



Programmierkonzepte und Algorithmen

Prof. Dr. Nikita Kovalenko

SoSe 2020

Häufigkeitsberechnung mit Hadoop

Andrej Loparev

Truong An Nguyen

Einleitung	2
Aufgabenstellung	2
Arbeitsaufbau	2
Planung	2
hadoop.MyMapper	3
hadoop.MyReducer	5
java_parallel & utils	6
Umsetzung	6
hadoop.MyMapper	6
hadoop.MyReducer	8
java_parallel & utils	9
Auswertung	11
Ausführliche Tests der Anwendung	13
Diagramme für die Laufzeit	18
Fazit	19
Anhang	20

Einleitung

Unsere Welt wird zunehmend digital. Der Trend wird durch die aktuelle Corona-Situation noch weiter verstärkt, in dem die Menschen auf die Telekommunikation- und Cloud-Technologien angewiesen sind. Das bringt mit sich einen enormen Datenzuwachs, der verarbeitet und gespeichert werden muss. Wenn alle existierende digitale Daten zusammengezählt werden, dann bewegen wir uns im Zettabyte-Bereich. Die Rede ist auch von Big Data.

Der Begriff Big Data bezeichnet Datenmengen, die aufgrund von ihrer Größe, Struktur, Schnelllebigkeit oder Komplexität nicht mit herkömmlichen Verarbeitungsmethoden auswertbar sind. Daher wurden neue Systeme entwickelt, um Big Data doch noch bearbeiten zu können, wie z.B. Spark und Hadoop. Das Ziel dieser Arbeit besteht darin, Hadoop für eine Textbearbeitungsaufgabe einzusetzen.

Aufgabenstellung

Die Aufgabe ist, in einem zur Verfügung gestellten Datensatz eine Vorkommenshäufigkeit der Wörter zu ermitteln. Anschließend sollen die Top-10 davon ausgegeben werden. Der Datensatz besteht aus acht Ordner mit literarischen Werken verschiedener Sprachen. Jeder Ordner repräsentiert eine konkrete Sprache aus der folgenden Liste: Niederländisch, Englisch, Französisch, Deutsch, Italienisch, Russisch, Ukrainisch und Spanisch. Um die Ergebnisse vergleichbar zu machen, werden zusätzlich alle Stoppwörter aus den Werken entfernt.

Die Herausforderung besteht dabei in der Aufteilung der Sprachen auf mehrere Ordner, sodass die Suche nicht ohne weitere Tricks durchführbar ist. Wir müssen also beim Zählen zwischen den Sprachen differenzieren können, um im Nachhinein eine Top-10 Aufstellung pro Sprache zu ermöglichen.

Arbeitsaufbau

Die Dokumentation ist in zwei Hauptphasen gegliedert: Planung und Umsetzung. Davor wird es kurz auf Aufgabenbeschreibung und Kontext eingegangen. Die Umsetzung erläutert die Implementierungsschritte, sowie den entstandenen Code. Anschließend folgen Tests der Anwendung, deren Schwerpunkt bei Zeitmessungen liegt. Das Dokument schließt mit einem Fazit. Darüber hinaus werden die zum Anwendungsstart notwendige Schritte im Anhang festgehalten, dazu gehört auch die Docker-Installation unter Windows.

Planung

Grundsätzlich wurden zwei Lösungsansätze gefunden und als machbar eingestuft. Aus Zeitgründen konnten wir davon allerdings nur einen umsetzen. Der erste Ansatz wäre, die Sprachen nacheinander zu bearbeiten und die Ergebnisse zwischenspeichern. Diese

Zwischenergebnisse würden dann in einem zweiten MapReduce-Durchgang nochmal aufgegriffen und in ein finales Ergebnis zusammengeführt.

Der zweite Ansatz, für den wir uns entschieden, besteht im Sprachmapping. Das bedeutet, dass wir nicht nur die Wörter in den Hadoop-Context schreiben, sondern auch deren Ordner, der mit Unterstrich vorne angehängt wird. Dadurch kann die Verarbeitung in einem Durchgang erfolgen. Zum Beispiel wird aus dem englischen „good“ ein Schlüssel-Wert-Paar (`"english_tee"`, `1`) was in den Context geladen wird. Daraus wird später die Sprachzugehörigkeit des Wortes ermittelt.

Das Hadoop-Programm besteht aus drei Teilen: Mapper, Reducer und Driver. Es ist im Paket „hadoop“ zu finden. Außerdem wurde, in einem zweiten Paket namens „java_parallel“, ein Java-Äquivalent zum Vergleich erstellt. Die Texte wurden im Ressourcen-Ordner untergebracht. Ein weiterer Ordner wurde mit Bashskripten für Docker-Deployment kreiert.

