An Introduction To

Spark

About Spark

Apache Spark is a general purpose cluster computing system. It provides APIs in Java, Scala and Python allowing the programmer to create custom programs and functions that take advantage of the power that the cluster provides. Apache Spark is commonly used as an alternative to MapReduce for certain applications. In practice, Spark can be 100 times faster than a comparable MapReduce job for iterative algorithms or interactive data mining. Spark runs on a Hadoop cluster with HDFS.

Why is Apache Spark faster than a traditional MapReduce job? The answer to that is that Spark uses in memory cluster computing. It keeps all of its data for its jobs in memory, and since memory accesses are much faster than disk accesses, the jobs complete more quickly. Because it uses Hadoop and HDFS and in memory computing it scales to very large clusters and has increased speed. Its high level APIs allow developers to get started quickly and provide powerful tools.

Since Spark is much faster than a MapReduce job and scales as well, one might ask oneself whether MapReduce will become obsolete once Spark becomes more commonplace in the Hadoop ecosystem. Spark performs its tasks much faster, can do the same tasks if programmed correctly, and scales so why would one still use a MapReduce job. The answer here is reliability. Spark's greatest strength is that it runs quickly because it is an in memory job, and doesn't use the write backs to disk that MapReduce uses. This is also its greatest weakness. What happens in the event of a failure? In a MapReduce job it is possible to recover from catastrophic failures of nodes and clusters if the written back files can be recovered. With Spark there is no such protection. The entire job is done in memory, in case of failure everything that was currently being worked on is lost. Since MapReduce and Spark are designed to be run on very large data sets, a failure in the middle of processing could have serious consequences. However, for relatively stable environments, Spark is a viable alternative for traditional MapReduce, especially for applications that require faster processing times or interactivity.

Installation Instructions

For this lab, it is required that you install Apache Spark on your cluster, or sandbox. Follow these steps to install Apache Spark.

1. Ssh into your cluster
2. Install ftp. Ftp will be used to get the spark install files.
    2.1. In your terminal execute: *yum install ftp*
    2.2. Verify that ftp has installed successfully: Launch an interactive ftp session by executing *ftp*
3. Open a connection to the mirror which hosts the spark installer
    3.1. In the ftp interactive session execute *open apache.mirrors.tds.net*
    3.2. When prompted, use the username *anonymous* and an empty password
4. Once the connection has been opened execute the following commands in the interactive ftp session in order to pull the installer files
    4.1. binary
    4.2. cd/pub/apache.org/spark/spark-1.1.0/
    4.3. get spark-1.1.0-bin-hadoop2.4.tgz
    4.4. close
    4.5. exit
5. The following commands untar the spark binaries and move them into your user binaries.
    5.1. tar xzvf spark-1.1.0-bin-hadoop2.4.tgz
    5.2. mv spark-1.1.0-bin-hadoop2.4 /usr/lib
6. Next, add spark to the system's path to make it executable from the command line outside of its own directory. This is done by editing the bashrc file like so
    6.1. nano /etc/bashrc
    6.2. At the end of the file, add *PATH=$PATH:/usr/lib/spark-1.1.0/bin*
7. Exit your ssh session and re-login to reload the bashrc file so Spark can be accesed through the newly modified PATH variable.
8. Start pyspark by executing *pyspark*
9. In the REPL execute the command *sc.parallelize(range(1000)).count()* if this returns 1000, you have successfully installed Spark!
10. Install spark on all nodes of the cluster. This ensures that not only will the job run, but you can start and execute a job on any node in your cluster.

Getting Started

This lab will be designed for and use python as the language of choice. Spark does provide APIs for both Java and Scala as well, but they will not be used here. If you want to investigate using these other languages for use with Spark there is documentation at the spark home page at http://spark.apache.org/docs/latest/quick-start.html that provide insight into how to use these languages.

For help with programming each of these problems please reference the Spark Programming Guide found at http://spark.apache.org/docs/latest/programming-guide.html. This document contains examples for python, java, and scala, and describes the functions available in Spark.

To submit any python job for Spark to process, use the command `spark-submit example.py arg1 arg2` … this will submit the job for processing by spark on the cluster if it has been successfully installed.

