Querying Wikipedia with Hadoop

Kelley Lynch, Ti Liang, Hao Wang

1.Summary

In our system, we first create a lemma index and an inverted index by using two Mapreduce programs. Based on the inverted index we created, we implement another Mapreduce program to query the Wikipedia dump file and find the positions of each query word. We used HTML, PHP, and Python to build a Web GUI to visualize our results.

2.Implementation

2.1 Lemma Index & Inverted Index

We built a lemma index using Hadoop and the Cloud9 library. The Cloud9 library includes a Wikipedia Page Input Format class, which allowed us to use wikipedia pages as values to a mapreduce job.

To build the lemma index with word positions, we tokenized the wikipedia page content during the map task. For each token in the page we found locations of the token in the raw content. The index of the first character for each occurrence was added to a data structure that implements the Writable Interface called “StringIntegerArray”. As an example, if we were creating a String Integer Array for the word “running” and the document “He is running,” we would store the integer 6, because “running” begins at index 6.

StringIntegerArray provided a high level data structure containing a string and a list of integers. We also added support for reading from and writing to a string with the format: <string#list of integers>. This format allowed us to easily convert the original index to an inverted index. Each line of the original index had the format:

“documentID <word1#offset1, offset2, offset3>,<word2#offset… >”

To create the inverted index, we used mapreduce to read each line and output words as keys, and StringIntegerArrays as values. The value StringIntegerArray has the document ID as its string. The resulting inverted index format was:

“word1 <documentID1#offset1,offset2>,<documentID2#offset…>”

2.2 Elimination of Stop and Scrub Words

We selected the words that have highest word frequency as stop words, and the words that have highest document frequency as scrub words.

To eliminate stop words and scrub words, we modified the inverted index Mapreduce program to get the word frequency and document frequency for each word. With this, we ranked both of them and create a list of the 50 most frequent words given each metric. We processed our inverted index again to partition it alphabetically and during the partitioning we used the word list to filter the index by removing stop and scrub words.

2.3 Inverted Index Partitioning

Since the inverted index is very large, it would take a couple of minutes to query a word using Hadoop. To reduce the amount of time it takes, we partitioned the inverted index alphabetically. We divided the inverted index into 27 parts (A~Z, A and a are the same, and non-English characters). In the first step of querying, we only need to search the file corresponding to the first letter of the query word. For example, if we searched for the word “dog”, we would only need to run a mapreduce job on the inverted index file containing words beginning with “d,” we found that this level of partitioning performed adequately. However, if we were to increase the total size of the inverted index, it may be beneficial to further partition the inverted index. For instance, by dividing each letter’s index into further partitions, and using the first two letters of a word to determine the appropriate partition to search.

2.4 Querying with Hadoop

Querying involves two steps. In the first step, a mapreduce job filters the inverted index. The result is much smaller index with only the relevant words. In the second step, the lines of the file generated by the first step are added to a dictionary in which the key is a word in the query and the value is a set of document ids corresponding to documents that the word appears in. Next, the query is processed recursively. As long as the query contains parentheses, the query within the parentheses is processed, the result of the subquery is added to the dictionary, and the subquery portion of the original query is replaced with token representing that subquery’s intermediate result. For example, when processing the query:

“chocolate or (vanilla and (strawberry or not butterscotch))”

First the subquery “vanilla and (strawberry or not butterscotch)” would be processed. This would require the subquery “strawberry or not butterscotch” to be processed. The words “strawberry” and “butterscotch” would be in the dictionary and the documents they occur in would be stored in a set. This would allow us to use set operations to produce the intermediate query result. The result would be stored in the dictionary with the key being “strawberry_or_not_butterscotch”. The query would then be updated to be:

“chocolate or (vanilla and strawberry_or_not_butterscotch)”

The 2nd recursive call to the query processor would terminate and now the 1st recursive call could be completed. The subquery “vanilla and strawberry_or_not_butterscotch” could be processed left to right, since all of the necessary results have been stored in the dictionary. The 1st recursive call would complete having added the result of the subquery “vanilla and strawberry_or_not_butterscotch” to the dictionary. The final query “chocolate or vanilla_and_strawberry_or_not_butterscotch” can now be processed.