Der Driver für die Ausführungssteuerung wird in einer separaten Klasse, WordCount, umgesetzt. Sie enthält eine Sprachliste, sowie eine Hadoop-Konfiguration. Die Konfiguration definiert die zu startende Java-Main-Klasse, sowie die aufgabenspezifische Map-Reduce-Implementierungen und Eingabe-Ausgabe-Formate. Beim Programmstart werden Parameter zu In- und Outputordnern übergeben, damit Hadoop weiß wo er die relevanten Dateien auf HDFS findet. Darüber hinaus gibt es einige hilfreiche Logausgaben, um den Programmverlauf nachvollziehen zu können.

hadoop.MyMapper

Hier wird die Methode `map()` von der Klasse Mapper beschrieben. Wobei wir unsere beide Iterationen erwähnen möchten. Bei der ersten Variante wurde das Basisbeispiel implementiert und bei der zweiten die Anpassung oben drauf für die gegebene Aufgabe gemacht.

Iteration 1

Zuerst teilen wir die Zeilenstrings aus den Textdateien in einzelne Worte auf. Danach erstellen wir Schlüssel- und Wertepaare, die an Reduzier über Context gesendet werden. Jeder Schlüssel entspricht exakt einem Wort aus der Textzeile. Und als Wert wird eine eins übergeben, die den Zählerstand signalisiert. Beispiel 1:

Parameter:

Worte: `"cat dog mouse"`.

Verarbeitung:

Schlüssel: `"cat"`, `"dog"`, `"mouse"`.

Wert : `1`.

Output:

`("cat", 1)`, `("dog", 1)`, `("mouse", 1)`.

Iteration 2

Laut der Aufgabenanforderungen brauchen wir jedoch die Häufigkeit für jede Sprache getrennt. Dazu ist zu der Struktur aus ersten Iteration die sprachliche Zuordnung hinzuzufügen. Und um zu wissen, zu welcher Sprache die Kette gehört, müssen wir den Namen des Ordners holen, der die Textdateien enthält und einer Sprache entspricht. Daraus kann der Schlüssel in die folgende Form konvertiert werden: "<Sprache>_<Wort>". Außerdem müssen alle Wörter in kleinschreibung umgewandelt werden, um Konflikte beim Schlüssel-Vergleich zu vermeiden.

Um diese zu illustrieren schauen wir nochmal auf die Wörter des ersten Beispiels: cat, dog, mouse. Sie stammen alle aus der englischen Sprache, sodass sich die Ausgabe ändert:

Parameter:

Worte: "cat dog mouse".

Verarbeitung:

Schlüssel: "english_cat", "english_dog", "english_mouse".

Wert : 1.

Output:

("english_cat", 1), ("english_dog", 1), ("english_mouse", 1).

Die Mapping-Phase wird durch Hadoop gesteuert, um die optimale Verteilung der Dateien zu erreichen. Die vorhandene Textdateien werden gruppiert und separat, parallel verarbeitet, wie auf Bild 1 dargestellt.

Nachdem der Mapping-Prozess beendet ist, werden die sogenannten Sort- und Shuffle-Phasen durchgeführt, um die Paare $\langle \text{Key}, \{\text{value}, \text{value}, \dots\} \rangle$ zu erstellen. Anschließend werden diese Paare an Reducer gesendet.

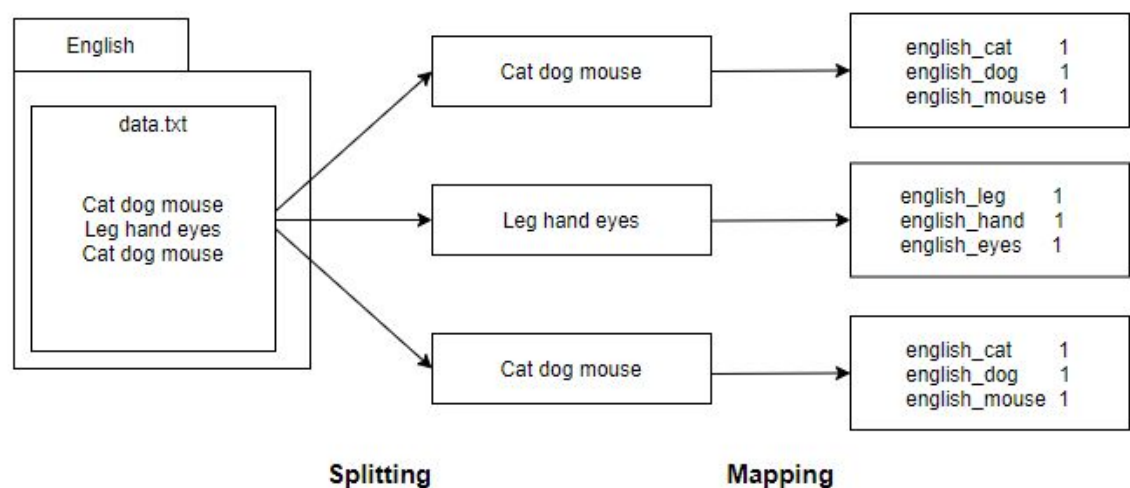


Bild 1: Mapper-Funktionsweise

hadoop.MyReducer

Das nächste Programmstück ist der Reducer. Seine Hauptaufgabe ist die vom Mapper gekriegten Schlüssel-Werte-Paare zu zählen und die Top-10 auszugeben.

