`

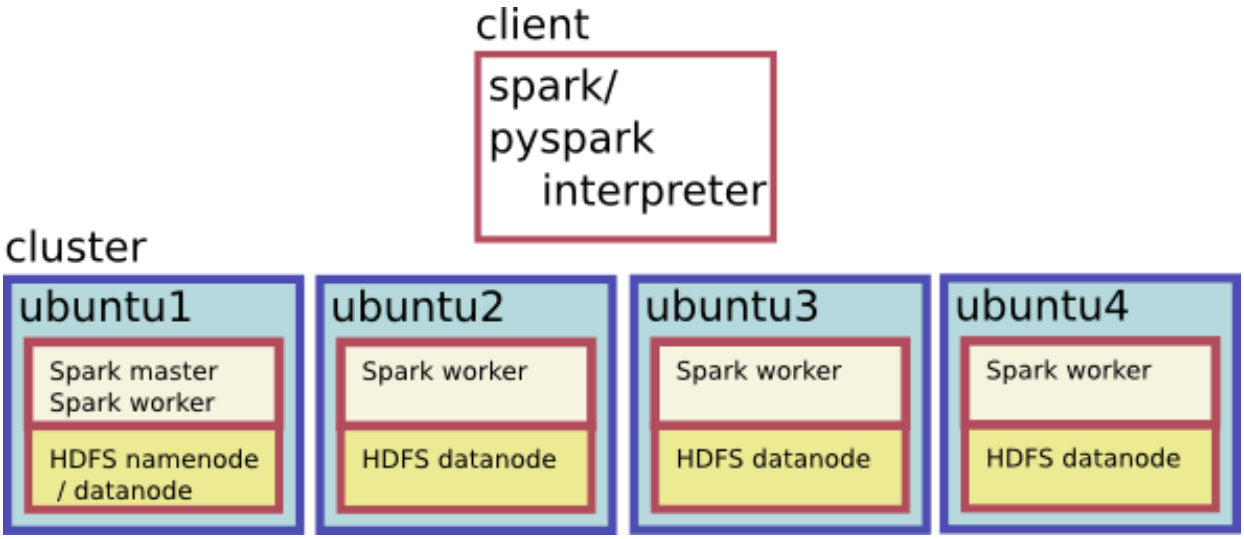
Getting started with PySpark - Part 1

Apache Spark (https://spark.incubator.apache.org/) is a relatively new data processing engine implemented in Scala and Java that can run on a cluster to process and analyze large amounts of data. Spark performance is particularly good if the cluster has sufficient main memory to hold the data being analyzed. Several sub-projects run on top of Spark and provide graph analysis (GraphX), Hive-based SQL engine (Shark), machine learning algorithms (MLlib) and realtime streaming (Spark streaming). Spark has also recently been promoted from incubator status to a new top-level project.

In this series of blog posts, we'll look at installing spark on a cluster and explore using its Python API bindings *PySpark* for a number of practical data science tasks. This first post focuses on installation and getting started.

Installing Spark on a Hadoop cluster

I found installing spark to be very easy. I already have a cluster of 4 machines (ubuntu1,ubuntu2,ubuntu3 and ubuntu4) running Hadoop 1.0.0. I installed spark on each of these machines, and another copy on a separate machine to be used as a client. I obtained spark 0.9.0 from the downloads page (https://spark.incubator.apache.org/downloads.html). I downloaded the spark binaries for Hadoop 1.



On all of the machines I installed spark to the same directory: /home/dev/spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1

Of course you don't have to install to this exact location, but it seems that you should install spark to the same location on each server and client machine. I hit some errors when my spark client installation was installed to a different path to the server machines.

Configuring and starting spark

There are a number of ways to deploy spark. Possibly the easiest is to deploy spark in **standalone mode** into a cluster. Spark will run a master server and one or more worker servers. Spark's documentation on installing standalone mode to a cluster (https://spark.incubator.apache.org/docs/latest/spark-standalone.html) has full details. Both the master and the workers should be started from the machine running the master. Note that you'll also need to configure passwordless ssh between the machine running the master and each machine running a worker. If you need to do that, you may want to look at this this article (http://www.debian.org/devel/passwordlessssh).

