Big Data Processing & Analytics – 2nd Assignment
String-similarity joins

Yijing RONG

I Pre-processing the input (10)

In order to reduce the executive time and test the code more quickly, we suppose to take the first 200 lines of the file pg100.txt instead of the full text file. If necessary, we can still run the code on the original file later and get the results. The sample input file here is named as *pg100_head.txt*.

- **Remove all stopwords [(1)] (you can use the stopwords file of your previous assignment), special characters [(2)] (keep only [a-z], [A-Z] and [0-9]) and keep each unique word only once per line [(4)]. Don't keep empty lines [(3)]. (2)**

(1) Remove stopwords using the previous file

The stopwords file according to the previous assignment is *Stopwords.txt*.
In assignment 1, we loaded the stopwords file in the java directly, this time we tried to load the stopwords file directly from the command line by putting the *Stopwords.txt* in HDFS. Thus, we need to specify the argument "-pass" + HDFS path of *Stopwords.txt* to the Hadoop command:

```
hadoop jar assignment2.jar yijing.assignment2.pre-processing
input/pg100_head.txt output -pass input/Stopwords.txt
```

jar: assignment2.jar
main class: yijing.assignment2

Add some configurations in the driver to define the "-pass" method:

```
// additional configurations to handle the "-pass" method
for (int i = 0; i < args.length; i += 1) {
    if ("-pass".equals(args[i])) {
        job.getConfiguration().setBoolean(
                "pre-processing.pass.patterns", true);
        i += 1;
        job.addCacheFile(new Path(args[i]).toUri());
        LOG.info("Added file to the distributed cache: " + args[i]);
    }
}
```

Add “loadfile” and “parsepassfile” methods in the map phase:
“loadfile” method: load the Stopwords.txt file
“parsefile” method: parse the file

```
public static class Map extends
        Mapper<LongWritable, Text, LongWritable, Text> {
    private Set<String> patternsToRemove = new HashSet<String>();
    private BufferedReader fileinput;


    // "loadfile" method to load the stopwords file
    protected void loadfile(Mapper.Context context) throws IOException,
            InterruptedException {
        if (context.getInputSplit() instanceof FileSplit) {
            ((FileSplit) context.getInputSplit()).getPath().toString();
        } else {
            context.getInputSplit().toString();
        }
        Configuration cfgr = context.getConfiguration();
        if (cfgr.getBoolean("pre-processing.pass.patterns", false)) {
            URI[] localPaths = context.getCacheFiles();
            parsefile(localPaths[0]);
        }
    }

    // "parsefile" method to parse the file
    private void parsefile(URI patternsURI) {
        LOG.info("Added file to the distributed cache: " + patternsURI);
        try {
            fileinput = new BufferedReader(new FileReader(new File(
                    patternsURI.getPath()).getName()));
            String pattern;
            while ((pattern = fileinput.readLine()) != null) {
                patternsToRemove.add(pattern);
            }
        } catch (IOException ioe) {
            System.err
                    .println("Exception while parsing the cached file '"
                            + patternsURI
                            + "' : "
                            + StringUtils.stringifyException(ioe));
        }
    }
```

Since the stopwords list is now available, we apply a method in an “if loop” to remove the stopwords from the input file:
`patternsToRemove.contains(word.toLowerCase())`

(2) Remove special characters (keep only [a-z], [A-Z] and [0-9])

In the mapping phase, we use a regex pattern to filter away any strings that don’t have A-z and/or 0-9:
`Pattern patt = Pattern.compile("[^A-Za-z0-9]");`

Still in the “if loop”, we add two conditions to test if the string contains such pattern

and to make sure that the special characters and empty words are removed properly:
`patt.matcher(word.toLowerCase()).find())`
`word.toLowerCase().isEmpty()`

(3) Remove empty lines

Since the concept of "empty lines" is equivalent to the length of such line is zero, we add another condition to the "if loop" to make sure that the empty lines are also exactly removed:
`(mytext).toString().length() == 0`

```
public void map(LongWritable key, Text mytext, Context mycontext)
        throws IOException, InterruptedException {

    for (String word : mytext.toString().split("\\s*\\b\\s*")) {

        // filter away any strings that don't have [a-z],[A-Z]and[0-9]
        Pattern patt = Pattern.compile("[^A-Za-z0-9]");

        if (mytext).toString().length() == 0
        // condition of empty lines

                // remove the stopwords list
                || patternsToRemove.contains(word.toLowerCase())

                // ensure the empty words and special characters are removed
                || word.toLowerCase().isEmpty()
                || patt.matcher(word.toLowerCase()).find()) {

            continue;
        }

        mycontext.write(key, new Text(word.toLowerCase()));
    }
}
```