Dafür werden Klassenvariablen für die gezählten Worte pro Sprache, sowie Stoppworte angelegt. Die Variablen sind HashMaps, die als Schlüssel eine Sprache haben und als Wert eine weitere HashMap. Diese weitere HashMap enthält letztendlich die Worte und deren Häufigkeit: `HashMap<String, HashMap<String, Integer>>`.

Methode `reduce()`

Reducer arbeitet mit dem vom Mapper erzeugten Datensatz. Entsprechen der Schlüsselstruktur müssen diese zunächst nach Sprache und Wort aufgeteilt werden. Danach können die Stoppworte für diese Sprache geladen werden, um die Filterung, bzw. Inkrementierung vorzunehmen.

Methode `cleanup()`

Um Top-10 Wörter zu entnehmen, benutzen wir die Methode `cleanup()` aus dem Reducer. Sie wird einmalig kurz vom Beenden der Ausführung aufgerufen. Hier werden die Wörter, mit einem customized Comparator, auch nach ihrer Häufigkeit sortiert. Anschließend nehmen wir nur die zehn obersten aus jeder Sprache und schreiben sie in Hadoop-Context. Bild 2 zeigt die übersicht der Zusammenarbeit zwischen Mapper und Reducer mit einem fiktiven Beispiel.

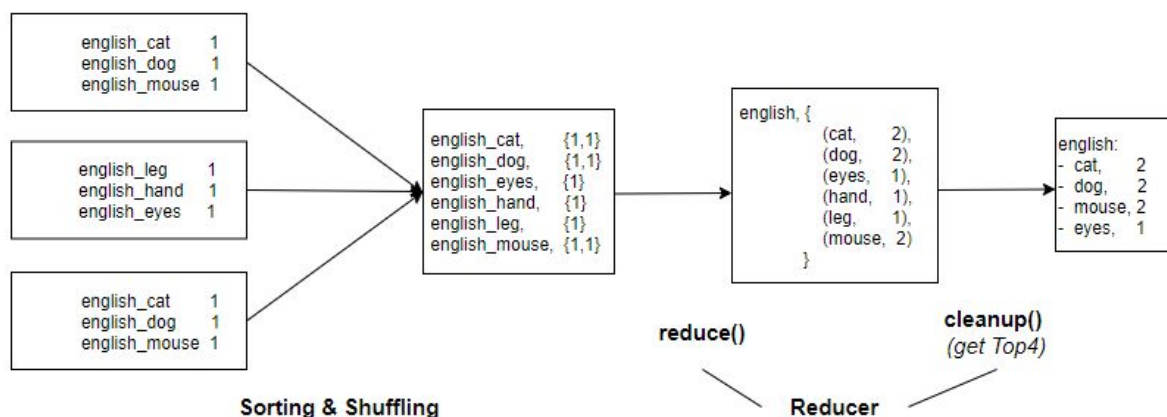


Bild 2: Reducer-Funktionsweise für Top-4 Häufigkeitsberechnung

java_parallel & utils

Als zusätzliche Fleißaufgabe, haben wir noch ein zweites Programm mit den gleichen Funktionen in Java entwickelt. Das fanden wir nicht nur wegen dem Vergleich spannend, sondern auch deshalb, weil Java die default Programmiersprache im Bachelor war. Der Beherrschungsgrad ist hoch, sodass eine Entwicklung innerhalb von „fünf Minuten“ zu erwarten war. Es ist zu beachten, dass ein Leistungs- und Zeitvergleich nur bei dem vorliegenden Datensatz von ca. 130 Megabyte möglich ist. Sollte es um eine wirkliche Big Data Auswertung gehen, wird voraussichtlich Hadoop die bessere Wahl sein. Allein schon wegen der limitierten RAM-Größe, die Java zur Verfügung steht.

Die Umsetzung wird auf drei Klassen verteilt: Main, WordCountJavaParallel und WordCountJavaResult. Main ist, wie auch bei Hadoop, die Driverklasse. Sie enthält die Sprachliste, führt damit die Zähllogik durch, bekommt daraufhin ein Result-Objekt und gibt dieses aus. Das Result-Objekt beinhaltet die zu untersuchende Sprache, Laufzeit und eine Liste mit Wort-Anzahl-Paaren. Die Laufzeiten pro Sprache werden vom Driver summiert und am Ende gezeigt.

