-hadoop2.6$ ls
bin      conf    ec2      lib      NOTICE  R        RELEASE
CHANGES.txt  data  examples LICENSE  python  README.md  sbin
c1@ubuntu:~/spark-1.5.0-bin-hadoop2.6$
```



Existen **dos** tipos de operaciones con los RDDs pueden ser **transformados**, crea un nuevo RDD (Map, Filter, union..) o **consultados** (acción), nos da un resultado (Reduce, count..)

Las transformaciones las realiza de forma perezosa (lazy) esto quiere decir que las diferentes transformaciones solicitadas se irán guardando, hasta que se ejecute una acción.

### Ecosistema SPARK: Dispone de herramientas de alto nivel

- Spark- Subimt: Un script para ejecutar aplicaciones
- Spark Streaming: Mezclar Batch processing con real time
- Spark -SQL: Un modulo para trabajar con bases de datos estructurados.
- Spark – Mlib: Machine Learning. Aprendizaje automático.

### Funcionamiento de Spark

A alto nivel, las aplicaciones Spark consisten en un programa principal que lanza varias operaciones paralelas en un Cluster . Este programa contiene la función principal de la aplicación y define los conjuntos de datos distribuida en el cluster , y aplica las operaciones con ellos .

El programa principal accede a Spark, a través de un objeto llamado SparkContext, que representa una conexión a un cluster, y los nodos esclavos con coordinados por este objeto.

### Consola interactiva PySpark

Cuando trabajamos directamente con la shell, es en si misma el programa principal , y se escriben de una en una las operaciones que se quieran aplicar.

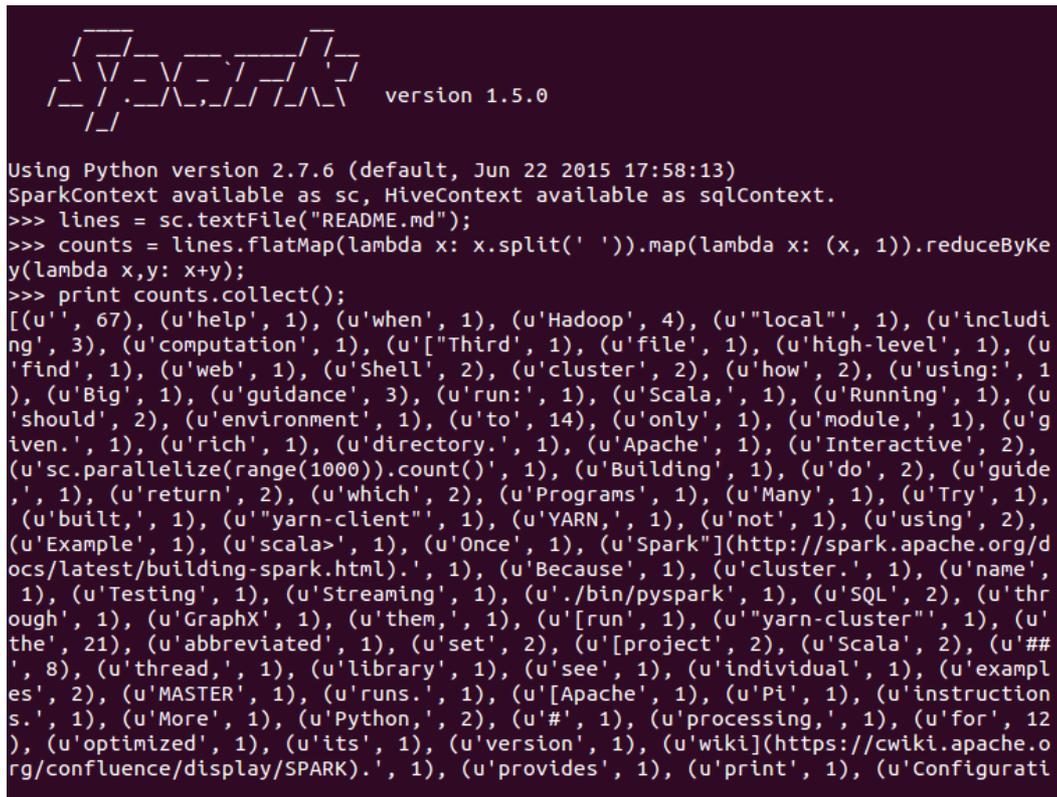
El SparkContext se crea automáticamente con la variable llamada sc . Ejecutamos Pyspark:

```
$ ./bin/pyspark
```

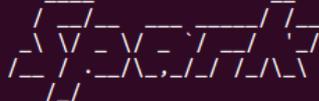
El primer paso es crear un objeto de SparkContext, con la variable sc. Una vez declarado, creamos el RDD, con `sc.textFile` declaramos las lineas de un fichero de texto.

Para ejecutar count() si estuviéramos en un cluster, el maestro lo enviaría a los esclavos para ejecutarlo, pero como estamos en local, nuestra máquina realiza todo el trabajo.