(4) Keep each unique word only once per line

We pick the corresponding list of words (*wordlist*) of each line and put them in a Hashset (*hashsets*) to keep each unique word only once per line:

```
// Keep each unique word only once per line
ArrayList<String> wordlist = new ArrayList<String>();

for (Text word : mytexts) {
    wordlist.add(word.toString());
}
```

- **Store on HDFS the number of output records (i.e., total lines). (1)**

First, we need to create a specific counter to count the total number of output lines. We define an enum (*MYCOUNTER*) and the counter we want to use (`totlines`):

```
// Define a custom counter to count the number of output records
public static enum MYCOUNTER {
    totlines,
};
```

Then, we use the following code to store this counter in HDFS until the job is compete:

```
// Store the counter in HDFS until the job is complete
job.waitForCompletion(true);
long mycounter = job.getCounters().findCounter(MYCOUNTER.totlines)
        .getValue();
Path outFile = new Path("totlines.txt");
BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(
        fs.create(outFile, true)));
bw.write(String.valueOf(mycounter));
bw.close();
return 0;
```

In the end of the Reduce phase (before writing the output into HDFS), we get the desired counter (*totlines*) via *getCounter* method and call the *increment* method as follows to increment this counter by 1:

```
// increment the counter by 1
mycontext.getCounter(MYCOUNTER.totlines).increment(1);
```

The output *totlines* value in HDFS is 149.

Next, we would like to add a global frequency to each words of each line. An efficient way to realize this is to use the output (*wordcounting.txt*) that count the frequency of each word in our pre-processing program. The output is got by a basic wordcount program (*wordcounting.java*).

In the Reduce phase, we need to load the *wordcounting.txt* file and then create an HashMap, which is a Hash table basically. Using HashMap enables us to have the frequency of a specific word simply by calling *wordcount.get(word)* method. Thus, we could add the global frequency of each word that counted in the wordcounting program.

```
// load the wordcounting file which have the global frequency of each word
HashMap<String, String> wordcounting = new HashMap<String, String>();
reader = new BufferedReader(
        new FileReader(
                new File(
                        "/home/cloudera/workspace/assignment2/output/
                            wordcounting.txt")));
String pattern;
while ((pattern = reader.readLine()) != null) {
    String[] word = pattern.split(",");
    wordcounting.put(word[0], word[1]);
}

HashSet<String> hashsets = new HashSet<String>(wordlist);

StringBuilder wordstringbuilder = new StringBuilder();

// add global frequency of each word computed in the wordcounting.java
String pr_1 = "";
for (String word : hashsets) {
    wordstringbuilder.append(pr_1);
    pr_1 = ", ";
    wordstringbuilder.append(word + "#" + wordcounting.get(word));
}
```

- **Order the tokens of each line in ascending order of global frequency. (7)**

The result of the last step should seem like this for each line:
[line], [word1]#[frequency1], [word2]#[frequency2], [word3]#[frequency3]

However, they are not in a desired order. Now, we would like to sort them in ascending order by global frequency. So we need to add a sorting method in the Reduce phase. What we do is first scanning the whole word list, getting the integers just after "#" (the frequency), comparing them and then returning the word list with the proper order of these integers:

```
// sort the word list in ascending order
java.util.List<String> wordcountlist = Arrays
        .asList(wordstringbuilder.toString().split("\\s*,\\s*"));

Collections.sort(wordcountlist, new Comparator<String>() {
    public int compare(String freq1, String freq2) {
        return extractInt(freq1) - extractInt(freq2);
    }

    int extractInt(String str) {
        String nb = str.replaceAll("[^#\\d+]", "");
        nb = nb.replaceAll("\\d+#", "");
        nb = nb.replaceAll("#", "");
        return nb.isEmpty() ? 0 : Integer.parseInt(nb);
    }
});
```