WordCountJavaParallel ist die Klasse mit der Zähllogik. Sie wird eine Entry-Point-Methode (Zeitmessung und Management) und zwei Worker-Methoden umfassen. Ein Worker soll die Worte lesen und vom Stoppwörtern befreien, während der andere sich mit der Zählung beschäftigt. Es wird geplant einen starken Gebrauch von Streams zu machen.

Zum IO-Handling werden statische Hilfsmethoden in der Utilsklasse erstellt. Sie sollen sowohl vom Hadoop- als auch vom Java-Code benutzt werden.

Umsetzung

In diesem Kapitel werden die geplanten Maßnahmen in den Code umgewandelt und anhand von Screenshots erklärt.

hadoop.MyMapper

Abbildung 3 zeigt unsere Mapper-Implementierung, dabei werden in den Zeilen folgende erwähnenswerte Schritte durchgelaufen:

1. Map für die Stoppworte wird klassenweit erzeugt und später mit der jeweiligen Wordsets pro Sprache gefüllt
2. beide Hadoop-Context-Variablen, Wort und Zähler, werden als Text initialisiert, das wird am Ende für die Ergebnisausgabe verwendet
3. die Datei wird eine Zeile nach der anderen als String gelesen und in die Kleinschreibung konvertiert
4. die Zeile wird mit Hilfe von Regex aufgeteilt und in die zuvor deklarierte String-Liste auf Einmal hinzugefügt

```

17 public class MyMapper extends Mapper<LongWritable, Text, Text, Text> {
18     private final static Text one = new Text( string: "1");
19     private Text word = new Text();
20     /**
21      * contains stopword lists for each language
22      * key:language value: stopwords of the language
23      */
24     private HashMap<String, HashSet<String>> stopwordList = new HashMap<>();
25
26     @Override public void map(LongWritable key, Text value, Context context) throws IOException, InterruptedException {
27         String line = value.toString().toLowerCase();
28         List<String> words = new ArrayList<>();
29         // splits value and remove the special character at the same time
30         Collections.addAll(words, line.split( regex: "\\P{L}+"));
31         // get language folder
32         String filePathString = ((FileSplit) context.getInputSplit()).getPath().toString();
33         String language = new Path(filePathString).getParent().getName().toLowerCase();
34         // load stopwords
35         if (!stopwordList.containsKey(language)) {
36             HashSet<String> stopwords = Utils.readFileStopword( fileName: language + ".txt");
37             if (stopwords == null) { // file not found, do nothing
38                 return;
39             }
40             // add the stopwords list of the associated language to the general list
41             stopwordList.put(language, stopwords);
42         }
43         // loop >> glue language and words
44         for (String w : words) {
45             // delete words less than or equal to 2
46             if (w.length() <= 2) {
47                 continue; // ignore
48             }
49             // if the word is stopword
50             if (stopwordList.get(language).contains(w)) {
51                 continue; // ignore
52             }
53             word.set(language + "_" + w); // language_word, Ex: english_water
54             context.write(word, one);
55         }
56     }
57 }

```

Bild 3: Mapper-Code

5. Regex „\P{L}+“ interessiert sich für alle nicht unicode Zeichen; dabei signalisiert das Pluszeichen, dass der Treffervorkommen von Einmal bis unendlich oft gematcht wird
6. Stoppworte werden geladen und ausgefiltert
7. alle Wörter mit zwei oder weniger Buchstaben werden rausgeschmissen; das ist sehr hilfreich mit Blick auf die zahlreiche existierende Stoppwörter, hat aber keine Auswirkungen auf das Ergebnis beim vorliegenden Datensatz, weil solche kurze Worte selten sind
8. wie schon in der Planung erläutert, holen wir uns die Sprache aus dem Ordernamen und mappen diese mit eine Unterstrich zu jedem Wort dieser Sprache für spätere wiedererkennung in einer For-Schleife
9. das wird dann in den Hadoop-Context mit dem Wert 1 im Textformat geschrieben

hadoop.MyReducer

```
14 public class MyReducer extends Reducer<Text, Text, Text, Text> {
15     /**
16      * This variable (allData) contains the words and the counter of words for each language
17      * <p>
18      * key: language
19      * value: {
20      *   key: word,
21      *   value: counter of the word
22      * }
23      */
24     private HashMap<String, HashMap<String, Integer>> allData = new HashMap<>();
25     private final int MAX_TOP = 10;
26
27     /**
28      * key: language_word values: counter of word
29      */
30     @Override
31     public void reduce(Text key, Iterable<Text> values, Context context) throws IOException, InterruptedException {
32         String[] language_key = key.toString().split(regex: "\\s");
33         String language = language_key[0]; //language
34         String str = language_key[1]; //word
35         //calculate the counter of word
36         int sum = 0;
37         for (Text val : values) {
38             sum += Integer.parseInt(val.toString());
39         }
40         //add word and word counters to the list according to the corresponding language
41         if (allData.containsKey(language)) {
42             allData.get(language).put(str, sum);
43         } else {
44             HashMap<String, Integer> hmap = new HashMap<>();
45             hmap.put(str, sum);
46             allData.put(language, hmap);
47         }
48     }
49 }
```