```
>>> lines = sc.textFile("README.md");
>>> counts = lines.flatMap(lambda x: x.split(' ')).map(lambda x: (x,
1)).reduceByKey(lambda x,y: x+y);
>>> print counts.collect();
```



```

 version 1.5.0

Using Python version 2.7.6 (default, Jun 22 2015 17:58:13)
SparkContext available as sc, HiveContext available as sqlContext.
>>> lines = sc.textFile("README.md");
>>> counts = lines.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey
(lambda x,y: x+y);
>>> print counts.collect();
[(u'', 67), (u'help', 1), (u'when', 1), (u'Hadoop', 4), (u'"local"', 1), (u'includi
ng', 3), (u'computation', 1), (u['"Third', 1), (u'file', 1), (u'high-level', 1), (u
'find', 1), (u'web', 1), (u'Shell', 2), (u'cluster', 2), (u'how', 2), (u'using:', 1
), (u'Big', 1), (u'guidance', 3), (u'run:', 1), (u'Scala,', 1), (u'Running', 1), (u
'should', 2), (u'environment', 1), (u'to', 14), (u'only', 1), (u'module,', 1), (u'g
iven.', 1), (u'rich', 1), (u'directory.', 1), (u'Apache', 1), (u'Interactive', 2),
(u'sc.parallelize(range(1000)).count()', 1), (u'Building', 1), (u'do', 2), (u'guide
', 1), (u'return', 2), (u'which', 2), (u'Programs', 1), (u'Many', 1), (u'Try', 1),
(u'built,', 1), (u'"yarn-client"', 1), (u'YARN,', 1), (u'not', 1), (u'using', 2),
(u'Example', 1), (u'scala', 1), (u'Once', 1), (u'Spark"(http://spark.apache.org/d
ocs/latest/building-spark.html).', 1), (u'Because', 1), (u'cluster.', 1), (u'name',
1), (u'Testing', 1), (u'Streaming', 1), (u'./bin/pyspark', 1), (u'SQL', 2), (u'thr
ough', 1), (u'GraphX', 1), (u'them,', 1), (u'[run', 1), (u'"yarn-cluster"', 1), (u'
the', 21), (u'abbreviated', 1), (u'set', 2), (u'[project', 2), (u'Scala', 2), (u'##
', 8), (u'thread,', 1), (u'library', 1), (u'see', 1), (u'individual', 1), (u'examp
les', 2), (u'MASTER', 1), (u'runs.', 1), (u'[Apache', 1), (u'Pi', 1), (u'instruction
s.', 1), (u'More', 1), (u'Python,', 2), (u'#', 1), (u'processing,', 1), (u'for', 12
), (u'optimized', 1), (u'its', 1), (u'version', 1), (u'wiki](https://cwiki.apache.o
rg/confluence/display/SPARK).', 1), (u'provides', 1), (u'print', 1), (u'Configurati

```

Obtendremos el resultado de la consulta, con las parejas clave-valor.

Python soporta definir funciones anónimas de una línea en el momento de ejecución (lambda)

### SparkSubmit

Para poder ejecutar scripts completos con Python utilizamos SparkSubmit, que viene incluido en los archivos de Spark, dentro de la carpeta bin.

Dentro de la carpeta /examples/src/main se encuentran ejemplos hechos con java, python y scala para poder realizar pruebas.

Para ejecutar un proceso desde Spark-Submit, salimos de la Shell, y ejecutamos:

```
bin/spark-submit script.py
```

Aquí se muestra el funcionamiento con el ejemplo de contar palabras:

```
$ ./bin/spark-submit examples/src/main/python/wordcount.py "README.md"
```

```

c1@ubuntu: ~/spark-1.5.0-bin-hadoop2.6
15/11/16 04:54:35 INFO DAGScheduler: ResultStage 1 (collect at /home/c1/spark-1.5.0-bin-hadoop2.6/examples/src/main/python/wordcount.py:35) finished in 0.238 s
15/11/16 04:54:35 INFO TaskSetManager: Finished task 0.0 in stage 1.0 (TID 1) in 233 ms on localhost (1/1)
15/11/16 04:54:35 INFO TaskSchedulerImpl: Removed TaskSet 1.0, whose tasks have all completed, from pool
15/11/16 04:54:35 INFO DAGScheduler: Job 0 finished: collect at /home/c1/spark-1.5.0-bin-hadoop2.6/examples/src/main/python/wordcount.py:35, took 2.532407 s
: 67
help: 1
when: 1
Hadoop: 4
"local": 1
including: 3
computation: 1
["Third: 1
file: 1
high-level: 1
find: 1
web: 1
Shell: 2
cluster: 2
how: 2
using:: 1
Big: 1
guidance: 3
run:: 1
Scala,: 1
Running: 1
should: 2

```

Todos los programas que creamos en modo local, pueden ser ejecutados también en modo cluster, de esta manera se pueden crear rápidas aplicaciones para hacer comprobaciones y después ejecutarlas de manera distribuida.

## 5. Rendimiento

Gracias al procesamiento de datos en memoria, y la posibilidad de trabajar en una arquitectura paralela con esclavos, aumenta la velocidad conseguida. El tamaño y la estructura del cluster va a afectar directamente en el rendimiento.

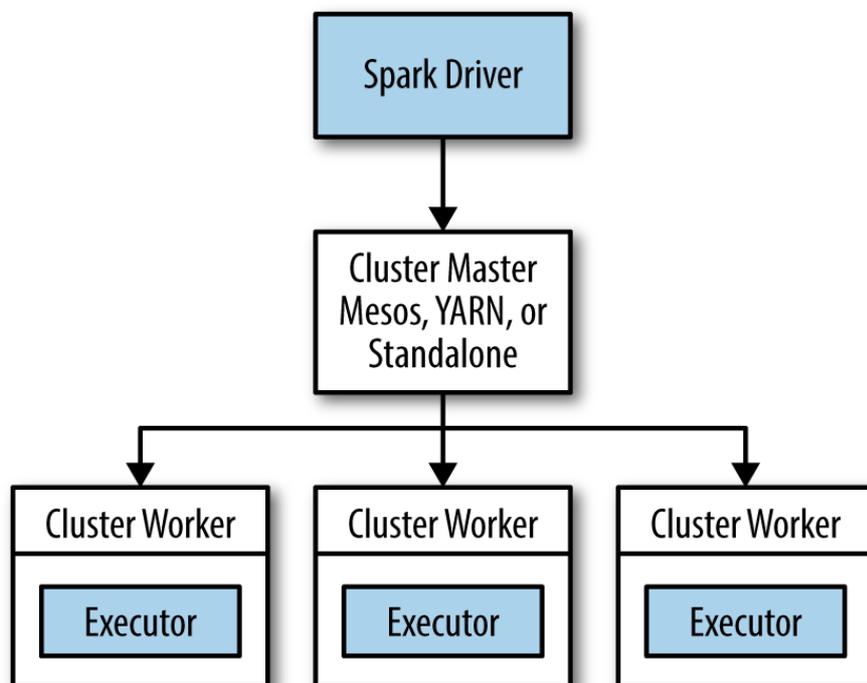
En la pagina oficial de Spark , <https://spark.apache.org/examples.html>, se muestra un gráfico que compara el tiempo de ejecución de un programa con Spark contra una aplicación Hadoop en 100 GB de datos en un Cluster de 100 nodos.

## 6. Caso de uso: Cluster en Spark

### Cluster Spark:

En el modo distribuido de Spark utiliza una arquitectura maestro/esclavo con un coordinador central y trabajadores distribuidos

