Niels Joubert CS162-PC
Nadeem Laiwala CS162-PD
Akhil Dhar CS162-KK
Eugene Li CS162-KI
William Wu CS162-KJ
April 27, 2008
Computer Science 162

CS162 Design Doc Phase 4: Building a Full Inverted Index

Task I: Running the Line Indexer MapReduce Job

This tasks attempts to introduce us to the idea of distributed computing by allowing us to learn the basics of executing a MapReduce-based algorithm (Line Indexer) on a single *Shakespeare Corpus* file.

Correctness Constraints

- The output file must match the expected behavior. Specifically, each word (key) must be associated with the <line #, line snippet> values where it appears in the *Shakespeare Corpus* file.
- MapReduce significantly decreases the execution time of the Line Indexer program in comparison to the execution time of the serial version of the Line Indexer program.
- Multiple executions of the code on different machines produce consistent output (robustness).

Implementation

```
Load LineIndexer code (Driver containing main method, Mapper, and Reducer)
into Eclipse package
Export the 3 classes to a JAR file and move the JAR file onto a Hadoop VM node
Copy the Shakespeare Corpus file into an "input" directory on the Hadoop DFS
Run the JAR file on Hadoop with the Shakespeare Corpus file as input
Once MapReduce has executed the job, copy back the output file from the VM
for testing
```

Test

- Compare output files generated by group members on different machines using the UNIX diff command.
- Randomly choose ten keys (words) and verify their values (line #, line snippet) in the *Shakespeare Corpus* file.

Building the Inverted Book Index

Task II: Scrubbing the input

For the best results in searching through a corpus of data, we need to support filtering the words so that we can get closer to finding their semantic meaning. This is opposite the idea of matching words on a purely syntactic basis. To support this, we filter all words through a set of rules to collapse words with the same semantic meaning into one keyword.

Correctness Constraints

- The output must be a more concise version of the input after the keyword filters are applied.
- Keywords must be grouped by semantic meaning even if their syntax differs.
- Syntactically different keywords can be collapsed into one keyword provided that their semantic meaning is the same, decreasing our corpus size.
- Keywords with different semantics must remain as different keywords.
- Scrubbing must improve the quality of our results.

Implementation

During the map phase of building our inverted book index, all words are handed to the KeywordFilter. The Filter will apply the necessary transformations to a word to collapse it into its common semantical meaning. We scrub the text to handle the following type of semantical inconsistencies:

- Capitalization differences - remove all capitalization
- Punctuation around words - remove all trailing and leading punctuation
- Leave only alphanumeric characters

It would be interesting to try and map singular and plural words to each other, but since the amount of corner cases is exceedingly large, it would be wiser to implement that at a later date after we have a working system. We implement a KeywordFilter class that exposes a static filter method. This method takes in a word, filters it according to the above rules (and the rules in part 3) and emits a valid keyword. The Mapper will run this filter on each word before it saves it as the key in the (key, value) pairs it creates.

```
class KeywordFilter

    static String filter (string input) {
        downcapitalize input
        strip off trailing and leading punctuation
        remove all non-alphanumeric characters
        return new keyword
    }
```

Testing

The best way to test this part of the project is to throw many different syntactical representations of the same word at my filter, and assert that they all map to the same output.

<u>Example Input:</u>
Buffalo

Buffalo,
Buffalo-
buffalo
buffalo...
"Buffalo!"
BUFFALO

Expected output:
buffalo

Task III: Removing Stop Words

This task forms a logical continuation of the filtering we did for Task 2. As part of the same KeywordFilter that we described for Task 2, a filter is applied to remove the most frequently occurring words in an attempt to remove this noise that obfuscates the more interesting parts of the document. This task has two parts: identify which words are noise words, and modify our filter from Task 2 to remove these words.

Correctness Constraints

- Noisy words should be calculated by analyzing a representative sample of text.
- Words should be classified as noise if they are significantly more frequent than the average occurrence of word.
- The final inverted index that we create must not contain the words identified as noise.

Defining stop words

To define a list of stop words, we first need a comprehensive list of the most frequently occurring words in the corpus we will be working with. We can use Shakespeare's assorted works as input to the distributed Word Count algorithm given to us. This will supply us with a lexically sorted list that maps each word to its respective word count.