I only needed to configure the installation of spark on the master machine, by configuring the file **conf/slaves**:

```
# A Spark worker will be started on each of the machines listed below.
ubuntu1 ubuntu2 ubuntu3 ubuntu4
```

I created a couple of scripts that I can run from the client machine to ssh to the master node (ubuntu1) and start and stop the spark master and workers. You can ssh to the master node and run sbin/start-master.sh, sbin/stop-master.sh, sbin/start-slaves.sh and sbin/stop-slaves.sh directly if you prefer.

Script start_spark.sh

```
#!/bin/sh

MASTER=ubuntu1

ssh $MASTER "(cd spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1;./sbin/start-master.sh)"

ssh $MASTER "(cd spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1;./sbin/start-slaves.sh)"
```

Script stop_spark.sh

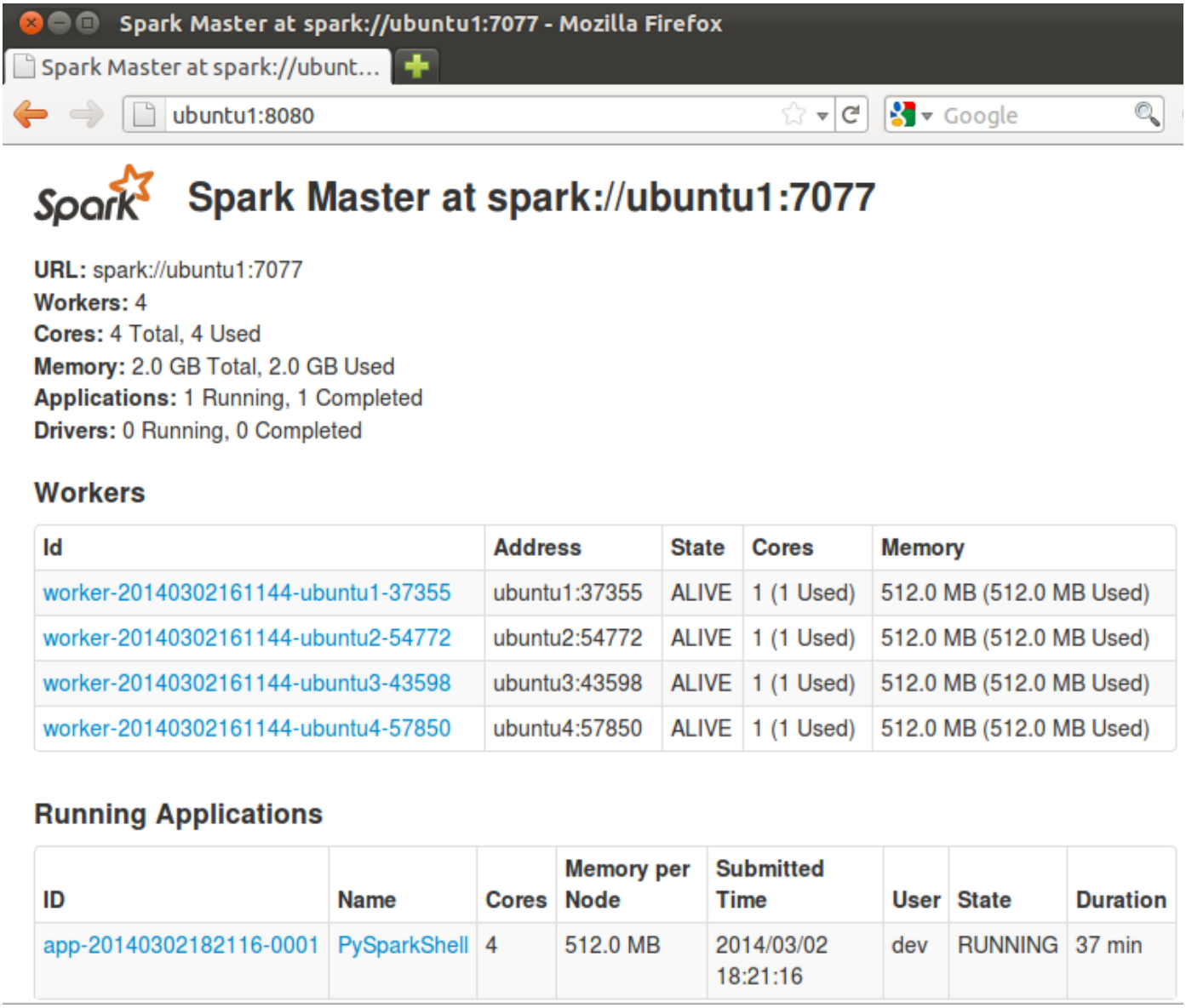
```
#!/bin/sh

MASTER=ubuntu1

ssh $MASTER "(cd spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1;./sbin/stop-slaves.sh)"

ssh $MASTER "(cd spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1;./sbin/stop-master.sh)"
```