Example - Word Count MapReduce

Spark can be used as a tool to perform a standard MapReduce job. To illustrate this fact we will perform a task previously completed when learning MapReduce. The job counts the occurrences of each word in a file, ignoring case and any non-alphanumeric characters. The input file to this program will be a standard text file passed in as a parameter. The output will also be a text file with its location specified as a parameter. The solution below involves the following steps:

1. Open the file using SparkContext
2. Use Flatmap to split each line of the record into individual words
3. Map each individual word separately and ensure that it is converted to uppercase and stripped of all characters that are non-alphanumeric
4. Reduce by key to add together the occurrences
5. Write the results to a textfile using SparkContext

```
import sys
import re
from pyspark import SparkContext
if __name__ == "__main__":
        if (len(sys.argv) != 3):
                print >> sys.stderr, "Usage: wordCount <input> <output>"
                exit(-1)
        sc = SparkContext(appName="PythonWordCount")
        lines = sc.textFile(sys.argv[1], 1)
        pattern = re.compile('[\W_]+')
        counts = lines.flatMap(lambda line: line.split(" ")) \
               .map(lambda word: (pattern.sub('',word.upper().encode('ascii')), 1)) \
               .reduceByKey(lambda a, b: a + b)
        counts.saveAsTextFile(sys.argv[2])
```

Note that when passing in path locations as arguments, the full hdfs path must be entered. For example, the command below properly accesses hdfs:

```
spark-submit wordCount.py hdfs://hadoop04.csse.rose-hulman.edu/tmp/wordCountInput.txt
hdfs://hadoop04.csse.rose-hulman.edu/tmp/wordCountOutput
```

Task - Max Temperature

Now that you have some experience writing a MapReduce job using Spark, we would like you to do it again. Compute the maximum temperature from a temperature dataset organized in the same format as Lab 1. Use the temperatureSample.txt file as your input. Pass the input filepath and output directory in as arguments.

For the temperatureSample.txt file, the expected output should display

```
(1949, 111)
(1950, 22)
```

Estimating Pi

In addition to MapReduce, Spark allows developers to easily write distributed programs. In many ways, a distributed program using Spark looks similar to a typical python program. The key point will be to make use of SparkContext's parallelize method. This and other useful functions (such as map and reduce) are described at http://spark.apache.org/docs/latest/programming-guide.html.

To show off distributed programming, we will be estimating the value of pi via Monte Carlo methods. We generate some number of test points with x and y in the range [0, 1] and determine if they lie within or on the unit circle. By dividing the number of points within or on the circle by the total number of points and multiplying by four (to account for the four quadrants of the circle), we can approximate a value for pi.

For this task:

1. Write a script that accepts the number of test points as an integer argument
2. Write a function that generates points and tests their location
3. Generate and test the test points in parallel
4. Calculate and print the result

The code below provides one solution to this task.

```
import sys
from pyspark import SparkContext
from random import random

def calculate_pi(sample_count):
    sc = SparkContext(appName="PiPy")
    samples = float(sc.parallelize(xrange(sample_count)).map(sample).sum())
    print "\nPi is approximately", 4 * (samples / sample_count)

def sample(n):
    return random()**2 + random()**2 <= 1

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print "Usage: python pi.py <number of samples>"
        exit(-1)
    sample_count = sys.argv[1]
    try:
        sample_count = int(sample_count)
    except ValueError:
        print "Usage: python pi.py <number of samples>"
        exit(-1)
    calculate_pi(sample_count)
```

Spark Streaming

Another feature that spark provides is the ability to handle stream processing. Currently, only Java and Scala are support, though python is currently under development. Streaming is done by creating a Streaming Context, which takes in a Spark Context. The Streaming Context must be started to begin processing data, and must also be told to wait for the computation to terminate.

Task - Word Count Streaming

This task asks you to mimic the word count example performed earlier using streaming input from a TCP Server rather than a text file. Also, print the results to console rather than saving the results. The program should take in the hostname and port as arguments, and perform analysis on data every second.

