CSS 534
Program 3: Writing an Inversed Indexing Program with MapReduce

Professor: Munehiro Fukuda
Due date: see the syllabus

1. Purpose

In this programming assignment, we will write and execute an inversed-indexing program with MapReduce.

2. Hadoop and MapReduce Installation

You need to install Hadoop and MapReduce on UW1-320 machines. You may install either hadoop-0.20.2, hadoop-2.7.3, or newer. Please note that Lab 3 will use hadoop-0.20.2 that won't use YARN.

Follow the instructions given in TaskParallel1.ppt or css534/programming/MapReduce/Hadoop-2.7.3-installation.docx. For more details, read:

Hadoop: The Definitive Guide, MapReduce for the Cloud, Tom White, O'Reilly, 2009

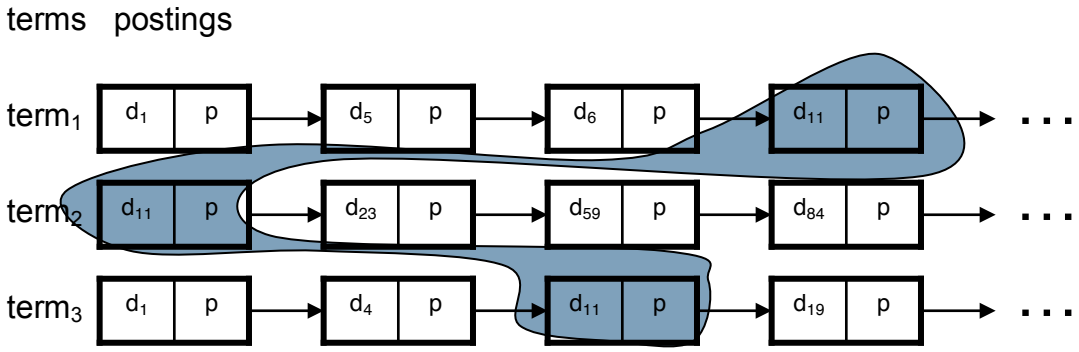
For Lab3, run ~css534/programming/MapReduce/wordcount_2.0/WordCount.java with some input files (which should be different from file01 and file02), and turn in

(1) A snapshot of an MapReduce execution on your account: bin/hadoop jar ....
(2) A snapshot of your /user/yourAccount/output
(3) A file of part-00000

3. Inversed Indexing

Inversed indexing is a batch processing to sort documents for each popular keyword as shown in the figure below, so that it makes easier and faster for document-retrieval requests to find documents with many hits for a set of user-given keywords. For instance, given a request for fining documents with term1, term2, and term3, we will perform intersection of the corresponding three lists of postings and realize that document 11 includes all of these keywords.

4. Algorithm



The main(String[] args) function receives keywords or terms in args. Our goal is to create a list of documents for each keyword. We need to pass these keywords to the map( ) function, for which purpose we will use the JobConf object:

```
public static void main(String[] args) throws Exception {
        // input format:
        // hadoop jar invertedindexes.jar InvertedIndexes input output keyword1 keyword2 ...
        JobConf conf = new JobConf(AAAAA.class);  // AAAAA is this program's file name
```

```
        conf.setJobName("BBBBB");                    // BBBBB is a job name, whatever you like

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(Text.class);

        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

        FileInputFormat.setInputPaths(conf, new Path(args[0]));  // input directory name
        FileOutputFormat.setOutputPath(conf, new Path(args[1])); // output directory name

        conf.set( "argc", String.valueOf( args.length - 2 ) );   // argc maintains #keywords
        for ( int i = 0; i < args.length - 2; i++ )
            conf.set( "keyword" + i, args[i + 2] );              // keyword1, keyword2, ...

        JobClient.runJob(conf);
}
```