Bild 4: Reducer-Code

Abbildung 4 demonstriert unsere Reducer-Implementierung, dabei werden in den Zeilen folgende erwähnenswerte Schritte durchgelaufen:

1. Wir erstellen eine klassenweite Mapvariable, allData, für die Kommunikation zwischen einzelnen Methoden
2. allData ist für die Akkumulierung der Ausgangsdaten vorgesehen, die Struktur ist `< Sprache, < Wort, Zähler > >`; dabei gibt es für jede Sprache eine weitere Map die die Häufigkeitsberechnung umfasst
3. die vom Context empfangene Strings werden in Sprache und Word aufgeteilt
4. die Worte werden summiert und in allData untergebracht, wenn noch nicht vorhanden wird ein neuer Worteintrag erstellt

```

30 // cleanup: Called once at the end of the task.
31 @Override
32 protected void cleanup(Context context) throws IOException, InterruptedException {
33     // Create a comparator to compare and sort
34     Comparator<Entry<String, Integer>> comparator = new Comparator<Entry<String, Integer>>() {
35         @Override
36         public int compare(Entry<String, Integer> o1, Entry<String, Integer> o2) {
37             return o2.getValue().compareTo(o1.getValue());
38         }
39     };
40     /**
41     * sort and print EX: ----- dutch
42     * Top 10 words=[den=1520, zoo=512, marten=330, baas=303, zien=230, jan=224, vrouw=201, oogen=194, riep=185, goed=178]
43     */
44     for (Entry<String, HashMap<String, Integer>> entry : allData.entrySet()) {
45         // create a list of word and counter
46         List<Entry<String, Integer>> values = new ArrayList<>(entry.getValue().entrySet());
47         // sort descending by counter
48         Collections.sort(values, comparator);
49         StringBuilder sb = new StringBuilder("-----\n");
50         sb.append(entry.getKey()); // print name of language
51         sb.append("\nTop " + MAX_TOP + " words=");
52         sb.append(values.subList(0, values.size() > MAX_TOP ? MAX_TOP : values.size()));
53         context.write(new Text(sb.toString()), new Text(" string: "\n"));
54     }
55 }

```

Bild 5: Reducer-Cleanup

Abbildung 5 illustriert die Cleanup-Methode. Sie startet nachdem alle Prozesse ihre Reduktion abschlossen, um die Worte absteigend nach dem Zähler zu sortieren und das Ergebnis in Hadoop-Ausgabe zu printen. Für die Sortierung wurde ein Comparator, basierend auf dem Map-Wert herangezogen. Für die Ausgabe bilden wir einen großen String mit allen Worten.

java_parallel & utils

Für die Implementierung wurde Java 8 verwendet. Um einen besseren Vergleich mit Blick auf die Parallelität zu erzielen, sind Streams und deren spezielle Operanden verwendet worden. Bei Operanden gibt es Funktionen wie map, filter und sogar reduce. Es ist also theoretisch möglich genau den selben Ablauf wie bei Hadoop umzusetzen.

Bild 6 illustriert die selbsterklärende Driverklasse, dazu sollten keine Frage aufkommen. Die viel spannendere Logikklasse ist auf Bild 7 und Bild 8 zu sehen. Am Anfang sind all die unterschiedlichen Regex zu finden, die wir testeten. Die vorhandenen Methoden sind statisch und können ohne Klasseninstantiierung aufgerufen werden.

```

1  package java_parallel;
2
3  import utils.Utils;
4
5  public class Main {
6      public static void main(String[] args) throws Exception {
7          long javaRunTimeTotalMs = 0;
8          String[] languages = new String[]{"dutch", "english", "french", "german", "italian", "russian", "spanish",
9              "ukrainian"};
10
11         for (String language : languages) {
12             System.out.println("java parallel: start processing " + language);
13             WordCountJavaResult javaResult = WordCountJavaParallel.getWordCountResultWithTime(language,
14                 Utils.readAllFilesFromDir(language), Utils.readStopwords(language));
15
16             System.out.println("java parallel: top10words=" + javaResult.top10Words);
17             System.out.println("java parallel: run time in ms=" + javaResult.runTimeMs);
18             System.out.println("-----");
19             javaRunTimeTotalMs += javaResult.runTimeMs;
20         }
21         System.out.println("\njava parallel: total run time in seconds: " + javaRunTimeTotalMs / 1000);
22     }
23 }