After starting the master and the workers you should be able to point your browser at port 8080 on the master node - this web page displays spark's status. Also, ensure that HDFS is running on the cluster.



In the client installation of spark, the only configuration step I needed was to reduce the loglevel (which by default is quite verbose). To do this:

```
cd /home/dev/spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1/conf
cp log4j.properties.template log4j.properties
```

edit log4j.properties and change the following line to set log4j.rootCategory to log only warnings and errors:

```
...
log4j.rootCategory=WARN, console
...
```

Testing the installation

You should now be able to start the PySpark interpreter on the client machine, using the following command (the MASTER environment variable needs to be set to tell spark client where the master service is running, the URL to which it should be set is displayed at the top of the web page displayed at port 8080 on the Spark master node):

```
cd /home/dev/spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1
MASTER=spark://ubuntu1:7077 bin/pyspark
```

All being well, you should see the following when starting the pyspark interpreter:

```
dev@dev-desktop:~/spark-0.9.0/spark-0.9.0-incubating-bin-hadoop1$ MASTER=spark://ubuntu1:70
77 bin/pyspark
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 0.9.0
      /_/

Using Python version 2.7.2+ (default, Jul 20 2012 22:15:08)
Spark context available as sc.
>>>
```

To test the installation, I obtained the texts of the 100 most popular books from project Gutenberg (http://www.gutenberg.org/) and copied them to folder /user/dev/gutenberg on HDFS. PySpark can be used to perform some simple analytics on the text in these books to check that the installation is working.

First we create a spark Resilient Distributed Dataset (RDD) (http://spark.incubator.apache.org/docs/latest/api/pyspark/pyspark.rdd.RDD-class.html) containing each line from the files in the HDFS folder:

```
>>> lines = sc.textFile('hdfs://ubuntu1:54310/user/dev/gutenberg')
```

PySpark provides operations on RDDs to apply transforms produce new RDDs or to return some results. To filter out empty lines we can use the following filter transformation.

```
>>> lines_nonempty = lines.filter( lambda x: len(x) > 0 )
```

At this point, no actual data is processed. Spark/PySpark evaluates lazily, so its not until we extract result data from an RDD (or a chain of RDDs) that any actual processing will be done. The following code returns the number of non-empty lines:

```
>>> lines_nonempty.count()
1240997
```

To run the traditional wordcount example, see python/examples/wordcount.py in the spark installation. You can also run the following from the PySpark interpreter to find the 10 most commonly occurring words with their associated frequencies (not suprisingly, these are the usual stopwords):

```
>>> words = lines_nonempty.flatMap(lambda x: x.split())
>>> wordcounts = words.map(lambda x: (x, 1)).reduceByKey(lambda x,y:x+y).map(lambda x:(x[1]
,x[0])).sortByKey(False)
>>> wordcounts.take(10)
[(463684, u'the'), (281800, u'and'), (281055, u'of'), (185953, u'to'), (138053, u'a'), (123
173, u'in'), (91436, u'that'), (88530, u'I'), (65400, u'with'), (63112, u'he')]
```

In the next post in this series (../pyspark2/index.html) I'll dig into how the above wordcount example works and look at some of the different transformations that PySpark supports.

7 Comments

Leave a comment

Anti-Spam Check

Hexadecimal number

0x19

Decimal equivalent

Comment

Name

Comment

Submit