- **Store them on HDFS in TextOutputFormat.**

Finally, we output the words with frequency in ascending order in *TextOutputFormat* in HDFS at the end of Reduce phase:

```
StringBuilder wordstringbuilder_asc = new StringBuilder();

String pr_2 = "";
for (String word : wordcountlist) {
    wordstringbuilder_asc.append(pr_2);
    pr_2 = ", ";
    wordstringbuilder_asc.append(word);
}

// increment the counter by 1
mycontext.getCounter(MYCOUNTER.totlines).increment(1);

mycontext.write(key, new Text(wordstringbuilder_asc.toString()));
```

II Set-similarity joins (90)

Efficiently identify all pairs of documents (d1, d2) that are similar (sim (d1, d2) >= t), given a similarity function *sim* and a similarity threshold *t*. Assume that:

- Each output line of the pre-processing job is a unique document
- Documents are represented as sets of words
- $sim\ (d1, d2) = Jaccard\ (d1, d2) = |d1 \cap d2|/|d1 \cup d2|$
- t = 0.8

a) Perform all pair-wise comparisons between documents, using the following technique: Each document is handled by **a single mapper** (remember that lines are used to represent documents in this assignment). The map method should emit, for each document, the document id along with one other document id as a **key** (one such pair for each other document in the corpus) and the document's content as a **value**. (2) In the reduce phase, perform the Jaccard computations for all/some selected pairs. (3) **Output** only similar pairs on HDFS, in TextOutputFormat. (4)
Make sure that the same pair of documents is compared **no more than one**. (1)
Report the executive time and the number of performed comparisons. (5)

For this part, we could first use the pre-processing program in the previous section, but this time, we don't need to have the global frequency of each word in the output file, only the sorted words in ascending order of frequency, so we call the *replaceAll* method when writing the output (*pre-processing_2.java*, *wordlist_asc.txt*):

```
// output file without printing the global frequency
mycontext.write(key, new Text(wordstringbuilder_asc.toString()
                              .replaceAll("#\\d+", "")));
```

(1) Make sure that the same pair of documents is compared no more than one.

The idea here is that we need to remove the same pair so that it would not be compared twice and consequently increase the executive time. Thus the mapper should emit a key (d1, d2) in a tuple format, in which the sequence of its objects doesn't matter. We could create a custom Writable class and write an implementation that represents a pair of strings, called TextPair. The implementation is shown as below: (*Hadoop: The Definitive Guide, Tom White, P.104/ Example 4-7*)

```
class TextPair implements WritableComparable<TextPair> {

    // add the pair objects as textpair fields
    private Text first;
    private Text second;


    // implement the set method that changes the pair content
    public void set(Text first, Text second) {
        this.first = first;
        this.second = second;
    }


    // implement the first getter
    public Text getFirst() {
        return first;
    }

    // implement the second getter
    public Text getSecond() {
        return second;
    }

    // implement the constructor
    public TextPair(Text first, Text second) {
        set(first, second);
    }

    public TextPair() {
        set(new Text(), new Text());
    }

    public TextPair(String first, String second) {
        set(new Text(first), new Text(second));
    }
```

```
    // write to out the serialized version of this such that can be deserialized in future
    // this will be use to write to HDFS
    @Override
    public void write(DataOutput out) throws IOException {
        first.write(out);
        second.write(out);
    }

    // read from in the serialized version of a pair and deserialize it
    @Override
    public void readFields(DataInput in) throws IOException {
        first.readFields(in);
        second.readFields(in);
    }
```

```
    // implement hash
    @Override
    public int hashCode() {
        return first.hashCode() * 163 + second.hashCode();
    }

    // implement equals
    @Override
    public boolean equals(Object o) {
        if (o instanceof TextPair) {
            TextPair tp = (TextPair) o;
            return first.equals(tp.first) && second.equals(tp.second);
        }
        return false;
    }

    // implement the comparison between this and other
    @Override
    public int compareTo(TextPair other) {
        int cmpFirstFirst = first.compareTo(other.first);
        int cmpSecondSecond = second.compareTo(other.second);
        int cmpFirstSecond = first.compareTo(other.second);
        int cmpSecondFirst = second.compareTo(other.first);

        if (cmpFirstFirst == 0 && cmpSecondSecond == 0 || cmpFirstSecond == 0
                && cmpSecondFirst == 0) {
            return 0;
        }

        Text thisSmaller;
        Text otherSmaller;

        Text thisBigger;
        Text otherBigger;

        if (this.first.compareTo(this.second) < 0) {
            thisSmaller = this.first;
            thisBigger = this.second;
        } else {
            thisSmaller = this.second;
            thisBigger = this.first;
        }

        if (other.first.compareTo(other.second) < 0) {
            otherSmaller = other.first;
            otherBigger = other.second;
        } else {
            otherSmaller = other.second;
            otherBigger = other.first;
        }

        int cmpThisSmallerOtherSmaller = thisSmaller.compareTo(otherSmaller);
        int cmpThisBiggerOtherBigger = thisBigger.compareTo(otherBigger);

        if (cmpThisSmallerOtherSmaller == 0) {
            return cmpThisBiggerOtherBigger;
        } else {
            return cmpThisSmallerOtherSmaller;
        }
    }


    // implement toString for text output format
    @Override
    public String toString() {
        return first + "\t" + second;
    }
}
```

