NYU

Big Data and Machine Learning HW1a

Nick Greenquist

October 11, 2017

2a) Compute word frequencies across several text documents. You can download large text documents from here – https://www.gutenberg.org. Choose any number you like, a minimum of 10. You can start by modifying the WordCountMapper.java that you worked with in exercise 1 of Hadoop above. (2 points)

InputFiles

hdsf -> /user/ntg251/Books

MapReduce Explanation:

Read file and seperate every line by spaces. Then map every word with the word as the key and a value of 1. Then reduce by key and add up all the values

Hadoop Code:

Java files in the zip, folder 1.a/hadoop

Spark Commands:

```
val file = sc.textFile("/user/ntg251/Books");
val counts = file.flatMap(file => file.split("␣")).map(word => (word, 1)).re
counts.collect().foreach(println);
```

2b) Exercise 2.3.1 in the textbook. Given a large file of integers, compute i) largest integer, ii) average of all the integers, iii) set of unique integers, iv) count of distinct integers. (4 points)

i) Largest integer

InputFiles

hdsf -> /user/ntg251/largeint.txt

MapReduce Explanation:

Map every number in the file with key as 1 and the number as the value. Reduce and compare every value to each other and take the larger value at each compare.

Hadoop Code:

Java files in the zip, folder 1.b/hadoop/largeint

Spark Commands:

```
val nums = sc.textFile("/user/ntg251/largeint.txt");
val pairs = nums.map(a => (1, a.toInt));
val max = pairs.reduceByKey((a,b) => { if(a > b) a else b });
print(max.first());
```

ii) Average

InputFiles

hdsf -> /user/ntg251/largeint.txt

MapReduce Explanation:

Map every number to key 1 and value as the number. In the reducer, sum all the values while also keeping a count and then set the sum/count back as the value.

Hadoop Code:

Java files in the zip, folder 1.b/hadoop/average

Spark Commands:

```
val nums = sc.textFile("/user/ntg251/largeint.txt");
val mean = nums.map(_.toInt).mean();
println(mean);
```

iii) Unique Set

InputFiles

hdsf -> /user/ntg251/largeint.txt

MapReduce Explanation:

Map every number to key as the number and value as 1 (doesn't matter). Then reduce and don't change the value. The result will be a list with all the keys being the unique numbers.

Hadoop Code:

Java files in the zip, folder 1.b/hadoop/uniqueset

Spark Commands:

```
val nums = sc.textFile("/user/ntg251/largeint.txt");
val pairs = nums.map(s => (s, 1));
val set = pairs.reduceByKey((a, b) => 1);
set.collect().foreach(println);
```

iv) Count of Distinct

InputFiles: hdsf -> /user/ntg251/largeint.txt

MapReduce Explanation:

Perform the unique set mapreduce above. Then read that output and map every number to number as key and value of 1. Then reduce by key and sum up the values.

Hadoop Code:

Java files in the zip, folder 1.b/hadoop/distinctcount

Spark Commands:

```
val nums = sc.textFile("/user/ntg251/largeint.txt");
val pairs = nums.map(s => (s, 1));
val set = pairs.reduceByKey((a, b) => 1);
val distinct = set.map(s => (1, 1));
val distinctCount = distinct.reduceByKey((a, b) => a + b);
distinctCount.collect().foreach(println);
```

2c) Matrix-vector multiplication as described in Section 2.3.1 in the textbook. Test your program on a small matrix of size 10x10 and then scale up to $10^6 x 10^6$. (4 points)

InputFiles

hdsf -> /user/ntg251/Matrix
hdsf -> /user/ntg251/MatrixLarge

MapReduce Explanation:

Hadoop: Read in every tuple (M or N, doesn't matter). Map every matrix tuple with key <i, 0> and value <M, j, value>. Map every vector tuple with key <i, 0> where you set i to number of rows of matrix and value <N, i value>. In the reduce, loop through every vaue in key <i, 0>. Place values in two hash maps, depending if they came from M or N. Then loop through number of rows in matrix and increment sum using product of hashmapA[i] and hashmapB[i].

Spark: MapReduce the vector file. Map every tuple to key of i and value of value. Reduce (isn't really needed). The convert the key value pairs to an in memory dictionary. Then map the matrix tuples to key i and the value is the value * the value of the vector dictionary at that tuples i. Then reduce by summing all the values per key.