Un cluster se trata de la unión de varios servidores que trabajan de forma paralela. Para crear un cluster lo podemos hacer usando servidores reales o maquina virtuales, que sera lo que utilizaremos para realizar el ejemplo, aunque no tiene sentido el utilizar las maquinas virtuales, porque se pierde rendimiento, pero podemos simular el funcionamiento de una manera fácil.



Spark se puede ejecutar en diferentes clusters mánager :

**Standalone** → Cluster propio y integrado de Spark. De fácil implementación

**Hadoop YARN** → Yet Another Resource Negotiator. Cluster introducido en Hadoop 2.0, es una versión mejorada de MapReduce

**Apache Mesos** → Apache Mesos es un gestor de cluster de propósito general que puede funcionar tanto de análisis de cargas de trabajo y servicios de larga duración

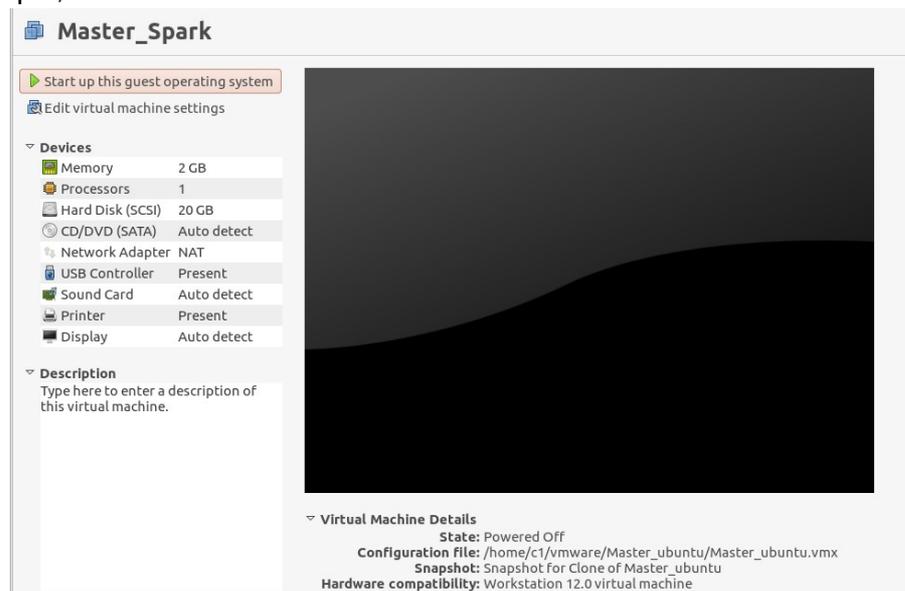
Para empezar una implementación distribuida, se recomienda con el cluster Standalone, ya que viene integrado con Spark y es de fácil implementación

Para crear las maquinas virtuales lo haremos con Vmware, y utilizaremos la LTS de Ubuntu 14.04

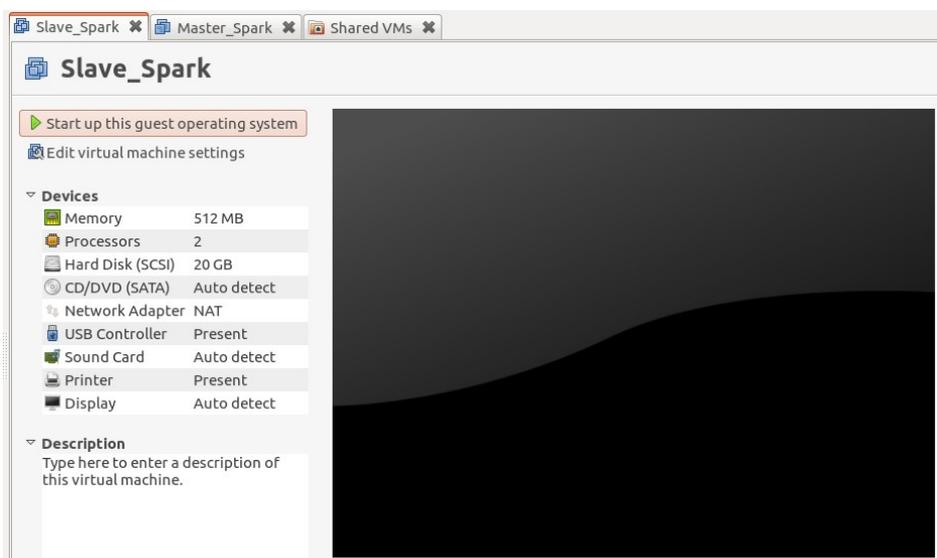
```
wget http://www.vmware.com/go/tryworkstation-linux-64
chmod +x Vmware-Workstation-Full-10.0.0-1295980.x86_64.bundle
sudo ./Vmware-Workstation-Full-10.0.0-1295980.x86_64.bundle
```

Nos pedirá la contraseña del root y continuará con la instalación,

1. Creamos una maquina virtual y le instalamos Apache Spark. Como es un sistema operativo limpio, debemos instalar también Java



2. Cuando todo este configurado, clonamos esta maquina, tantas veces como esclavos queramos tener. En este caso solo la clonaremos una vez. Y también le reduciremos la memoria, se puede modificar en función de lo que cada máquina permita.



Primero miramos cuales son sus Ips para asignar maestro/esclavo



Maestro: 192.168.1.30

Esclavo: 192.168.1.31

Antes de continuar tenemos que saber que la comunicación entre el maestro de Spark y los esclavos es mediante SSH, por ello instalamos en los esclavos:

```
sudo apt-get install openssh-server
```

Configurar el acceso SSH sin contraseña del maestro a los demás. Para ello es necesario tener la misma cuenta de usuario en todas las máquinas.

Creamos una clave SSH privada en el maestro a través de `ssh-keygen`, y la adición de esta clave al archivo `.ssh / authorized_keys` de todos los trabajadores.

Desde el master:

```
$ ssh-keygen -t dsa
```

```
Enter file in which to save the key (/home/you/.ssh/id_dsa): [ENTER]
```

```
Enter passphrase (empty for no passphrase): [EMPTY]
```

```
Enter same passphrase again: [EMPTY]
```

Le damos enter hasta que finalice, por defecto lo guardar en `~/.ssh/ id_rsa.pub`

```

c1@ubuntu: ~/spark-1.5.1-bin-hadoop2.6/sbin
Your identification has been saved in /home/c1/.ssh/id_rsa.
Your public key has been saved in /home/c1/.ssh/id_rsa.pub.
The key fingerprint is:
a5:70:f9:44:42:9d:7e:e3:6e:fd:e8:d3:2f:51:a7:e3 c1@ubuntu
The key's randomart image is:
+--[ RSA 2048]-----+
  |         .o...      |
  |        +o         |
  |       . o.o       |
  |      o = . o  o   |
  |     S .o . o.    |
  |      . +         |
  |       . o +      |
  |      o E..       |
  |       . .o++     |
  +-----+
c1@ubuntu:~/spark-1.5.1-bin-hadoop2.6/sbin$

```

Si queremos tener acceso a los esclavos, esta clave la copiamos como parte pública a `~/.ssh/authorized_keys`. Desde el máster:

```
ssh-copy-id -i ~/.ssh/id_rsa.pub user_slave1@IP_slave1
```

Se ha generado el fichero de `authorized_keys`

En la carpeta `conf`, tenemos los ficheros de configuración, debemos editar dos ficheros dentro del máster:

```

c1@ubuntu: ~/spark-1.5.0-bin-hadoop2.6/conf
c1@ubuntu:~/spark-1.5.0-bin-hadoop2.6/conf$ ls
docker.properties.template      slaves
fairscheduler.xml.template     slaves.template
log4j.properties               spark-defaults.conf.template
log4j.properties~              spark-env.sh
log4j.properties.template      spark-env.sh.template
metrics.properties.template
c1@ubuntu:~/spark-1.5.0-bin-hadoop2.6/conf$

```

- **slaves.template**: dentro de la carpeta `conf/` copiamos y renombramos `slaves.template` por `slaves`, y ponemos la ip del esclavo: 192.168.1.31

- **spark\_env.sh.template**: en la misma carpeta encontramos `spark_env.sh.template`, copiamos y renombramos como `spark_env.sh` y añadimos:

```
#!/usr/bin/env bash
```

```
export SPARK_MASTER_IP= 192.168.1.30
```

```
export SPARK_WORKER_CORES=1 # Número de cores que se ejecutan en la máquina
```

```
export SPARK_WORKER_MEMORY=800m # Memoria total que un worker tiene disponible
```

```
export SPARK_WORKER_INSTANCES=3 # Número de procesos worker por cada nodo
```

Estos dos ficheros los debemos configurar tanto en el máster como en los esclavos.

Ahora toca levantar las maquinas, para ello lo hacemos dentro de la carpeta ~/sbin:

```

c1@ubuntu: ~/spark-1.5.1-bin-hadoop2.6/sbin
c1@ubuntu:~/spark-1.5.1-bin-hadoop2.6/sbin$ ls
slaves.sh                start-slaves.sh
spark-config.sh          start-thriftserver.sh
spark-daemon.sh          stop-all.sh
spark-daemons.sh        stop-history-server.sh
start-all.sh             stop-master.sh
start-history-server.sh  stop-mesos-dispatcher.sh
start-master.sh           stop-mesos-shuffle-service.sh
start-mesos-dispatcher.sh stop-shuffle-service.sh
start-mesos-shuffle-service.sh stop-slave.sh
start-shuffle-service.sh stop-slaves.sh
start-slave.sh            stop-thriftserver.sh
c1@ubuntu:~/spark-1.5.1-bin-hadoop2.6/sbin$ ./start-master.sh
starting org.apache.spark.deploy.master.Master, logging to /home/c1/spark-1.5.1-
bin-hadoop2.6/sbin/./logs/spark-c1-org.apache.spark.deploy.master.Master-1-ubun
tu.out
c1@ubuntu:~/spark-1.5.1-bin-hadoop2.6/sbin$

```

Se puede hacer uno a uno:

```
./start-master.sh
```

```
./start-slaves.sh
```

O levantar todo el cluster:

```
./start-all.sh
```

Una vez se ha iniciado, el **máster** informa de la situación de la aplicación mediante una interficie web, para verlo abrimos un navegador y vamos a <http://localhost:8080>, y vemos que ya tenemos al worker preparado.

Spark Master at spark://192.168.1.30:7077 - Mozilla Firefox

Spark Master at spark://192.168.1.30:7077

URL: spark://192.168.1.30:7077  
 REST URL: spark://192.168.1.30:6066 (cluster mode)  
 Alive Workers: 1  
 Cores in use: 2 Total, 0 Used  
 Memory in use: 800.0 MB Total, 0.0 B Used  
 Applications: 0 Running, 0 Completed  
 Drivers: 0 Running, 0 Completed  
 Status: ALIVE

**Workers**

Worker Id	Address	State	Cores	Memory
worker-20151117015308-192.168.1.31-38876	192.168.1.31:38876	ALIVE	2 (0 Used)	800.0 MB (0.0 B Used)

**Running Applications**

Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
----------------	------	-------	-----------------	----------------	------	-------	----------

Firefox automatically sends some data to Mozilla so that we can improve your experience. Choose What I Share

A continuación ejecutamos un ejemplo para que el máster reparta al esclavo, con spark-submit

```
bin/spark-submit --master spark://host:7077 my_script.py
```

Donde `--master` especifica una dirección URL de Cluster para conectarse, en nuestro caso Standalone, por defecto utiliza el puerto 7077.

A parte de esto ofrece una variedad de opciones que le permiten controlar los detalles específicos sobre la configuración:

```
# Submitting a Java application to Standalone cluster mode
```

```
$ ./bin/spark-submit \
  --master spark://hostname:7077 \
  --deploy-mode cluster \
  --class com.databricks.examples.SparkExample \
  --name "Example Program" \
  --jars dep1.jar,dep2.jar,dep3.jar \
  --executor-memory 10g \
  myApp.jar "options" "to your application" "go here"
```

Ejecutaremos el mismo ejemplo de contar palabras que hemos utilizado anteriormente con la shell de Python, pero en este caso lo haremos distribuido en el Cluster. Desde el terminal del máster ejecutamos:

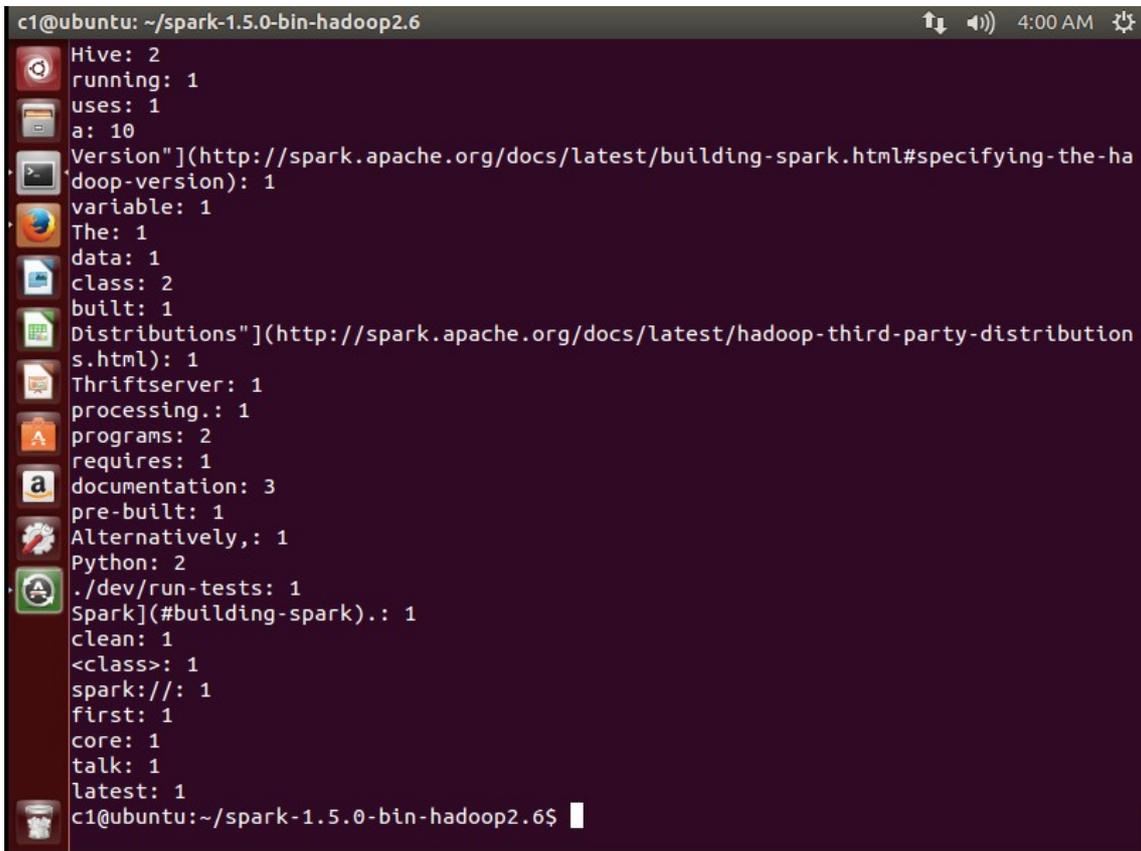
```
c1@ubuntu:~/spark-1.5.0-bin-hadoop2.6$ ./bin/spark-submit --master
spark://192.168.1.30:7077 --num-executors 2
examples/src/main/python/wordcount.py "README.md"
```

Abrimos el navegador y podremos ver como se esta ejecutando la aplicación:

The screenshot shows the Spark Master web interface. At the top, it says 'Spark Master at spark://192.168.1.30:7077 - Mozilla Firefox'. The browser address bar shows 'localhost:8080'. The main content area is divided into three sections:

- Worker Id:** A table with columns: Worker Id, Address, State, Cores, Memory. One worker is listed as 'worker-20151130033459-192.168.1.31-50091' with state 'ALIVE', 2 Cores, and 1024.0 MB Memory.
- Running Applications:** A table with columns: Application ID, Name, Cores, Memory per Node, Submitted Time, User, State, Duration. One application is listed: 'app-20151130034428-0002' (PythonWordCount) with state 'RUNNING'.
- Completed Applications:** A table with columns: Application ID, Name, Cores, Memory per Node, Submitted Time, User, State, Duration. Two applications are listed: 'app-20151130034303-0001' and 'app-20151130034145-0000', both with state 'FINISHED'.

Y el resultado en el terminal es el mismo que hemos visto antes:



```
c1@ubuntu: ~/spark-1.5.0-bin-hadoop2.6 4:00 AM
Hive: 2
running: 1
uses: 1
a: 10
Version](http://spark.apache.org/docs/latest/building-spark.html#specifying-the-hadoop-version): 1
variable: 1
The: 1
data: 1
class: 2
built: 1
Distributions](http://spark.apache.org/docs/latest/hadoop-third-party-distributions.html): 1
Thriftserver: 1
processing.: 1
programs: 2
requires: 1
documentation: 3
pre-built: 1
Alternatively,: 1
Python: 2
./dev/run-tests: 1
Spark](#building-spark).: 1
clean: 1
<class>: 1
spark://: 1
first: 1
core: 1
talk: 1
latest: 1
c1@ubuntu:~/spark-1.5.0-bin-hadoop2.6$
```