ATLANTIS PRESS

Advances in Engineering Research, volume 127

3rd International Conference on Electrical, Automation and Mechanical Engineering (EAME 2018)

Test Spark Installation on Linux Clusters

Lee-Hur Shing[1,*], Lee-Pin Shing[2], Marn-Ling Shing[3] and Chen-Chi Shing[2]

[1]Information Technology Department, Virginia Tech, Blacksburg, VA, USA

[2]Information Technology Department, Radford University, Radford, VA, USA

[3]Early Child Education Department and Institute of Child Development, University of Taipei, Taiwan

[*]Corresponding author

***Abstract*—Apache Spark is an open source big data analytic framework on clusters. It runs on top of Hadoop distributed clusters but faster using in-memory processing. Very often it seems run normally even Hadoop runs abnormally. This paper describes that after install Spark 1.4.0 over Hadoop 2.7.0, how to test both Hadoop and Spark on CentOS 7.0 Linux clusters.**

Keywords—spark cluster; big data; cloud computing

I. INTRODUCTION

Computers become indispensable in everyday life globally. People use all kinds of computers from smartphones to supercomputers in their work and home environment to access information and communicate through Internet. Home appliances and motor vehicles can be controlled remotely. Robots become commonly used to do repetitive and dangerous work in manufacturing. Huge amount of data are collected and also analyzed using computers to optimize the processing and help to make business decisions. The database management system such as Oracle and MS Access are used to store and process structured data. However, for unstructured data such as email, a NoSQL [1] technology is used in the Big Data technology [2]. Google first applies parallel computations on distributed environment using Map Reduce model [3]. Afterwards Apache creates an open source Hadoop [4] used in Cloud Computing [5] such as Amazon Web Services, Google Cloud Platform, Microsoft Azure and Rackspace. Hadoop, written in JAVA language, uses Hadoop Distributed File System(HDFS) to store and retrieve data across nodes within a cluster and guarantees fault tolerant. Although Hadoop becomes the de-facto Bid Data technology, it is stable, however, generally runs very slow.

Apache Spark [6], designed to improve the Hadoop's processing speed, written in the functional and object oriented mixed language Scala, has been tested faster in big data analytics [7]. It uses read-only data objects called resilient distributed datasets (RDD) [8] and in memory concurrency within a cluster to run efficiently [9]. It is good for processing dynamic streaming data.

Spark has the following useful packages for applications [10]:

- Spark SQL: regular database SQL language for handling structured data
- Spark R: data statistical analysis tool similar to R language
- Streaming: perform streaming analytic for unstructured data
- MLIB: library for machine learning framework
- Graph X: programming tool for implementing graph algorithm using parallelism

Spark can be installed on many different cluster frameworks such as Hadoop. Since Hadoop is stable on static big data processing, Hadoop can be installed before installing Spark on CentOS 7.0 Linux clusters. After installing Hadoop 2.7.0 and Spark 1.4.0 on Linux clusters [11], Hadoop will be tested first before Spark in the next two sections.

II. TEST HADOOP ON MASTER AND SLAVE NODES

A. *Test Map-Reduce Application on Master Using Existing Jar File*

First, Download map-reduce example and store it in /home/hduser mapredexample/datadir. The steps are as below where $ is the prompt:

$ ssh hduser@master; mkdir mapredexample; cd mapredexample; mkdir datadir; cd datadir; wget http://www.gutenberg.org/files/4300/4300.zip; unzip 4300.zip; rm 4300.zip; cat 4300.txt

In order to process a text file with hadoop, you first need to prepare the file to a directory on hadoop, then copy it to the Hadoop File System (HDFS) so that the hadoop namenode and datanodes can share it.

The next step is to create Hadoop HDFS for the downloaded data as below: Go back to home path by $ cd. Then create the corresponding HDFS folder called hdfs_datadir from datadir folder using $hdfs dfs –mkdir –p hdfs_datadir (Note: if you already did this once, you will need to delete your directory hdfs_datadir or use a different hdfs directory name in order to do this next step or you will not get the right results.). $ hdfs dfs –copyFromLocal /home/hduser/mapredexample/datadir hdfs_datadir. To shows list of hdfs files using $ hdfs dfs –ls. Now 4300.txt can be seen in HDFS folder: hdfs_datadir using $ hdfs dfs –ls hdfs_datadir

The third step is to download a map-reduce java program that is included in Hadoop-0.20.2-examples.jar as below:

$ cd; wget http://www.java2s.com/Code/JarDownload/hadoop/hadoop-0.20.2-examples.jar.zip; unzip *.zip; cd mapredexample/datadir

Finally, compile and run wordcount.java and automatically creates a HDFS output folder that contains the output file called part-r-00000 as below: (Note: Do NOT create an HDFS output file beforehand using $hdfs dfs –mkdir –p hdfs_output)

$ hadoop jar /home/hduser/hadoop-0.20.2-examples.jar wordcount hdfs_datadir hdfs_output; hdfs dfs –ls; hdfs dfs –ls hdfs_output; hdfs dfs –cat hdfs_output/part-r-00000 > wordcount_out.txt or $hdfs dfs -copyToLocal hdfs_output/part-r-00000 wordcount_out.txt. And remove the HDFS output directory hdfs_output for next time use by $ hdfs dfs –rm –r hdfs_output (Note: Every time you want to create an output you will need to remove this file or you will get an "Output File Exists Exception" notification.)

B. Test Map-Reduce Application on Master by Creating Own Jar File

First, Copy an examples folder that contains a map-reduce java program:

$ ssh root@master; cd /home/hduser; cp –R /root/hadoop-2.7.0-src/hadoop-mapreduce-project/hadoop-mapreduce-examples/src/main/java/org/apache/hadoop/examples; chown –R hduser:hadoop examples. Now return to hduser on master computer by $ su hduser

Go into the new examples directory you copied from the root by $ cd examples; vi m WordCount.java. And change package line to

package org.myorg;

The next step is to compile WordCount.java and create a Jar file wc.jar as below: $ mkdir WordCountClasses; hadoop com.sun.tools.javac.Main –d WordCountClasses WordCount.java; jar cf wc.jar –C WordCountClasses

The third step is to Prepare a HDFS input file hdfs_datadir as below:

$ mkdir datadir; cd datadir; vim inputdata1.txt

Type in a line: (Use "i" to insert,"Esc" key to exit insert mode, "ZZ" to exit editor)

Hello World

Bye bye World

Edit an input data file using $ vim inputdata2.txt and type in a line:

Hello Hadoop Goodbye Bye Hadoop

Now return to examples folder using $ cd; hdfs dfs –copyFromLocal /home/hduser/examples/datadir hdfs_input; hdfs dfs –ls hdfs_input; hdfs dfs –cat hdfs_input/inputdata1.txt; hdfs dfs –cat hdfs_input/inputdata2.txt

Finally, Run WordCount.java using the created Jar wc.jar. (Note: Do NOT create an HDFS output file beforehand using $hdfs dfs –mkdir –p hdfs_outputdir) and run wordcount on hdfs_datadir file and creates an output directory called "hdfs_outputdir" as below:

$ hadoop jar /home/hduser/examples/wc.jar org.myorg.WordCount hdfs_input hdfs_outputdir; hdfs dfs –cat hdfs_outputdir/part-r-00000 > WordCount_out.txt; cat WordCount_out.txt. It shows

Goodbye	1
Bye	2
Hadoop	2
Hello	2
World	2
bye	1

And remove the HDFS output directory hdfs_outputdir using $ hdfs dfs –rm –r hdfs_outputdir. Eventually, shutdown Hadoop by $ stop-dfs.sh; stop-yarn.sh. In the next section, HADOOP will be installed first on both master and slave nodes.

III. TEST SPARK

The Spark installation can be tested either interactively or in batch. Test Spark interactively are exemplified first:

A. Test Spark Interactively

The examples used in the section can be found in [10].

1) Before it starts, one must have an HDFS hdfs_datadir, which contains a file 4300.txt, ready (See Section IIA). Then follow the steps below: (Note: > is the prompt of an interactive mode)

$ su hduser; spark-shell

> val textFile = sc.textFile("hdfs_datadir")

> textFile.count() // Number of items in this RDD

> textFile.first() // First item in this RDD

> val linesWithMe = textFile.filter(line => line.contains("me"))

> textFile.filter(line => line.contains("me")).count() // Combine last 2 lines: How many lines contain "me"?

> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b) // find the line with the most words

> import java.lang.Math // these 2 lines using JAVA to re-do the last line

> textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b)) // find the line with the most words

> val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a, b) => a + b) // use Map-Reduce Or

> val wordCounts = textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_ + _) // use Map-Reduce

> wordCounts.saveAsTextFile("wordcounts-output")

> wordCounts.count() // find total # of words

> wordCounts.collect() // find (word,# of word) pair

> linesWithMe.cache() // place object in cache for frequent access

> linesWithMe.count() // How many lines contain "me"?

