The Problem

Processing tweet streams Given a twitter tweet stream: list the users that have tweeted about the same pair of hash tags.

```
For instance: For the pair of tags #tag1 and #tag2. The output should list all the users that have tweeted about both of them
tag1, tag2 :userA userB UserC ....
```

This needs to be implemented in the Map/Reduce paradigm, using no auxiliary data structures and no shared data. The only files used are the inputs to the map stage. The solution will demonstrate one potential parallelism tradeoff, it will expand the data sending more data than is intuitively necessary over the network and examining the expanded data in the reducers. The expansion of data prevents random I/O and allows for a high degree of parallelism.

The input to the Map/reduce program will be a twitter stream. You may assume the following about the input:

1. Each new line is a single tweet
2. Each tweet is of the form : username tweetcontent
3. All hash tags will be preceded by a #
4. A tweet may have 0 or more hash tags
5. Duplicate hash tags are possible within the input
6. Case does not matter. For instance #Blackfriday and #blackfriday are considered to be the same hashtag.

An example : kindle_usa fatal voyage: the wrecking of the costa concordia (kindle single):fatal voyage: the wrecking #blackfriday

This is a single tweet with username = kindle_usa and tweetcontent = "fatal voyage: the wrecking of the costa concordia (kindle single): fatal voyage: the wrecking #blackfriday" and one hash tag which is"#blackfriday".

Note: You will need a two phase or chained map reduce program to solve this problem. In other words you will need to two mappers and two reducers. The input given to you will be the input to mapper1. The output of reducer1 will be the input to Mapper 2.

We provide two input data sets. The first tweets.simple is a small set of files for testing and debugging. The larger data set tweets.large is to be used for larger scale testing. The inputs to the project are available at Amazon's S3 to be used in Amazon's Elastic Map/Reduce at s3n with bucket names /tweets.simple and /tweets.large.

Step 1: Installing Hadoop!

Download and install Hadoop! 0.20 on your local machine. This version is required for AWS, not 0.21. Follow the Hadoop! single node guide for standalone operation http://hadoop.apache.org/common/docs/r0.21.0/single_node_setup.html [http://hadoop.apache.org/common/docs/r0.21.0/single_node_setup.html]. I would also recommend following the Hadoop! tutorial http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html [http://hadoop.apache.org/common/docs/r0.20.2/mapred_tutorial.html] for WordCount 1.0. If you don't do this now, no problem, you will do it later when you start working with Hadoop! in Java.

Step 2: Getting a Streaming Version to Work

Write a mapper progam and reducer program in your favorite scripting language. I used Python. My mapper for Phase1 is called is called p1.mapper.py and my reducer is called p1.reducer.py. Similarly my mapper and reducer for Phase 2 are p2.mapper.py and p2.reducer.py. The neat thing about streaming is that you can test a serial version of your code using the following command sequence from the shell:

```
cat tweet.simple/* | python p1.mapper.py  | sort | python p1.reducer.py | python p2.mapper.py | sort | python p2.reducer.py
```

If this produces the correct output, you can run these scripts in the Map/Reduce streaming mode, which looks something like:

Phase 1: /Users/priya/hadoop-0.20.2/bin/hadoop jar /Users/priya/hadoop-0.20.2/contrib/streaming/hadoop-0.20.2-streaming.jar -mapper ./p1.mapper.py -reducer p1.reducer.py -input ./tweets.simple/* -output ./simple.output1

Phase 2: /Users/priya/hadoop-0.20.2/bin/hadoop jar /Users/priya/hadoop-0.20.2/contrib/streaming/hadoop-0.20.2-streaming.jar -mapper ./p2.mapper.py -reducer p2.reducer.py -input ./simple.output1/ -output ./simple.output2

This process is surprisingly easy (or at least we hope it was).