```

Bild 6: Java-Parallel-Driver-Code

Entrypoint ist `getWordCountResultWithTime`, die Methode nimmt als Parameter die Sprache und zwei Stringlisten mit Textzeilen/Stopwörtern. Die Listen, geladen mit Hilfe von `Utils`, werden aus dem Driver übergeben und an die Workermethoden für die Verarbeitung weitergeleitet. Die Methode startet eine Zeitmessung und führt währenddessen die Filter-/Zählvorgänge durch. Die Filterzwischenergebnisse landen in einem `Supplier` als `Stream`. Das spart später eine erneute `List-Stream-Konvertierung`. Zählzwischenergebnisse landen in einer Liste mit Wort-Anzahl-Tupeln. Da es in Java keine Tupeln „out of the box“ gibt, wird dafür eine `Mapentry` genommen. Anschließend erzeugt `getWordCountResultWithTime` ein Ergebnisobjekt mit Sprache, Zeit und Top10 Häufigkeiten, was an den Driver zurückgeliefert wird.

```

11  public class WordCountJavaParallel {
12
13      //private static final String splitRegex1 = "[\\s\\d\\p{P}]+(?![!-])";
14      private static final String splitRegex2 = "\\P{L}+";
15      //private static final String splitRegex3 = "\\W+";
16      //private static final String splitRegex4 = "\\P{L}+ | (?![\\p{L}])\\p{L}(?!\\p{L})";
17      //private static final String splitRegex5 = "\\P{L}+ | \\b.{0,2}\\b";
18
19      public static WordCountJavaResult getWordCountResultWithTime(String language, List<String> text, List<String> stopwords) {
20          long startTime = System.nanoTime();
21          Supplier<Stream<String>> words = () -> splitAndCleanLines(text, stopwords);
22          List<Map.Entry<String, Long>> wordsFreqSorted = countWords(words);
23          long endTime = System.nanoTime();
24          long totalTime = TimeUnit.MILLISECONDS.convert(endTime - startTime, TimeUnit.NANOSECONDS);
25          return new WordCountJavaResult(language, wordsFreqSorted.subList(0, 10), totalTime);
26      }

```

Bild 7: Java-Parallel-Logik-Code (Teil 1)

`splitAndCleanLines` führt den Filtervorgang, basierend auf den beiden Listen, durch. Dazu wird zunächst ein paralleler `Stream`, dann werden die Zeilen aufgeteilt. Die Worte werden herausgezogen, in Kleinschreibung umgewandelt und mit `Regex` eliminiert, falls es sich um ein Stoppwort handelt. Außerdem filtern wir auch hier alle Worte mit weniger als drei Buchstaben aus. Output ist ein `Stream` mit den zu zählenden Worten.

```

28     private static Stream<String> splitAndCleanLines(List<String> text, List<String> stopwords) {
29         return text
30             .stream()
31             .parallel()
32             .map(line -> line.split(" "))
33             .flatMap(Stream::of)
34             .map(String::toLowerCase)
35             .map(word -> word.replaceAll(splitRegex2, ""))
36             .filter(word -> !stopwords.contains(word) && word.length() > 2);
37     }
38
39     private static List<Map.Entry<String, Long>> countWords(Supplier<Stream<String>> words) {
40         return words.get()
41             .parallel()
42             .collect(Collectors.groupingBy(String::toString, Collectors.counting()))
43             .entrySet()
44             .stream()
45             .sorted(Map.Entry.comparingByValue(Comparator.reverseOrder()))
46             .collect(Collectors.toList());
47     }
48 }
49

```

Bild 8: Java-Parallel-Logik-Code (Teil 2)

Bild 7 demonstriert auch die Zählermethode namens `countWords`. Sie braucht zum Starten einen Wörter-Stream, der über `Supplier` übergeben werden kann. Die Wörter werden gruppiert, gezählt und sortiert als Output ausgegeben.

Auswertung

Die Ausführung von Hadoop-Programm wurde sowohl in einer Windows-Umgebung als auch in einem Docker-Container durchgeführt. Start ist mit dem Befehl `hadoop jar <application_path> <input_folder> <output_folder>` möglich.

Auf Bild 9 ist der Aufruf und Bearbeitungslogs zu sehen. Die Logs sind unter anderem für die Zeitmessung ganz praktisch. Aus dem Bild können wir uns den Startzeitpunkt um 12 Uhr 5 Minuten merken.

Während der Ausführung printet Hadoop auch wie weit die Programmphasen, Map und Reduce, sind. Sollte alles erfolgreich durchlaufen, kommen zum Schluss noch ein Paar statistische Ausgaben zu Zähler, Errors usw. Für uns ist der Endzeitpunkt interessant, wie auf Bild 10 demonstriert, beträgt er 12:26. Daraus ergibt sich eine Laufzeit von 21 Minuten.