From this word count list, we will create a list of words that appear most often overall: a stop word list. To do this, we first read in the entire word count list and store each (word, count) element in a heap sorted by largest elements. Then to define the boundary between stop words and queryable words, first we find the mean and the standard deviation of the count of the words. Then we say that stop words are all words whose count is greater than the mean count plus twice the standard deviation. We call this number the upper limit for queryable words. For example, if the mean count for all words was 20 and the standard deviation was 100, then any word whose count is greater than 220 is a stop word. Then our only task is to retrieve out of the heap all words whose count is greater than the previously defined upper limit. We can perform all this in a separate script.

Using stop words in the KeywordFilter

The stop words file is residently stored in the distributed file system used by Hadoop. When building the inverted index, the KeywordFilter class will store a single copy of this stop words list as a static HashSet containing String objects. This will allow us to search for the existence of a word in constant time. The HashSet gets initialized when a Mapper tries to use the Filter for the first time. We specify the location of the stop word file on the Hadoop DFS by passing in its path as an argument to the JAR that creates the index.

As part of the filtering task described in Part 2, we check each of the scrubbed keywords word to see if it exists in the stop words list. If it is present in the list, we return null so that the Mapper knows not to include the word in the final index. Thus, the Mapper will call the filter method, and get back either a normalized version of the word, or null if the word is a stop word.

```
class KeywordFilter

    static HashSet<String> stopWords
    static boolean initialized = false

    static void initialize(Path stopPath)
        if stopPath is null
            return
        if stopWords is not null
            return
        stopWords = new HashSet<String>
        create a BufferedReader called readStopWords that reads from stopPath
        for every line in the stopwords file
             load stop word into stopWords
        set initialized to true

    static String filter (String input, Path stopPath)
        if not initialized
            call initialize method, passing in stopPath
        make a new StringBuilder called outputBuilder
        for each character in input
            if character is alphanumeric
                add the lowercase version of the char to outputBuilder
        if outputBuilder has length of zero
            return ""
        make a String called output from outputBuilder
        if stopWords is null or if stopWords doesn't contain output:
            return output
        return ""
```

Test

We can run the script to find stop words on some word count files for which the distribution of words is known. Then we can inspect the results to see that they match our expectations.
We can also create a word count file for which the distribution of words is uniform to verify that no words are identified as stop words.

Task IV: Creating a Full Inverted Index

This task asks us to design a MapReduce-based algorithm to calculate the inverted index over the Project Gutenburg corpus of Shakespeare's works.

Correctness Constraints

- The final inverted file index should not contain any stop words identified in Step 2
- The format of the MapReduce output must be simple enough to be machine-parsable
- The output will have a mapping from a word to every document

Implementation

We define the MakeIndexMapper class as follows:

```
class MakeIndexMapper extends MapReduceBase implements Mapper {
   Text word
   Text summary
   String inputFile

   public void configure(JobConf job) {
      set inputFile to the file associated with the current job
   }

   public void map(WritableComparable key, Writable values, OutputCollector
output, Reporter reporter) throws IOException     {
      change values to a string, and set it to line
      create a tokenizer for line and set it to itr

      while(itr has more tokens) {
         scrub the next token from itr, and set it into word
         if (word is null)
            continue
         else
            set into summary the file name, followed by a colon, the key,
another colon, and then the line
            output collects (word, summary)
      }
   }
}
```

The "key" variable must be a String that we can interpret as the rough location of a word in a document. The "values" variable must be a String that we interpret as the line (text snippet) specified by the key. In the mapping phase, we first change values into a string. Then for every word in values, we scrub the word using the scrubber from Part 2. Then we associate each scrubbed word with a "summary" value. The value of summary should be a Text concatenation of the document identifier and the line snippet. We then let "output" collect each of these <word, summary> pairs to produce our intermediate results.

We now define the MakeIndexReducer class as follows:

```
class MakeIndexReducer extends MapReduceBase implements Reducer {

   public void reduce(WritableComparable key, Iterator values, OutputCollector
output, Reporter reporter) throws IOException {
        set boolean firstInstance to true
        create StringBuilder object called toReturn

        while(values next value is not null){
           if(firstInstance is not true)
              append a '^' to the toReturn object
           set firstInstance to false
           append the next item in values into toReturn
        }
        output collects (key, the string toReturn)
     }
  }
```

The "key" variable in reduce will be a single scrubbed word. The intermediate phase will gather all intermediate results having a distinct key and aggregate their "summary" values into an iterator called "values". Then the job of the Reducer is to create an overall "result" value which is a combination of all the tokens inside of the iterator. For the purposes of building an inverted index, we can simply walk through the iterator and concatenate all the "summary" values together separated by a unique separator, which we will call "^".