> linesWithMe.count() // How many lines contain "me"?

> exit // quit spark

$ hdfs dfs –cat wordcounts-output/part-00000 > wordcounts_out.txt; sort wordcounts_out.txt|more.

B. Test Spark in Batch

There are five steps described below using four examples to test different tools in Spark:

1) Create a scala program SimpleApp.scala with input data path to test Spark Application

$ mkdir testspark; cd testspark

Example 1.

$ vim SimpleApp.scala

Type the following lines in:

```
/* SimpleApp.scala */

/* This program counts # of lines containing character “a” and counts # of lines containing character “b” */

import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

import org.apache.spark.SparkConf

object SimpleApp {

  def main(args: Array[String]) {

    val logFile = "hdfs_datadir/4300.txt" // Should be HDFS file for input data on your system

    val conf = new SparkConf().setAppName("Simple Application")

    val sc = new SparkContext(conf)

    val logData = sc.textFile(logFile, 2).cache()

    val numAs = logData.filter(line => line.contains("a")).count()

    val numBs = logData.filter(line => line.contains("b")).count()

    println("Lines with a: %s, Lines with b: %s".format(numAs, numBs))

    } }
```

Example 2.

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

import org.apache.spark.SparkConf

import java.lang.Math

object CountWords{

  def main(args: Array[String]){

    val conf= new SparkConf().setAppName("Simple Application")

    val sc= new SparkContext(conf)

    val textFile=sc.textFile("hdfs_datadir/4300.txt")

    val wordCounts= textFile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey((a,b)=>a+b)

    // val wordCounts=textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b)) // find the line with the most words

    wordCounts.saveAsTextFile("hdfs_wordcount-outputs")

    textFile.first() // Reset the input: First item in this RDD

    val linesWithMe = textFile.filter(line => line.contains("me"))

    val linecounts = linesWithMe.count() // How many lines contain "me"?

    println("Lines with me: %s".format(linecounts))

  } }
```

Example 3.

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

import org.apache.spark.SparkConf

import java.lang.Math

object Pi{

  def main(args: Array[String]){

    val NUM_SAMPLES = 2000000000

    val conf= new SparkConf().setAppName("Simple Application")

    val sc= new SparkContext(conf)

    val count = sc.parallelize(1 to NUM_SAMPLES).map{i =>

    val x = Math.random()

    val y = Math.random()

    if (x*x + y*y < 1) 1 else 0

    }.reduce(_ + _)

    println("Pi is roughly " + 4.0 * count / NUM_SAMPLES)

  } }
```

Example 4. Using GraphX

```
import org.apache.spark.SparkContext

import org.apache.spark.SparkContext._

import org.apache.spark.SparkConf
```

```
import org.apache.spark._
import org.apache.spark.graphx._
import org.apache.spark.rdd.RDD
object GraphWorld {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Simple Application")
    val sc = new SparkContext(conf)
    val users: RDD[(VertexId, (String, String))] =
    sc.parallelize(Array((3L, ("rxin", "student")), (7L, ("jgonzal", "postdoc")), (5L, ("franklin", "prof")), (2L, ("istoica", "prof"))))
    // Create an RDD for edges
    val relationships: RDD[Edge[String]] = sc.parallelize(Array(Edge(3L, 7L, "collab"), Edge(5L, 3L, "advisor"), Edge(2L, 5L, "colleague"), Edge(5L, 7L, "pi")))
    // Define a default user in case there are relationships with missing user
    val defaultUser = ("John Doe", "Missing")
    // Build the initial Graph
    val graph = Graph(users, relationships, defaultUser)
    val facts: RDD[String] =graph.triplets.map(triplet => triplet.srcAttr._1 + " is the " + triplet.attr + " of " + triplet.dstAttr._1) facts.collect.foreach(println(_))
    } }
```

2) Create Jar File using SBT

a) Not using GraphX

$ vim simple.sbt

Type the following in:

```
name := "Simple Project"
version := "1.0"
scalaVersion := "2.11.7"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.4.0"
```

$ mkdir src; mkdir src/main; mkdir src/main/scala; mv SimpleApp.scala src/main/scala

Using GraphX

$ vim build.sbt

Type the following in:

```
name := "Graph Project"
version := "1.0"
scalaVersion := "2.11.7"
libraryDependencies += "org.apache.spark" %% "spark-core" % "1.2.0"
libraryDependencies += "org.apache.spark" %% "spark-graphx" % "1.2.0"
```

$ mkdir src; mkdir src/main; mkdir src/main/scala; mv GraphWorld.scala src/main/scala

Check current directory structure

$ find .