Map: We need to retrieve all the keywords from JobConf. When a Map object is instantiated, the MapReduce system automatically calls configure(JobConf job) where we should memorize this job object. The file name that a Map object currently handles can be obtained from the Reporter argument. The following code snippet shows how to retrieve arguments and the current file name in a Map object.

```
public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text,
Text> {
        JobConf conf;
        public void configure( JobConf job ) {
            this.conf = job;
        }
        public void map(LongWritable docId, Text value, OutputCollector<Text, Text> output,
                        Reporter reporter) throws IOException {
            // retrieve # keywords from JobConf
            int argc = Integer.parseInt( conf.get( "argc" ) );
            // get the current file name
            FileSplit fileSplit = ( FileSplit )reporter.getInputSplit( );
            String filename = "" + fileSplit.getPath( ).getName( );
```

The rest of the code is quite straightforward. We will read each line of file splits; tokenize each word with a space; check if it is one of the given keywords; and increment this keyword's count. Once you read all the file splits, you will generate and pass pairs of a keyword and a document id to Reduce.

F.Y.I. my implementation is:
(*keyword1*, *filename_count*), (*keyword2*, *filename_count*), (*keyword3*, *filename_count*), …
where *fiename* is the current file name; *count* is the number of *keyword* appearances; and both *keyword* and *filename_count* are a Text object.

Reduce: Each Reduce object is guaranteed to receive one of the keywords and all document ids, (i.e., *filename_count*). The reducer's algorithm is straightforward, too. We will prepare a hashtable, a tree, or a list to maintain all *filename_count*s, (defined as a document container); read each *filename_count* from value; check if the same *filename* is already stored in the document container; if so, add up the *count* of the current *filename_count* to that found in the document container; otherwise insert the current *filename_count* into the document container. Once you read all values, we will retrieve all *filename_count*s from the document container and write them to the final output.

F.Y.I. my output is

```
public static class Reduce extends MapReduceBase implements Reducer<Text, Text, Text, Text> {
```

```
        public void reduce(Text key, Iterator<Text> values, OutputCollector<Text, Text> output,
                           Reporter reporter) throws IOException {
            // actual computation is here.
            // finally, print it out.
            output.collect(key, docListText );
}
```

where docListText is a string concatenation of all *filename_count*s. For simplicity, you don't have to sort filenames in terms of their count. Of importance is to list all file names including a given keyword.

5. Coding, Compilation, Input Set-up, and Execution

(1) Coding: refer to the following example code when coding your inverted indexing program:

```
~css534/programming/MapReduce/wordcount_2.0
```

(2) Compilation: follow the two instructions below:

```
javac -cp `hadoop classpath`:. AAAAA.java
```

where AAAAA.java is your java program.

```
jar -cvf BBBBB.jar *.class
```

where BBBBB.jar is your jar file name.

(3) Input uploading:

The following directory has all 169 RFC documents regarding IETF (Internet Engineering Task Force) protocol standards and draft standards. Don't copy to your own home or sub directory. Simply upload them to your hadoop account.

```
bin/hdfs dfs -put ~css534/prog3/rfc/* /user/myAccount/rfc
```

where myAccount is your account name.

(4) Execution:

```
bin/hdfs dfs -rmr /user/myAccount/output
```

The above command is intended to make sure that you haven't left the output directory used for the previous run.

```
bin/hadoop jar BBBBB.jar AAAAA rfc output keyword1 keyword2 ...
```

where AAAAA is your program name, and BBBBB is your jar file name.

(5) Output downloading: follow the two instructions below:

```
rm part-00000
```

where part-00000 is a previous output you might have downloaded.

```
bin/hdfs dfs -get /user/myCount/output/part-00000 part-00000
cat part-00000
```

6. Statement of Work

Step 1: Write your InvertedIndexing.java, compile it, and run it with the following five key words.

```
bin/hadoop jar BBBBB.jar AAAAA rfc output TCP UDP LAN PPP HDLC
```

Step 2: Compare the performance between single-node and four-node executions. The following shows my program execution performance

hadoop-0.20.0 on Red Hat Linux with dual-CPU machines
1 computing node: 295.507 seconds
4 computing nodes: 99.951 seconds
Performance improvement: 295.507 / 99.951 = 2.96

hadoop-2.7.3 on Ubuntu Linux with four-CPU-core machines
1 computing node: 8.952 seconds
4 computing nodes: 7.884 seconds
Performance improvement: 8.952 / 7.884 = 1.13

As shown above, it seems like hadoop-2.7.3 itself improved both single and parallel execution. However, please note that, without using YARN, all MapReduce tasks will run locally on the master node, which thus showed no difference from sequential execution.