(2) The map method should emit, for each document, the document id along with one other document id as a key and the document's content as a value.

In the Map phase, we load the pre-processed file without the frequency number (*wordlist_asc.txt*) and add a new HashMap object to define the document id as key

element and the wordlist as the value element:

```
public static class Map extends Mapper<Text, Text, TextPair, Text> {

    private BufferedReader reader;
    private static TextPair textPair = new TextPair();

    // load the wordcount list (without frequency)
    @Override
    public void map(Text key, Text value, Context context)
            throws IOException, InterruptedException {

        HashMap<String, String> lineshashmap = new HashMap<String, String>();
        reader = new BufferedReader(
                new FileReader(
                        new File(
                                "/home/cloudera/workspace/assignment2/output/
                                    wordlist_asc.txt")));
        String pattern;
        while ((pattern = reader.readLine()) != null) {
            String[] line = pattern.split(",");
            lineshashmap.put(line[0], line[1]);
        }
```

Then, we use the TextPair class to emit the document id along with one another as the key element, and the document's content as the value element:

```
// document id as key element, document content as value
for (String line : lineshashmap.keySet()) {
    if (key.toString().equals(line)) {
        continue;
    }

    textPair.set(key, new Text(line));
    context.write(textPair, new Text(value.toString()));
}
```

(3) In the reduce phase, perform the Jaccard computations for all/some selected pairs.

Define a function to calculate the jaccard similarity (intersection / union):

```
// compute the jaccard similarity
public double similarity (TreeSet<String> str1, TreeSet<String> str2) {
    if (str1.size() < str2.size()) {
        TreeSet<String> s1bis = str1;
        s1bis.retainAll(str2);
        int intersection = s1bis.size();
        str1.addAll(str2);
        int union = str1.size();
        return (double) intersection / union;
    } else {
        TreeSet<String> s1bis = str2;
        s1bis.retainAll(str1);
        int intersection = s1bis.size();
        str2.addAll(str1);
        int union = str2.size();
        return (double) intersection / union;
    }
```

In the Reduce phase, we load the word list (without frequency) as before into an HashMap object:

```
@Override
public void reduce(TextPair key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

    // load the word list (without frequency) into a HashMap
    HashMap<String, String> lineshashmap = new HashMap<String, String>();
    reader = new BufferedReader(
            new FileReader(
                    new File(
                            "/home/cloudera/workspace/assignment2/output/
                                wordlist_asc.txt")));
    String pattern;
    while ((pattern = reader.readLine()) != null) {
        String[] line = pattern.split(",");
        lineshashmap.put(line[0], line[1]);
    }
```

Then, we create a first set that take the wordlist of the second document (doc2) as values in the pair of (doc1, doc2):

```
// take as values the words list of doc2 in the pair of (doc1, doc2)
TreeSet<String> words_2_treeset = new TreeSet<String>();
String words_2_string = lineshashmap.get(key.getSecond()
        .toString());
for (String word : words_2_string.split(" ")) {
    words_2_treeset.add(word);
}
```

A second set which takes the wordlist of the first document as values in the pair of (doc1, doc2):

```
// take as values the words list of doc1 in the pair of (doc1, doc2)
TreeSet<String> words_1_treeset = new TreeSet<String>();

for (String word : values.iterator().next().toString().split(" ")) {
    words_1_treeset.add(word);
}
```

Thus, we could implement the similarity function to compute the jaccard similarity between these two sets and increment the number of comparisons by 1 as well:

```
// increment the counter by 1
context.getCounter(MYCOUNTER.totcompare_a).increment(1);

// compute the jaccard similarity between 2 sets
double sim = similarity(words_1_treeset, words_2_treeset);
```

(4) Output only similar pairs on HDFS, in TextOutputFormat.