Hadoop Code:

Java files in the zip, folder 1.c/hadoop
NOTICE: You must set the dimensions of the matrix in Matrix.java
NOTICE: every line in matrix.txt must be <M,i,j,val>
NOTICE: every line in any vector.txt must be <N,i,j,val>

PySpark Code: matrix.py in 1.c NOTICE: every line in matrix.txt must be <M,i,j,val>
NOTICE: every line in any vector.txt must be <N,i,j,val>

Spark Commands:

```
spark-submit matrix.py /user/ntg251/Matrix /user/ntg251/Output/matrix
hadoop fs -getmerge /user/ntg251/Output/matrix $HOME/homework1/1c/output.tx
```

2d) A modified matrix-vector multiplication as described in Section 2.3.2, in which the vector is assumed to be too large to fit into main memory and hence both the matrix and the vector are divided into equal number of stripes. Use the same matrices from the previous problem and repeat. (6 points)

InputFiles

hdsf -> /user/ntg251/MatrixStripe
hdsf -> /user/ntg251/MatrixStripeLarge

MapReduce Explanation:

Hadoop: Read in every tuple (M or N, doesn't matter). Map every matrix tuple with key <i, 0> and value <M, j, value>. Map every vector tuple with key <i, 0> where you set i to number of rows of matrix and value <N, i value>. In the reduce, loop through every vaue in key <i, 0>. Place values in two hash maps, depending

if they came from M or N. Then loop through number of rows in matrix and increment sum using product of hashmapA[i] and hashmapB[i]. This approach works for matrix and vector split into any number of stripes.

Spark: For every vector file, MapReduce the vector file. Map every tuple to key of i and value of value. Reduce (isn't really needed). The convert the key value pairs to an in memory dictionary. Then map the matrix tuples to key i and the value is the value * the value of the vector dictionary at that tuples i. Then reduce by summing all the values per key. Now, repeat this process for every vector and matrix stripe file and merge the RDD's after the matrix RDD is done reducing. The end result is the complete result vector.

Hadoop Code:

Java files in the zip, folder 1.d/hadoop
NOTICE: You must set the dimensions of the matrix in Matrix.java
NOTICE: every line in any matrix.txt must be <M,i,j,val>
NOTICE: every line in any vector.txt must be <N,i,j,val>
NOTICE:The hadoop code will take almost an hour on the massive matrix while spark is about a minute
NOTICE:Same code works for part c and d for hadoop as the input is labeled with M for every matrix triple and N for every vector triple in the file. This allows the hadoop code to handle any amount of splits as all information is known. The spark solution needed more work - see below
NOTICE: I learned much of this approach from this blog post that was related to the Mining Massive Datasets book: https://lendap.wordpress.com/2015/02/16/matrix-multiplication-with-mapreduce/

PySpark Code:

matrix.py in 1.d
NOTICE: every line in any matrix.txt must be <M,i,j,val>
NOTICE: every line in any vector.txt must be <N,i,j,val>

Spark Commands:

If the matrix and vector files are split FIVE ways. I have included commands for using the small matrix or the $10^6 x 10^6$ matrix

```
//Matrix and Vector files must be split with matrix being split vertically

spark-submit matrix.py /user/ntg251/MatrixStripe/matrix.txtaa
/user/ntg251/MatrixStripe/matrix.txtab
/user/ntg251/MatrixStripe/matrix.txtac
/user/ntg251/MatrixStripe/matrix.txtad
/user/ntg251/MatrixStripe/matrix.txtae
/user/ntg251/MatrixStripe/vector.txtaa
/user/ntg251/MatrixStripe/vector.txtab
/user/ntg251/MatrixStripe/vector.txtac
/user/ntg251/MatrixStripe/vector.txtad
/user/ntg251/MatrixStripe/vector.txtae /user/ntg251/Output/stripes

hadoop fs -getmerge /user/ntg251/Output/stripes
```

```
$HOME/homework1/1d/output.txt
s
spark-submit matrix.py /user/ntg251/MatrixStripeLarge/matrix0.txt
/user/ntg251/MatrixStripeLarge/matrix1.txt
/user/ntg251/MatrixStripeLarge/matrix2.txt
/user/ntg251/MatrixStripeLarge/matrix3.txt
/user/ntg251/MatrixStripeLarge/matrix4.txt
/user/ntg251/MatrixStripeLarge/vector0.txt
/user/ntg251/MatrixStripeLarge/vector1.txt
/user/ntg251/MatrixStripeLarge/vector2.txt
/user/ntg251/MatrixStripeLarge/vector3.txt
/user/ntg251/MatrixStripeLarge/vector4.txt /user/ntg251/Output/stripeslarge
hadoop fs -getmerge /user/ntg251/Output/stripeslarge
$HOME/homework1/1d/output.txt
```

3) [4 points] Experiment with varying number of Map tasks and Reduce tasks. The number of Map and Reduce tasks that you choose for your implementation affects the speed. For this exercise, take the simple Word Count application and vary the number of Map and Reduce tasks - Map tasks from 1 to 100 and Reduce tasks from 1 to 100 - and plot a graph of the times taken to execute.