Although we won't expect any performance improvement in inverted indexing, either with version 0.20.2 or 2.7.3, go through the following procedure to change the number of computing nodes in order to get familiar with it for program 5, (i.e., the final projet).

To change the number of computing nodes involved in the computation, change the hadoop-0.20.2/conf/slaves or hadoop-2.7.3/etc/hadoop/slaves file. You must clear all node-local /tmp/hadoop-yourUnetID/ directories, and thereafter restart hadoop and mapreduce:

(1) Delete the input/output directories:
```
bin/hdfs dfs -rmr /user/myAccount/rfc
bin/hdfs dfs -rmr /user/myAccount/output
```
(2) Stop hadoop and mapreduce.
```
sbin/stop-dfs.sh
```
(3) Change the hadoop-2.7.3/ect/hadoop/slaves file.
(4) Visit each computing node and delete all files of the following directories:
```
cd /tmp; rm -rf hadoop-YourUnetID
```
(5) Reformat and restart hadoop
```
bin/hdfs namenode -format
sbin/start-dfs.sh
```
(6) Create your own account again.
```
bin/hdfs dfs -mkdir /user
bin/hdfs dfs -mkdir /user/myAccount
bin/hdfs dfs -chown myAccount:myAccount /user/myAccount
bin/hdfs dfs -mkdir /user/myAccount/rfc
```
(7) Upload RFC documents again.
```
bin/hdfs dfs -put ~css534/prog3/rfc/* /user/myAccount/rfc
```
(8) Start your program.
```
bin/hadoop jar BBBBB.jar AAAAA rfc output TCP UDP LAN PPP HDLC
```

Step 3: Add a new work to your inversed indexing program: any of (1) to try to use YARN, (2) to sort documents, or (3) to increase input files to see spatial or temporal scalability. In your report, you have to explicitly mention about your additional feature implemented in your program.

Step 4: Consider what if you wrote and ran inverted indexing with MPI.

(1) Explain how you will distribute files over MPI ranks. How tedious is it as compared to MapReduce?
(2) Explain how you will collect document IDs for a given keyword in MPI. How tedious is it as compared to MapReduce?
(3) Estimate the amount of boilerplate code specific to MapReduce and MPI respectively, which is irrelevant to the essence of the inverted indexing algorithm. Which would have a larger amount of boilerplate?
(4) Estimate the execution performance of MapReduce and MPI respectively. Which would run faster?
(5) Explain how you will recover from any computational crash that may happen during inverted indexing in MPI.

7. What to Turn in

This programming assignment is due at the beginning of class on the due date. Please turn in the following materials in a hard copy. No email submission is accepted.

Criteria	Grade
Documentation of your inserted-indexing program in just **one page please**.	20pts
Source code that adheres good modularization, coding style, and an appropriate amount of commends. • 25pts: well-organized and correct code receives • 23pts: messy yet working code or code with minor errors receives • 20pts: code with major bugs or incomplete code receives	25pts
Execution output that verifies the correctness of your implementation and demonstrates any improvement of your program's execution performance. • 25pts: Correct execution both in sequential and parallel versions, and any additional work (YOU MUST EXPLICTLY MENTION ABOUT IT). • 23pts: Correct execution both in sequential and parallel versions • 20pts: Correct execution but no performance comparison with parallel execution • 15pts: Wrong or incomplete work	25pts
Discussions about in one page. Compare MapReduce and MPI in: • File distribution over a cluster system (5pts) • Collective/Reductive operation to create inverted indexing (5pts) • Amount of boilerplate code (5pts) • Anticipated execution performance (5pts) • Fault tolerance; recovery from a crash (5pts)	25pts
Lab Session 3 Please turn in your lab 3 by the due date of program 3. Your source code and execution outputs are required.	5pts
Total Note that program 3 takes 15% of your final grade.	100pts