Tiny Google: Report

Author: Nick Risinger <ncr13@pitt.edu>
Class: CS1699 Cloud Computing
Professor: Chatree Sangpachatanaruk
Submission Date: 12/17/16

Implementations

Spark

The Spark implementation was the easier of the two. The inverted index was simple to reason about in Scala. The ability to transform the data as required was also quite helpful. The final version sets up the configuration and context and parallelizes the input files. The words are normalized and then split up. The map is flattened so each word is paired with each filename before the count is added and initialized to one. The list is then reduced by adding up the counts for each filename for each word. The resulting posting list structure is then converted to a string and written to the output directory.

MapReduce

The MapReduce implementation was more difficult to implement. It required more forethought about how I was going to create the inverted index while using the more restrictive framework. Despite having outlined the process in the design document, when I went to implement the design, I found that it had flaws. It would have required more work than was necessary, both in implementation and runtime.

After several iterations, I arrived at the final version, which uses a single mapper and reducer to create the inverted index. The program sets up the Hadoop configuration, and creates a new job. The job takes the input directory and maps the files to the PostingMapper. The map function takes each line, normalizes the text, and splits it into words. Each word is used as a key for the intermediate values of a book-count Map. The count is initialized at 1 for every word.

The reducer then takes the intermediate values and combines them by word. The first time a book is encountered for a word, the book is added to the result map. If the book is contained in more than one Map (i.e. meaning the word appeared multiple times in the book), the count is incremented. The final output values are stored as text. Because the posting list for each word is stored as a MapWritable, which does not print out in a human-friendly manner, I had to extend the class to make a better toString function.

Optimization

The spark version has a lot of transformation of the data that could likely be simplified. To speed up execution, the number of partitions could be increased by splitting the files by line instead of by file. The current version also places the entire inverted index into a single partition, so simply removing that repartition and handling multiple output files in the ranking and retrieval stage could further improve performance.

The MapReduce version could also use some improvement. The mapper could count the words contained in the line as opposed to simply placing each word with a count of one into the intermediate values.

To make the program more scalable, the ranking and retrieval could also be done via MapReduce or Spark. Currently it would likely take more time to get results because of setting up the application, but with a larger data source, reading the entire inverted index on the local machine might become slow, unreasonable, or impossible. In the distributed version, the mapper would read in the values and the reducer would select the results that are most relevant up to a certain point.

Performance

Neither implementation has a clear performance advantage over the other, as far as the time of execution for the few text files provided. Both versions take about 12 seconds on average to index the provided input files. This is a bit unexpected because of the performance gains spark can provide. The reason is likely due in part to my Hadoop installation being purely local. I also sort the posting lists in my Spark implementation, while my MapReduce implementation does not sort the lists.