It shows:., ./simple.sbt, ./src, ./src/main, ./src/main/scala, ./src/main/scala/SimpleApp.scala

3) Compile and Execute the test scala program on Spark (Note: input and output data files must be in HDFS data file format) using $ sbt package

Use spark-submit and execute on master only using 4 threads

For Example 1:

$ spark-submit –-class “SimpleApp” -–master local[4] target/scala-2.11/simple-project_2.11-1.0.jar > SimpleApp_out.txt; cat SimpleApp_out.txt

It shows:

Lines with a: 23691, Lines with b: 12216

If anything wrong with SimpleApp.java, do the following:

$ vim src/main/scala/SimpleApp.java

Modify your program before you redo $sbt package

For Example 2:

$ spark-submit –class “CountWords” -–master local[4] target/scala-2.11/simple-project_2.11-1.0.jar > lineouts.txt; cat lineouts.txt

It shows:

Lines with me: 5678

If anything wrong with CountWords.java, do the following:

$ vim src/main/scala/CountWords.java

Modify your program before you redo $sbt package

For Example 3:

$ spark-submit –-class “Pi” -–master local[4] target/scala-2.11/simple-project_2.11-1.0.jar > pi.txt; cat pi.txt

It shows:

Pi is roughly 3.14158212

If anything wrong with Pi.java, do the following:

$ vim src/main/scala/Pi.java

Modify your program before you redo $sbt package

For Example 4:

$ spark-submit –-class “GraphWorld” -–master local[4] target/scala-2.11/graph-project_2.11-1.0.jar > GraphWorld_out.txt; cat GraphWorld_out.txt

It shows:

rxin is the collab of jgonzal

franklin is the advisor of rxin

istoica is the colleague of franklin

franklin is the pi of jgonzal

If anything wrong with Pi.java, do the following:

$ vim src/main/scala/Pi.java

Modify your program before you redo $sbt package

Check Spark Log and Hadoop Log

$ more /home/hduser/spark/logs/*; more /usr/local/hadoop/logs/*

If the job is terminated due to not enough data blocks available and spark is in safe mode, then force spark to get out of the safe mode before run the same job again using: $ hdfs dfsadmin -safemode leave

a) Shutdown hadoop

$ stop-dfs.sh; stop-yarn.sh

b) Shutdown spark

$ /usr/local/spark/sbin/stop-all.sh.

Although Spark is powerful in Cloud Computing, it is challenging to install it on Linux. In order for Spark to run correctly, Hadoop must be tested thoroughly. The log for Hadoop on both master and slave nodes needs to be constantly examined.

REFERENCES

[1] N. Leavitt,.”Will NoSQL Databases Live Up to Their Promise?”, *IEEE Compute*, 2010.

[2] J. Hellerstein, “Parallel Programming in the Age of Big Data,” Gigaom Blog, 2008.

[3] R. Lämmel, "Google's Map *Reduce*programming model — Revisited,".Science of Computer Programming, **70**: 1, 2008.

[4] D. Cutting, "New Mailing Lists Request: Hadoop," issues.apache.org, 2006.

[5] G. Gruman, "What cloud computing really means," .InfoWorld, 2008.

[6] The Apache Software Foundation Announces Apache&#8482 Spark&#8482 as a Top-Level Project.apache.org. Apache Software Foundation. 27 February 2014.

[7] R. Xin, “Apache Spark officially sets a new record in large-scale sorting“, https://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html, 2014.

[8] M. Zaharia; M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, J., Michael;S. Shenker and I. Stoica, “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing,” USENIX Symp. Networked Systems Design and Implementation, 2012.

[9] A. Farzad, Justin Langseth, Spark User Concurrency and Context/RDD Sharing at Production Scale. https://spark-summit.org/east-2015/spark-user-concurrency-and-contextrdd-sharing-at-production-scale/, 2015.

[10] Apache Spark: http://spark.apache.org/docs/latest/, 2015.

[11] L. Shing, H. Chao, L. Shing, M. Shing, and C. Shing, “Spark Installation on Clusters”, accepted to be published in the proceedings of 2016 2nd IEEE International Conference on Computer and Communications in press.