Assignment 2: MapReduce/Hive Top K Words

Authored by Team 21: Shiva Keshav Govindaraju
17 May 2020

The Goal

The intention of this assignment was to accomplish a similar task as we did in Assignment 1, this time with a few additional caveats and technical allowances. We are given access to the SCU Discovery Center's servers as well as the Hadoop cluster and HDFS on the DC server. We are also informed that we need only find the top 100 most common/frequent words (instead of some variable K) in a given input file. Additionally, we have been instructed to make use of two distributed algorithm models for dealing with large quantities of data: MapReduce and Apache Hive. However, we must allow for our code to not only output the top 100 words, but must also design a variation of the application which lists the top 100 words from a subset of the total file consisting only of those words with more than 6 characters (ie. >= 7 characters in a word).

Application Design

For the simplicity of generating this report, we shall assume that the readers are familiar with distributed technologies such as Hadoop and have a basic understanding of MapReduce algorithms and Apache HIVE data query and analysis.

When implementing my overall application design in both MapReduce and in HIVE, I employed similar strategies for both problems: "Top 100 Words and "Top 100 Words Greater Than 6 Characters". The algorithm was two-fold: First, generate a list of all unique words and the number of occurrences for each word that appears in the input file. Second, filter that list to generate the desired output for each problem.

To accomplish these two steps via MapReduce programming, I created several scripts to handle the different aspects of the program: one script would handle the first step of utilizing the MapReduce programming model to generate word-counts for the entire input file, and two scripts for the different kinds of post-processing required to generate the Top 100 words (either with the greater-than-6-chars qualifier or not). The MapReduce algorithm for generating word-counts from an input file is a fairly classic algorithm and has been implemented in several ways.

Given I chose to utilize Python for my programming language (in order to leverage Python's ease of use and simplicity of programming), I made this first step far easier by importing a package, `MRJob`, developed by Yelp to handle the back-end of interfacing with the Hadoop cluster and running the Mapping, Combining and Reducing tasks for generating word-counts. Utilizing the package library, I was able to build an object that could easily be run either locally on the server I was working from or upon the Hadoop cluster and generate word-counts via

Map-Reduce. This involved three steps: Mapping words from the input files to initial values, Combining those (word, count) pairs together, and Reducing the overall results into a singular list of (word, count) pairs.

The second step of my algorithm was accomplished by utilizing Linux shell commands to redirect the output of the MapReduce task to another Python script for post-processing the word-counts. Depending on whether one wanted to filter the output for only words with more than 6 characters, the scripts would parse the word-count pairs through a `Counter` object from the Python `Collections` library (similarly to my implementation for Assignment 1). If using the script that only chose those words with more than 6 characters, words with 6 characters or less were removed in this parsing step. After parsing was complete, the script would sort the word-counts by the count values using the built-in `Counter` function `most_common([n])` and output the top 100 word-count pairs left from the parsing.

The implementation of these problems in Apache Hive were remarkably similar. The first step involved loading the data from the input file into a Hive Table on the distributed warehouse software, and stored in text file format for ease of processing. This table would then be parsed to form another table which found word-count pairs for every unique word in the input file dataset. This was done using built-in "User-Defined Table Generating Functions (UDTFs)" that came with the installation of Hive on the DC servers. Using such functions as `Lateral View`, `explode()`, `split()` and `count()`, I was able to leverage Hive's data-query functions to generate a table of word-counts for the entire input file. Step two involved creating further tables that contained the first 100 entries of the word-count table after it had been sorted in descending order by count. For the two different required outputs, a qualifier was added to different parts of step 2 to determine whether to filter out those words with 6 or fewer characters or not. After all processing was complete, the tables were "dropped" and deleted.

System Requirements

The design and implementation for Top K Words that I devised has a few hardware and software requirements for proper execution:

a. It can only run on a Linux-based OS
    i. Or, the OS must allow for Linux terminal commands to function exactly as they do on a native Linux OS.
b. The machine must have Python 3.6 available (or better)
c. For the Hive portion of this assignment, there must be two directories labeled '`/hive_output_tkw`' and '`/hive_output_tkw_sevenplus`' within the same folder as the HQL script.
    i. The '`/hive_output_tkw`' and '`/hive_output_tkw_sevenplus`' directories must be modified to permit read-write operations from group-users. This may be done via the Linux command `chmod`.
d. The input data file should only contain alphabetic characters (of either case) and spaces.

   i. The input data files and the desired output files should be located on the same server/folder that you are running the scripts from. If the input or output files are located on the Hadoop Distributed File System, the scripts might run into permission errors depending on read-write permissions for the file and the folder it is located in.

e. The Hadoop cluster should be installed and set up properly.
   i. Apache Hive should also be installed and set up properly as well.
f. The machine must have MRJob installed as an available package for python scripts to import and utilize.
   i. This may be accomplished via ‘`pip install mrjob`’ in the command line
   ii. It may be required to edit (or even create) the `mrjob.conf` file to resolve issues between internal bash scripts of both the Hadoop cluster and MRJob. It’s not a common bug. The file can/should be found in either `/etc/mrjob.conf` or `~/.mrjob.conf` depending on where you’re running the MapReduce scripts. The configuration file contents I used are below (modify as fits your own set-up):

```
runners:
  hadoop:
    setup:
      - 'set -e'
    sh_bin: '/bin/bash -x'
```

Limitations

Given our access to the SCU Discovery Center’s servers and the Hadoop cluster for computation, while I still labored under the same limitations as I did for Assignment 1, my lacking computational power on my laptop was less of a limiting factor as it was for Assignment 1: Top K Words. I was also able to take advantage of the server’s larger RAM and access to the cluster for faster computation. Mostly, I struggled with connecting to the cluster properly and ensuring that data was being moved from the local server to the HDFS and back correctly.