Das Java-Programm kann direkt aus der IDE gestartet werden. Die Laufzeit hier beträgt etwa 20 Sekunden. Die Ergebnisse beider Programmen sind auf den Bildern 11 und 12 zu sehen.


```

Administrator: Command Prompt - C:\Windows\System32\cmd.exe /k "M: & cd M:\hadoop-3.0.0-alpha2\sbin"
M:\hadoop-3.0.0-alpha2\sbin>hadoop jar Y:/bd/WordCount.jar /input /output
2020-06-04 12:05:07,987 INFO  hadoop.WordCount: hadoop word count up and running
2020-06-04 12:05:08,012 INFO  hadoop.WordCount: run args: in=/input out=/output
2020-06-04 12:05:09,539 INFO  hadoop.WordCount: Output folder already exists... Overwriting
2020-06-04 12:05:09,664 INFO  client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
2020-06-04 12:05:10,271 WARN  mapreduce.JobResourceUploader: Hadoop command-line option parsing not performed. Implement
the Tool interface and execute your application with ToolRunner to remedy this.
2020-06-04 12:05:10,623 INFO  input.FileInputFormat: Total input files to process : 363
2020-06-04 12:05:10,769 INFO  mapreduce.JobSubmitter: number of splits:363
2020-06-04 12:05:10,784 INFO  Configuration.deprecation: yarn.resourcemanager.system-metrics-publisher.enabled is deprec
ated. Instead, use yarn.system-metrics-publisher.enabled
2020-06-04 12:05:10,895 INFO  mapreduce.JobSubmitter: Submitting tokens for job: job_1591264548533_0001
2020-06-04 12:05:11,671 INFO  impl.YarnClientImpl: Submitted application application_1591264548533_0001
2020-06-04 12:05:11,726 INFO  mapreduce.Job: The url to track the job: http://DESKTOP-C4PTQAP:8088/proxy/application_159
1264548533_0001/
2020-06-04 12:05:11,727 INFO  mapreduce.Job: Running job: job_1591264548533_0001
2020-06-04 12:05:21,916 INFO  mapreduce.Job: Job job_1591264548533_0001 running in uber mode : false
2020-06-04 12:05:21,917 INFO  mapreduce.Job: map 0% reduce 0%
2020-06-04 12:05:45,915 INFO  mapreduce.Job: map 1% reduce 0%
2020-06-04 12:05:55,346 INFO  mapreduce.Job: map 2% reduce 0%
2020-06-04 12:06:13,417 INFO  mapreduce.Job: map 3% reduce 0%
2020-06-04 12:06:28,275 INFO  mapreduce.Job: map 4% reduce 0%
2020-06-04 12:06:40,033 INFO  mapreduce.Job: map 5% reduce 0%
2020-06-04 12:06:53,817 INFO  mapreduce.Job: map 6% reduce 0%
2020-06-04 12:07:11,854 INFO  mapreduce.Job: map 7% reduce 0%
2020-06-04 12:07:25,986 INFO  mapreduce.Job: map 8% reduce 0%
2020-06-04 12:07:36,064 INFO  mapreduce.Job: map 9% reduce 0%
2020-06-04 12:07:55,424 INFO  mapreduce.Job: map 10% reduce 0%

```

Bild 9: Hadoop-Start

```

2020-06-04 12:25:59,870 INFO  mapreduce.Job: map 98% reduce 33%
2020-06-04 12:26:08,474 INFO  mapreduce.Job: map 99% reduce 33%
2020-06-04 12:26:18,776 INFO  mapreduce.Job: map 100% reduce 33%
2020-06-04 12:26:23,795 INFO  mapreduce.Job: map 100% reduce 52%
2020-06-04 12:26:29,834 INFO  mapreduce.Job: map 100% reduce 76%
2020-06-04 12:26:35,844 INFO  mapreduce.Job: map 100% reduce 100%
2020-06-04 12:26:35,848 INFO  mapreduce.Job: Job job_1591264548533_0001 completed successfully
2020-06-04 12:26:35,951 INFO  mapreduce.Job: Counters: 49
    File System Counters
        FILE: Number of bytes read=316344481
        FILE: Number of bytes written=701315523
        FILE: Number of read operations=0
        FILE: Number of large read operations=0
        FILE: Number of write operations=0
        HDFS: Number of bytes read=141542901
        HDFS: Number of bytes written=1484
        HDFS: Number of read operations=1094
        HDFS: Number of large read operations=0
        HDFS: Number of write operations=2
    Job Counters
        Launched map tasks=363
        Launched reduce tasks=1
        Data-local map tasks=363
        Total time spent by all maps in occupied slots (ms)=5923625
        Total time spent by all reduces in occupied slots (ms)=1014499
        Total time spent by all map tasks (ms)=5923625
        Total time spent by all reduce tasks (ms)=1014499
        Total vcore-milliseonds taken by all map tasks=5923625
        Total vcore-milliseonds taken by all reduce tasks=1014499
        Total megabyte-milliseonds taken by all map tasks=6065702000
        Total megabyte-milliseonds taken by all reduce tasks=1038846976
    Map-Reduce Framework
        Map input records=1371628
        Map output records=17221283
        Map output bytes=281901909
        Map output materialized bytes=316346653
        Input split bytes=52948
        Combine input records=0
        Combine output records=0
        Reduce input groups=638116
        Reduce shuffle bytes=316346653
        Reduce input records=17221283
        Reduce output records=8
        Spilled Records=34442566
        Shuffled Maps =363
        Failed Shuffles=0
        Merged Map outputs=363
        GC time elapsed (ms)=74159
        CPU time spent (ms)=0
        Physical memory (bytes) snapshot=0
        Virtual memory (bytes) snapshot=0
        Total committed heap usage (bytes)=112926916608
    Shuffle Errors
        BAD_ID=0
        CONNECTION=0
        IO_ERROR=0
        WRONG_LENGTH=0
        WRONG_MAP=0
        WRONG_REDUCE=0
    File Input Format Counters
        Bytes Read=141489953
    File Output Format Counters
        Bytes Written=1484
Done!