In this exercise, we assume the threshold t = 0.8, thus we add this condition to the if loop and output the pairs that accord with this restriction:

```
// Assuming threshold t = 0.8
if (sim >= 0.8) {
    context.write(new Text("(" + key.getFirst() + ", " + key.getSecond() + ")")
        ,
            new Text(String.valueOf(sim)));
}
```

(5) Report the executive time and the number of performed comparisons.

To have the number of comparisons, we defined an enum (*MYCOUNTER*) and the counter we used to count (*totcompare_a*) at begining:

```
public class similarityjoin_A extends Configured implements
        Tool {

    // define a customer counter to count the number of comparisons
    public static enum MYCOUNTER {
        totcompare_a,
    };
```

And we also add the following to save this counter value in HDFS as a text file before the completion of the job, the same as we did in the pre-processing:

```
// save the counter value in HDFS as a text file
myjob.waitForCompletion(true);
long mycounter = myjob.getCounters()
        .findCounter(MYCOUNTER.totcompare_a).getValue();
Path outFile = new Path("totcompare_a.txt");
BufferedWriter br = new BufferedWriter(new OutputStreamWriter(
        fs.create(outFile, true)));
br.write(String.valueOf(mycounter));
br.close();
return 0;
```

The result of the total number of comparisons is 19900.

Executive time: 55 seconds

b) Create an inverted index, only for the first |d| - [t|d] + 1 words of each document d (remember that they are stored in ascending order of frequency) (1). In your reducer, compute the similarity of the document pairs. (2) Output only similar pairs on HDFS, in TextOutputFormat. (3) Report the execution time and the number of performed comparisons. (4)

(1) Compute the prefix filtering number.

In the map phase, we define the number |d| - [t|d] + 1 (t = 0.8) for each line to keep only this number of words of each document and output an inverted index of each words:

```
public static class Map extends Mapper<Text, Text, Text, Text> {
    private Text word = new Text();

    // define the number |d| - [0.8|d] + 1
    @Override
    public void map(Text key, Text value, Context context)
            throws IOException, InterruptedException {

        String[] wordlength = value.toString().split(" ");
        long nb_keep = Math.round(wordlength.length
                - (wordlength.length * 0.8) + 1);
        String[] keeplength = Arrays.copyOfRange(wordlength, 0,
                (int) nb_keep);

        // output an inverted index for each selected words
        for (String keep : keeplength) {
            word.set(keep);
            context.write(word, key);
        }
    }
}
```

(2) In the reduce phase, compute the similarity of the document pairs.

The same as we did in (a):

```
public static class Reduce extends Reducer<Text, Text, Text, Text> {

    private BufferedReader reader;

    // compute the jaccard similarity (intersection/union)
    public double similarity(TreeSet<String> str1, TreeSet<String> str2) {

        if (str1.size() < str2.size()) {
            TreeSet<String> s1bis = str1;
            s1bis.retainAll(str2);
            int intersection = s1bis.size();
            str1.addAll(str2);
            int union = str1.size();
            return (double) intersection / union;
        } else {
            TreeSet<String> s1bis = str2;
            s1bis.retainAll(str1);
            int intersection = s1bis.size();
            str2.addAll(str1);
            int union = str2.size();
            return (double) intersection / union;
        }

    }
```

And also, load the wordlist (without frequency):

```
// load the word list (without frequency) into a HashMap
@Override
public void reduce(Text key, Iterable<Text> values, Context context)
        throws IOException, InterruptedException {

    HashMap<String, String> lineshashmap = new HashMap<String, String>();
    reader = new BufferedReader(
            new FileReader(
                    new File(
                            "/home/cloudera/workspace/assignment2/output/wordlist_asc.txt")));
    String pattern;
    while ((pattern = reader.readLine()) != null) {
        String[] line = pattern.split(",");
        lineshashmap.put(line[0], line[1]);
    }
```

If a word is not present in only one document, we get all possible pairs of documents in the list:

```
//If a word is not present in only one document, we get all possible pairs
if (wordlength.size() > 1) {
    ArrayList<String> pairs = new ArrayList<String>();
    for (int i = 0; i < wordlength.size(); ++i) {
        for (int j = i + 1; j < wordlength.size(); ++j) {
            String pair = new String(wordlength.get(i) + " "
                    + wordlength.get(j));
            pairs.add(pair);
        }
    }
```