Step 3: Streaming on Amazon's Elastic Map/Reduce

You will need to upload your mapper and reducer programs to a bucket in Amazon's S3 (Simple Storage Service). I used the S3 FireFox plugin [http://www.s3fox.net/] for this task, but there are many other options. Login to the AWS Management Console for Elastic Map Reduce and create a streaming job flow. Here's what mine looked like. Phase 1:

```
Input: s3n://tweets.simple/
Output: s3n://tweet.output/phase1.trial1
Mapper: s3n://tweet.scripts/p1.mapper.py
Reducer: s3n://tweet.scripts/p1.reducer.py
```

Phase 2:

```
Input: s3n://tweet.output/phase1.trial1
Output: s3n://tweet.output/phase2.trial1
Mapper: s3n://tweet.scripts/p2.mapper.py
Reducer: s3n://tweet.scripts/p2.reducer.py
```

No bootstrap actions are necessary. I recommend that you experiment with using the small input set first and configure your job to use only a single small instance of Map/Reduce, because in this way, each failed attempt only costs $0.10 and you find out quickly that whether you have configured your job correctly.

You may choose to use the command line tools for launching map/reduce jobs on AWS instead of the management console. That's up to you.

Step 4: Java Implementation

The streaming Map/Reduce takes some liberties with typing in that the data are not really separated into keys and values. This costs performance. In this step, you will discover how sorting actually works in Map/Reduce. Reimplement your algorithm in Java. The examples in class and the Wordcount 1.0 example provide some guidelines as to how the toolchain works. My compilation and execution process looked something like:

```
javac  -classpath /Users/priya/hadoop-0.20.2/hadoop-0.20.2-core.jar -d tweet_classes tweet.java
jar cvf ./tweet.jar -C tweet_classes/  .
/Users/priya/hadoop-0.20.2/bin/hadoop jar ./tweet.jar cs420.tweet ./tweets.simple java.p1 java.p2
```

This should produce similar output to your streaming version, although not necessarily identical. **Caution** you should not use your Reducer as a combiner class as they do in the wordcount example. Leave the combiner class undefined.

Step 5: Custom JAR on Amazon's Elastic Map/Reduce

Having gotten the Java implementation of your program running, you can now execute your code as a Custom JAR job on AWS. Again, experiment with the small input and a single instance in order to save money.

My job looked like:

```
Jar location: s3n://priya.jars/tweet.jar
Jar arguments: cs420.tweet s3n://tweets.simple/ s3n://tweet.output/simple.jar1/
```

I also recommend turning on logging in the advanced options. I set my log path to:

```
s3n://priya.logs/tweet.log/
```

This will let you know why your job failed.

Having gotten everything working, run the same job on the large input using 4 instances.

Some links to help you get your AWS EMR jobs running.

- http://soam.org/?p=59 [http://soam.org/?p=59]

Submission Instructions

Due date: Monday April 9 2012

Please prepare a PDF document that provides the following items and answers the following questions:

Randal will be revising these questions. I will let you know when they are final.

1. Describe your map/reduce algorithm for the tweet stream processing problem.
    I. Describe the operation of the mappers and reducers in both phases. How does this combination solve the tweet stream processing problem?
    II. What is the potential parallelism? How many mappers does you implementation allow for? Reducers?
    III. What types are used as the input/output to the mappers? Motivate the transformation.
2. On combiners
    I. Why did you leave the combiner class undefined?
    II. Generalize the concept: What sort of computations cannot be conducted in the combiner?
3. Analyze the parallel and serial complexity of the problem and your M/R implementation (in Big-O notation). You should assume that there are n friends list each of length l, i.e. n users that each have l friends.
    I. What is the fundamental serial complexity of the problem? Think of the best serial implementation.
    II. How much work in total (over all mappers and reducers) does the Map/Reduce algorithm perform?
    III. How much work is performed by each mapper? By each reducer?

IV. Based on your answers to the above, describe the tradeoff between complexity and parallelism (qualitatively, you have already quantified it in the previous steps).

Please submit your code for both the streaming version and custom jar version.

1) README.txt: containing step-by-step explanation you used to execute your code on the Amazon S3. 2) Source code including scripts you might have used to run or analyze your output. 3) Do not include your logs. 4) Writeup as requested. 5) Please provide **publicly readable** links in S3 to the output of your streaming program and custom jar program.

Notes

Keep track of your AWS usage. The $100 voucher should be more than sufficient for all course assignments. If you expect to exceed the amount in your voucher, please consult the Instructor or TA. You are responsible for the charges incurred on AWS.

randal/teach/cs420/programming_assignment.txt · Last modified: 2012/04/05 15:10 by priya