Maven Dependency:

In the pom.xml file of your project, add the following dependency:

Group Id:	org.apache.spark
Artifact Id:	spark-streaming_2.10
Version:	1.1.0

Documentation:

The following documentation will be useful in writing your program:

[JavaStreamingContext]
[Interface JavaDStreamLike (for flatMap and mapToPair)]
[JavaPairDStream (for reduceByKey and print)]

Testing the Program:

Begin by moving your jar to the root directory of your node. The example below assumes my jar is named NetworkWordCount.jar.

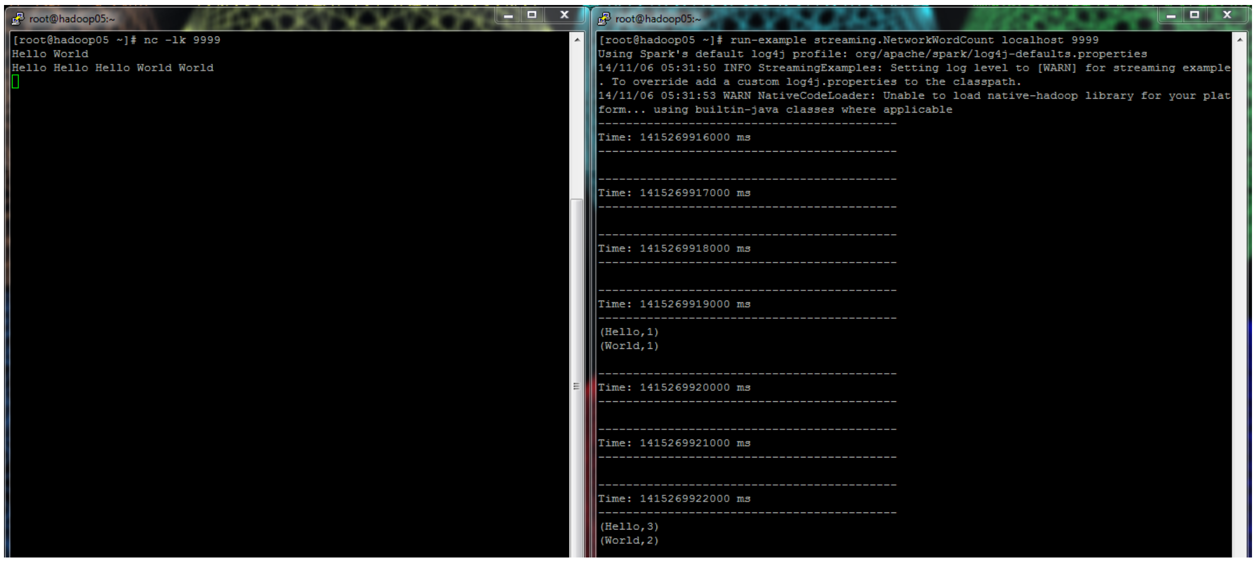
Open up two terminals on your node. The first will be used as are server to listen to, and is set up by running a Netcat server.

nc -lk 9999

On the second terminal, run the program using the following command:

run-example streaming.NetworkWordCount localhost 9999

Every second, you should see the program print to the console. In the first terminal, enter in text to see the program perform the word count analysis.



```
[root@hadoop05 ~]# nc -lk 9999
Hello World
Hello Hello Hello World World
```

```
[root@hadoop05 ~]# run-example streaming.NetworkWordCount localhost 9999
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
14/11/06 05:31:50 INFO StreamingExamples: Setting log level to [WARN] for streaming example
. To override add a custom log4j.properties to the classpath.
14/11/06 05:31:53 WARN NativeCodeLoader: Unable to load native-hadoop library for your plat
form... using builtin-java classes where applicable
-------------------------------------------
Time: 1415269916000 ms
-------------------------------------------

-------------------------------------------
Time: 1415269917000 ms
-------------------------------------------

-------------------------------------------
Time: 1415269918000 ms
-------------------------------------------

-------------------------------------------
Time: 1415269919000 ms
-------------------------------------------
(Hello,1)
(World,1)

-------------------------------------------
Time: 1415269920000 ms
-------------------------------------------

-------------------------------------------
Time: 1415269921000 ms
-------------------------------------------

-------------------------------------------
Time: 1415269922000 ms
-------------------------------------------
(Hello,3)
(World,2)
```