And we now compute the jaccard similarity between 2 documents (doc1, doc2):

```
// create 2 sets to compute the similiarity
for (String pair : pairs) {
    TreeSet<String> words_1_treeset = new TreeSet<String>();
    String words_1_string = lineshashmap
            .get(pair.split(" ")[0].toString());
    for (String word : words_1_string.split(" ")) {
        words_1_treeset.add(word);
    }

    TreeSet<String> words_2_treeset = new TreeSet<String>();
    String words_2_string = lineshashmap
            .get(pair.split(" ")[1].toString());
    for (String word : words_2_string.split(" ")) {
        words_2_treeset.add(word);
    }

    // compute the jaccard similarity between 2 sets
    double sim = similarity(words_1_treeset, words_2_treeset);
```

(3) Output only similar pairs on HDFS, in TextOutputFormat.

```
// Assuming threshold t = 0.8
if (sim >= 0.8) {
    context.write(new Text("(" + pair.split(" ")[0] + ", "
            + pair.split(" ")[1] + ")"),
            new Text(String.valueOf(sim)));
}
```

(4) Report the execution time and the number of performed comparisons.

Define an enum (*MYCOUNTER*) and the counter we used to count (*totcompare_b*) to have the number of comparisons:

```
// define a customer counter to count the number of comparisons
public static enum MYCOUNTER {
    totcompare_b,
};
```

Increment the number of comparisons by 1:

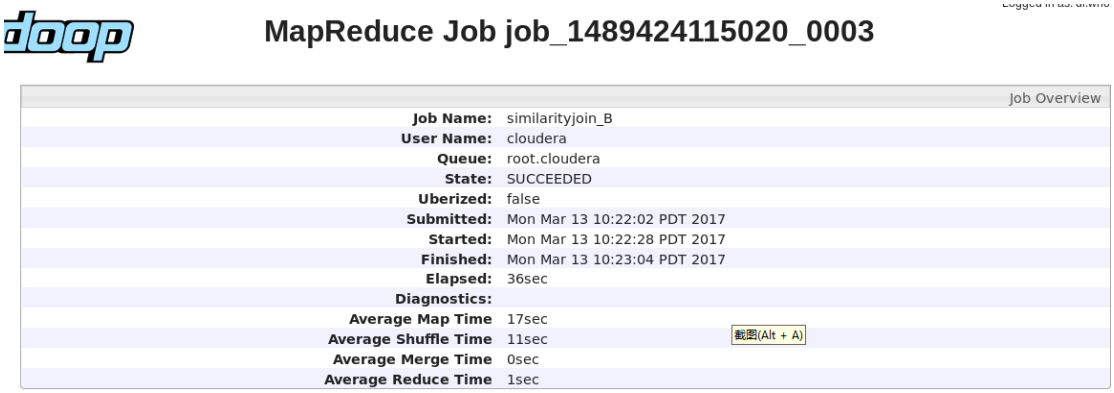
```
// increment the counter by 1
context.getCounter(MYCOUNTER.totcompare_b).increment(1);
```

Save this counter value in HDFS as a text file before the completion of the job:

```
// save the counter value in HDFS as a text file
myjob.waitForCompletion(true);
long mycounter = myjob.getCounters()
        .findCounter(MYCOUNTER.totcompare_b).getValue();
Path outFile = new Path("totcompare_b.txt");
BufferedWriter br = new BufferedWriter(new OutputStreamWriter(
        fs.create(outFile, true)));
br.write(String.valueOf(mycounter));
br.close();
return 0;
```

The result of the total number of comparisons is 111.

Executive time: 36 seconds



c) Explain and justify the difference between a) and b) in the number of performed comparisons, as well as difference in execution time.

Number of comparisons: (A) 19900 (B) 111
Execution time: (A) 55s (B) 36s

The number of comparisons of the first method is obviously greater than the second one, it's because the second method doesn't index all words for each documents, which are some words with lower frequency for this document in the given corpus. This enables the job to skip a large number of pairs containing documents that are not similar and all similar pairs are compared. Consequently, the second method takes less execution time as it put emphasis on reducing the number of comparisons.