For example, if the key was the word "arch" and the values in the iterator were {"Doc10, The arch was big", "Doc50, Oh my what a big arch", "Doc95, The arch is fun"}, then the output of the reduce method would be: ("arch", "Doc10, The arch was big^Doc50, Oh my what a big arch^Doc95, The arch is fun").

To index the full Project Gutenburg corpus, we need to store the corpus on the Hadoop DFS and submit the job to a JAR file that contains a Driver, the InvertedIndexMapper, and the InvertedIndexReducer. Then we will be returned with an output file that contains the results of all the runs of reduce.

Test

We will create a few smaller documents that we can create a full inverted index ourselves. We can then run our algorithm to obtain the full inverted index and then compare the results.

Task V: Querying the Full Inverted Index

This task asks us to design a MapReduce-based algorithm that allows us to query the inverted index generated in Task IV.

Correctness Constraints

- Accept a user-specified query and respond with the document ID and text snippet where the query word is found.
- Query program must support "and", "or", and "not" operations.
- Utilize the MapReduce model in order to achieve execution time gains.
- Map should return all the lines that a given keyword is found on as keys.
- The Barrier should unify all keys that are equal and combine their values into a value set.

- The Reducer should remove all keys that has a value set not conforming to the query.

Implementation

A query consists of keywords and modifiers. Keywords are searched for in the text itself during the map stage. Modifiers change how the results from map are reduced into results.

The Mapper takes in (key-value) pairs that are entries in our InvertedIndex. It receives the query from a shared variable we save in a distributed cache - this query will not change for the course of this MapReduce execution. If the key matches any of the keywords in the query, each value in the inverted index (thus, each line this keyword appears on) is created as a output key, with the value of the keyword that we just found. Thus, we flipped input values to become output keys.

At the barrier stage, the values of all the equivalent keys will get unified from all the mappers. Thus, if you query for "WORD1 AND WORD2 AND WORD3", at the barrier stage you will have a key for each line where any one of the WORDs appear, and the values of these keys will be unified so that one line as key will map to all the words as value that appear in that line.

The QueryMapper class is defined as follows:

```
class QueryMapper extends MapReduceBase implements Mapper {
    Path queryPath;
    String keywords;

    public void configure(JobConf job) {
        try {
            obtain the actual query path from the distributed cache
        } catch (IOException e) {
             error if this couldn't happen
        }
        try {
            create a BufferedReader object called readQuery using the queryPath
            set keywords to be the query text changed to lower case
            replace all ands, ors, and nots with spaces
            create a queryitr tokenizer with this keywords string

            while(queryitr has more tokens) {
                create a string called keyword, and set it to the next token in
queryitr
                in keywords, replace all this particular keyword with a scrubbed
version of itself
            }
        } catch (IOException e) {
            error occured while reading query
        }
    }

    public void map(WritableComparable key, Writable values, OutputCollector
output, Reporter reporter) throws IOException {
        convert values to a string and set it to variable called line
        create tokenizer keyworditr from the keywords string
        create tokenizer keyitr from the line string
        create string word from the next token in keyitr

        while(keyworditr has more tokens) {
            set the string keyword to the next token in keyworditr

            if (word matches with keyword){
                create the string allValues and set it equal to the line with the word
cut off
                create a tokenizer valueitr from the allValues string with the ^
delimiter

                while(valueitr has more tokens) {
                    create the string value and set it to the next token in valueitr
                    create a Text object out of value, create a Text object of word
                    output collect (value, word)
                }
                break;
            }
        }
    }
}
```

During the reduce stage, each key is judged according to its list of values on whether it should remain in the final result set. The set of values is examined according to the query's modifiers. If the set of values conform to the query, the line is emitted as a key-value pair, with the line number as the key and the line's contents as the value.

The boolean logic for the reduce stage is as follows:

AND: The value set in the reduce stage must contain BOTH keywords referenced in the AND statement.
OR: The value set in the reduce stage must contain EITHER of the keywords referenced in the OR statement.
NOT: The value set in the reduce state must not contain the keyword references in the NOT statement.

We accomplish this by running it through a simple parser, moving down the list of modifiers in the query, testing each one against the value set.

