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 cssmpi1h-8 machines. Lab 3 and Program 3 won't use YARN to make all things simple, (e.g., avoiding the use of Kerberos in Hadoop). For this purpose, we will stick to hadoop-0.20.2. (Expecting that Kerberos is correctly installed on the machines, if you challenge for hadoop-2.7.3 or newer, you can go for it with YARN installation.)

Follow the instructions given in TaskParallel1.ppt or /home/NETID/css534/programming/MapReduce/Hadoop-0.20.2-installation.docx. For more details, read:

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

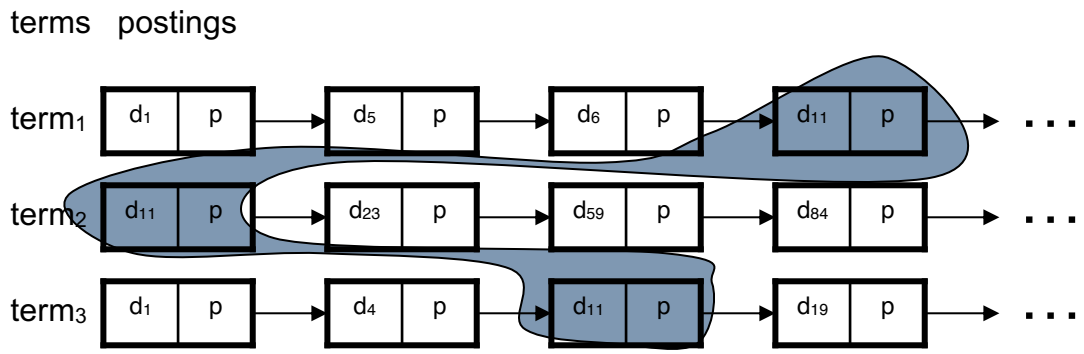
For Lab3, run /home/NETID/css534/programming/MapReduce/wordcount_2.0/WordCount.java with some input files (which should be different from file01 and file02), and turn in

(1) A snapshot of a 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. Note that " " between *filename* and *count* is a space.

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 reduce( ) structure 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:
`/home/NETID/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/hadoop fs -put /home/NETID/css534/prog3/rfc/* /user/myAccount/rfc`
where myAccount is your account name.

(4) Execution:
`bin/hadoop fs -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/hadoop fs -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 four-CPU-core machines
1 computing node: 150.307 seconds
4 computing nodes: 51.120 seconds
Performance improvement: 150.307 / 51.120 = 2.88

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.

Go through the following procedure to change the number of computing nodes involved in the computation. To do so, 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/hadoop fs –rmr /user/myAccount/rfc
bin/hadoop fs –rmr /user/myAccount/output
```
(2) Stop hadoop and mapreduce.
```
bin/stop-all.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/hadoop namenode -format
bin/start-all.sh
```
(6) Create your own account again.
```
bin/hadoop fs –mkdir /user
bin/hadoop fs –mkdir /user/myAccount
bin/hadoop fs –chown myAccount:myAccount /user/myAccount
bin/hadoop fs –mkdir /user/myAccount/rfc
```
(7) Upload RFC documents again.
```
bin/hadoop fs –put /home/NETID/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 additional work to your inversed indexing program: one of the following options:

(1) To increase #input files to see spatial or temporal scalability. You may use html files under ~css534/prog3/blood/. They are from https:/dmice.ohsu.edu/trec-gen/2006data.html and an archive of Blood Journal papers. Don't share them with any other users beyond this class as they are intended for research uses only. If you use a different dataset, please indicate where you got. Don't duplicate RFC documents just for increasing # files, (which won't be counted as additional work).
(2) To sort documents, utilizing MapReduce's shuffle-and-soft feature between map and reduce functions. If you don't use that feature, thus sorting files only within reduce(), the work won't be considered as additional work. Furthermore, this may cause an out-of-memory exception or may complete in success by chance.
(3) To upgrade to Hadoop-2.7.3 or higher and to use YARN. You need Kerberos for inter-machine authentication. If cssmpi1h-12h have installed Kerberos by the time when this assignment was given, you can try this work on these cluster machines. Otherwise, if you choose this option, you need to use your own cluster system, (i.e., Amazon EC2).
(4) To consider any other challenging work such as finding top 5 keywords frequently appeared in the documents. If you take option 4, you should consider a substantial amount of work. Decorating results, sorting documents in reduce( ), or using different keywords is not considered as sufficient work.

In your report, you have to explicitly mention about your additional feature implemented in your program.

Note: regarding the ~css534/prog3/blood/ dataset, if you run the inverted indexes program with 5 keywords such as DNA, platelet, cytokine, macrophage, and erythrocyte, MapReduce will spend:

hadoop-0.20.0 on cssmpi1h-4h.uwb.edu
1computing node: 48,473.851 seconds (13hrs+)
4 computing nodes: 7,211.389 seconds (2hrs)
Performance improvement 48,473.851 / 7,211.389 = 6.72

So, to save your time, you might want to choose a smaller dataset.

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. • Any additional work. YOU MUST EXPLICTLY MENTION ABOUT IT. (2pts) • Performance improvement (2.8 times or better) with four computing nodes. (3pts) • Correct execution (5pts) • Minimum work: incomplete or wrong results. (15pts guaranteed)	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