Data Analysis

For the purposes of analyzing my implementation of MapReduce and Hive for the purposes of Top 100 Words, I used the 400 MB text file given by the TA. The file contains the dataset for the novel, The Scarlet Pimpernel, and I shall refer to it as such as “400 MB dataset” is cumbersome.

After experiencing such difficulties in implementing a Top K Words application in Assignment 1, the main metrics I focused on in this assignment were execution time, code length and ease of use. As evinced by my report for Assignment 1 and a perusal of my submitted code, it should be clear that my implementation for a Top K Words application on my limited resources was clunky, required long and somewhat complicated code, and took an exceptionally long period of time to execute, even for modest datasets like the Scarlet Pimpernel data.

The utilization of MapReduce programming models and Apache Hive's database software actually made things a lot easier, as did access to the Discovery Center's servers which host far more powerful CPUs than the personal computer I utilized in Assignment 1. They were also much, much faster to execute, as shown in Table 1.

Table 1. Average Execution Time Comparisons

Implementation	Task A	Task B
Assignment 1 (50 Shards)	141.39 sec	*N/A*
MapReduce	54.76 sec	55.106 sec
Apache Hive	201.302 sec	

In Table 1, we designate Task A as the first part of the application requirements: to be able to display the top 100 most common/frequent words in the given input. Task B is the ability to take the given input and display the top 100 most common/frequent words that are greater than 6 characters in length. We shall use the same nomenclature throughout the rest of this section.

The results listed in the table are derived from the average of five execution times. Based on these average execution times, we can see that utilizing Python and MRJob to accomplish MapReduce is vastly superior to the sharding-input-on-a-single-machine approach I used in Assignment 1, even when I utilized the number of shards designated as optimal according to my analysis in Assignment 1. Similarly, the Hive implementation is almost twice as slow as both MapReduce Tasks combined, despite the fact that I am recalculating word-count pairs between Tasks, something I only do once in the Hive script. Were the two Tasks to be combined into a singular script so that they could make use of a singular computation of all word-counts in the input without having to recompute them between the tasks, MapReduce would likely be the faster approach.

Table 2. Lines of Code in Each Implementation

Implementation	Non-Task Specific	Task A	Task B
Assignment 1	38		*N/A*
MapReduce	14	9	11
Apache Hive	10	3	3

To estimate how long it would take to write each implementation from scratch (discounting all the testing and configurations and installation of packages and what-not), Table 2 displays the amount of lines of code I wrote for each implementation. These lines don't include things like program comments nor User Interface programming, but do include all the library imports and function names and declarations utilized in each implementation to accomplish both Tasks. From this, we can see that while Apache Hive is shown in Table 1 to be quite slow in comparison to standard MapReduce when it comes to accomplishing both tasks, it required far fewer lines and was thus much easier to write quickly. Assuming someone was equally proficient in Python as well as in HQL, the HQL code would be quicker to write and likely easier to use if one wished to develop this application rapidly.

Conclusions

As shown in my Data Analysis section, utilization of MapReduce produced results for finding the Top 100 Words in the 400MB Pimpernel dataset around 86-87 seconds faster on average than my implementation in Assignment 1 could at its most optimal configuration. That is an improvement of 61.3%! It also accomplished this in 60% the amount of code that my Assignment 1 implementation required. By this, I can state that the utilization of MapReduce was much better than my original implementation for Top K Words in Assignment 1.

However, I found that Hive, while far shorter in terms of the amount of code required, actually took longer to execute both tasks on average in comparison to MapReduce. This was likely due to the fact that Apache Hive's HQL operates on databases rather than on text files, and excels when dealing with strctured data. As the given input for the 400MB testing file was not a structured table of data stored in memory, but actually a text file containing The Scarlet Pimpernel, a novel, the HQL script was less efficient in parsing, sorting and counting all the data within in comparison to my MapReduce program.

Still, I would also note that when it came to "ease of use", I struggled with both Map Reduce as well as Hive programming. I ended up utilizing the MRJob package to make my MapReduce programming simpler as well, which saved me a great deal of time in programming MapReduce while also simplifying my need to deal with being careful about providing separate scripts for mapping, combining, reducing, and streaming all the data together through the cluster as the MRJob package could handle dealing with that quickly and efficiently.

Programming in Hive, on the other hand, did not require the importing of a non-native library, but did present me with an altogether different challenge: I have never coded in any SQL-based language before. Using Hive and HQL for the first time was bizarre and confusing as I had no grounding in the language and needed to not only learn it from scratch but also figure out how to program the script to do what I wanted to do. I ended up spending a lot of time just poring through the documentation for Hive and searching online for simpler explanations of how to use HQL in order to write the HQL script.

Thus, while utilizing Hive and HQL would likely be the far faster and simpler method to go about solving problems like this in the future, I find that my first tendency would be to resort to MapReduce as I have far more experience writing Python code and could visualize how to implement an algorithm far more easily in Python than I could in HQL. Still, with practice comes experience, so I cannot say that if I was given much more time to become familiar with HQL and to use it more often, that this opinion might change in favor of Hive.

Even so, MapReduce still produced results faster than Hive did, which I can attribute not only to the efficacy of utilizing MapReduce's programming model for problem solving, but also to the fact that Hive is not suited to unstructured data inputs like The Scarlet Pimpernel dataset (a novel), while Python's Regular Expression library is quite adept at parsing such data.