We now define the QueryReducer class:

```
class QueryReducer extends MapReduceBase implements Reducer {
    private Path queryPath
    private String query
    private String allValues

    public void configure(JobConf job) {
        try {
            obtain queryPath from distribute cache
        } catch (IOException e) {
            error
        }
        try {
            create bufferedReader readQuery with the queryPath
            set String query to lower-case version of the query
            create tokenizer queryitr with the query string

            while(queryitr has more tokens) {
                set keyword string to the next token of queryitr
                if (the keyword is not "and," "or," "not")
                    replace the keyword in the query with a scrubbed version of the
word
            }
        } catch (IOException e) {
            error while reading query
        }
    }

    public void reduce(WritableComparable key, Iterator values, OutputCollector
output, Reporter reporter) throws IOException {
        create boolean called first, set it to true
        create stringbuilder object toReturn

        while(values has a next value){
            if(first is false)
                append a space to toReturn
            set first to false
            append to toReturn the string value of the next item in values
        }
        set allValues to string version of toReturn
        create tokenizer validqueryitr with query
        if (validQuery(validqueryitr) returns false)
            return
        create tokenizer checkqueryitr with query
        if (checkQuery(checkqueryitr, false)) {
            output collect key with Text object of empty string
        }
    }
```

Note that before output does its collect function, we must check to see if the query is valid and to also check against the query. A valid query is defined by various rules. 1) The query cannot start with a "not" term, and cannot end with "and," "or," or "not." 2) The query cannot have a "and" or "not" following another modifier. 3) The query cannot have "not" or a keyword following a keyword.

```
private boolean validQuery(StringTokenizer remainingQuery) {
    while(remainingQuery has more tokens) {
        set string nextElement to the next token in remainingQuery
        if (nextElement equals "not")
```

```
            if (remainingQuery has more tokens)
                continue
            else
                return false
        else if (nextElement equals "and" OR nextElement equals "or")
            return false
        else {
            if (remainingQuery has more tokens) {
                set string followingElement to the next token in remainingQuery
                 if (followingElement does not equal "and" AND does not equal "or"
AND does not equal "not")
                    return false
                if (followingElement equals "not"))
                    return false
                if (remainingQuery does not have more tokens &&
                   (followingElement equals "and" OR followingElement equals "or"))
                    return false
            }
        }
    }
    return true
}
```

The checkQuery function checks to see if the results from the reducer adheres to the query restrictions. The results from the allValues string are checked in a recursive manner by whittling down the query as specific keywords are hit. The popping of next elements in the query iterator guarantees that we'll reach the base case of this recursive function.

```
private boolean checkQuery(StringTokenizer remainingQuery, boolean notFlag)
{
        create boolean called prevItem, set it to false
        if remainingQuery does not have more tokens
            return true
        create string nextElement, and set it to the next token in remainingQuery

        if (nextElement equals "and" or equals "or")
            return false;
        else if (nextElement equals "not")
            return checkQuery(remainingQuery, !notFlag);
        else {
            create tokenizer valueitr from the string allValues
            while(valueitr has more tokens)
                if (next token in valueitr equals nextElement)
                    prevItem = true
                    break
            if (remainingQuery does not have more tokens)
                return prevItem
            set nextElement to next token in remainingQuery

            if (nextElement equals "and")
                return ((!notFlag and prevItem) OR (notFlag and !prevItem))
```

```
                    AND  checkQuery(remainingQuery, false)
            else if (nextElement equals "or"))
               return ((!notFlag and prevItem) OR (notFlag and !prevItem))
                    OR checkQuery(remainingQuery, false)
            else
               return false
         }
      }
   }
```

The final result will return a list of all the lines satisfying the query.

Test

- Submit a query for a rare word without any boolean modifiers and check correctness of output
- Submit a query for two words with the AND modifier and check correctness of output
- Submit a query for two words with the OR modifier and check correctness of output
- Submit a query for a word with the NOT modifier and check correctness of output
- Submit a query for two words with the AND modifier, one word having an additional NOT modifier, and check correctness of output
- Write a quick procedure that performs the Query program utilizing a serial model rather than MapReduce Model to assess the execution time gains

Design Questions (to be answered)

1. How long did you originally think this project would take you to finish?
40 hours

2. How much time did you actually spend on this project?
45 hours

3. What, if any, "stop words" did you find as a result of your WordCount? By what criteria did you determine that they were "noisy?" Were you surprised by any of these words?
We determined stop words to be all words occurring more then twice the standard deviation above the average word frequency. As part of our Make Index script, we use the Word Count MapReduce task to count the number of occurrences for each word and build a stopwords file. I was surprised to find "Enter", "Lord" and "God" as some of the most used words in Shakespeare!