We assume that at least one word in the query will not be negated. Negated words and phrases are handled in the context of the documents in which the query words appear. For example, querying “not strawberry and not banana” will not return all of the wikipedia articles that contain neither strawberry nor banana. It will return no documents. The maximum set of documents returned for a query is the union of the documents for each of the words in the query.

After constructing a list of documents for a given query, we use a forward index to allow random access to a compressed version of the Wikipedia dump file. This functionality was included in the Cloud9 toolkit. This significantly increased our query response time. Without the ability to randomly access documents given their document IDs, we would have to run another mapreduce job on the Wikipedia dump file to get the relevant documents for a query.

2.5 Result ranking

We rank a query’s results by counting the number of query words in each document. For example, when we query “dog”, if “dog” appears 100 times in doc1, and 200 times in doc2, doc2 will be ranked higher than doc1 and will be displayed above doc1 in the Web GUI.

2.6 Web GUI

In the front-end design, we use bootstrap to design the display of the webpage.

In the server, we use PHP and Python. Python is activated by a PHP command. A Python script connects to the cluster via ssh and runs the command to perform each component of querying. After the mapreduce job finishes and the query has been processed, python will print all the output into a local file called output.txt.

The main functions of the webpages are implemented in PHP. PHP will extract and process the output from the local output file. This includes highlighting the keywords using the position values from the queries’ results, displaying the abstract of relevant articles in the main page, and reading the whole articles after clicking the button for a particular article.

We used PHP to read the output file and highlight the query word. The output file would be in this format:

<title>…</title>
<content>…</content>
<position><word>String1#position1,…><word>String2#position1…></position>

We store title, content and position into arrays, and substring the content based on the position. For example: if the word “dog” is in position 300, and length of “dog” is 3. We take a substring of length 3 beginning at the offset position. In this example we would divide the string into 3 substrings, [0,299],[300,302],[303,end] and highlight the second substring.

When we began writing the web interface, we struggled to connect to the cluster with PHP. We solved this problem by using the Python “pexpect” module. This module allowed us to send commands to the cluster and receive the response. There were further issues with Python version compatibility with the pexpect module. We were able to solve this problem by using the correct version of Python for pexpect module.

3.Experiment

3.1 Size of Output

The inverted index is around 2.5 GB, after removing the stop words and scrub words it is 1.9 GB. After partitioning, each file does not exceed 200MB.

3.2 Running Time

Totally	63s
Finding words from inverted index partition	13s(70s without partition)
Processing query and do random access	20s
Copy file and Mapreduce setup	Over 30s

It takes about a minute to process a query. The number of query words has little impact on running time. The main reason for the slow response time is the amount of time it takes to set up a mapreduce job. Given the size of our inverted index, it may be faster to process queries without using mapreduce however, using mapreduce makes the system more scalable.

3.3 Comments

This project allowed us to gain experience working with Hadoop and working on a cluster remotely. One difficulty we had was dealing with version issues between Cloud9 and Hadoop. The versions installed on the cluster were incompatible in a way that prevented us from easily creating the files for random access. This problem was solved by downloading the Cloud9 source code and modifying the pom file to create a jar that was compatible with the version of Hadoop on the cluster. We also had problems with character encoding between Java on the cluster, the Python server code, and the PHP front end. This made it difficult to correctly highlight words in the query results.

One assumption that we made when building the system to process queries, was that the only part of the query processing that would require parallelization is retrieving the document lists from the inverted index. This assumption is fine for our task with the size of our input, however for a much larger task, this system may be insufficient. A larger task may benefit from processing different components of the query in parallel. For example, given “(dog or cat) and (strawberry or banana)”, it is possible to process the 2 subqueries in parallel. We also relied on being able to store in memory the lists of documents that query words appear in. For a much larger task this may not be possible. In order to make our system more scalable, we could process the queries with the files on hdfs.