```

Bild 10: Hadoop-Ende

```

-----
dutch
Top 10 words=[den=1520, zoo=512, marten=330, baas=303, zien=230, jan=224, vrouw=201, oogen=194, riep=185, goed=178]

-----
ukrainian
Top 10 words=[треба=1029, сказав=810, над=805, мати=783, перед=758, добре=749, серце=738, уже=738, чінка=738, усе=731]

-----
german
Top 10 words=[hand=5610, augen=5172, herr=4815, leben=3924, sprach=3701, stand=3630, vater=3553, ließ=3375, mußte=3243, alte=3102]

-----
spanish
Top 10 words=[don=3073, casa=2451, ojos=1816, vida=1599, hombre=1565, dios=1508, señor=1497, padre=1395, mujer=1314, mano=1262]

-----
russian
Top 10 words=[глаза=298, сердце=271, жизни=234, знаю=224, горский=224, руки=209, слова=200, вера=192, точно=191, руку=183]

-----
english
Top 10 words=[man=11763, time=9585, good=6752, holmes=5899, long=5765, great=5534, day=5295, men=5018, thought=4828, night=4820]

-----
italian
Top 10 words=[disse=8759, rispose=4023, occhi=3614, donna=3087, mano=3055, altra=2968, voce=2862, uomini=2851, signor=2710, più=2626]

-----
french
Top 10 words=[homme=10266, faire=9207, monsieur=7084, point=6670, jeune=5891, femme=5866, fit=5731, yeux=5615, jamais=5572, oui=5526]

```

Bild 11: Hadoop Top10

```

java parallel: start processing dutch
java parallel: top10words=[den=1520, zoo=501, marten=330, baas=303, zien=230, jan=223, vrouw=201, oogen=194, riep=185, goed=178]
java parallel: run time in ms=332
-----
java parallel: start processing english
java parallel: top10words=[man=10631, time=9388, good=6181, holmes=5658, long=5562, great=5476, thought=4803, men=4662, day=4559, hand=4469]
java parallel: run time in ms=3915
-----
java parallel: start processing french
java parallel: top10words=[dune=9477, faire=9191, homme=7873, monsieur=7077, point=6633, jeune=5870, femme=5837, yeux=5578, jamais=5572, oui=5490]
java parallel: run time in ms=3989
-----
java parallel: start processing german
java parallel: top10words=[hand=5602, augen=5165, herr=4814, leben=3923, sprach=3693, stand=3629, vater=3551, ließ=3372, mußte=3243, alte=3102]
java parallel: run time in ms=7711
-----
java parallel: start processing italian
java parallel: top10words=[disse=8756, rispose=4023, occhi=3520, donna=3085, mano=3055, sera=2869, voce=2848, uomini=2741, signor=2710, più=2626]
java parallel: run time in ms=2281
-----
java parallel: start processing russian
java parallel: top10words=[глаза=297, сердце=270, жизни=234, горский=224, знаю=223, руки=209, слова=199, вера=192, точно=190, руку=182]
java parallel: run time in ms=291
-----
java parallel: start processing spanish
java parallel: top10words=[don=3071, casa=2437, ojos=1810, vida=1593, hombre=1559, dios=1490, señor=1481, padre=1375, mujer=1304, mano=1258]
java parallel: run time in ms=1135
-----
java parallel: start processing ukrainian
java parallel: top10words=[чого=1219, може=1109, треба=1028, коло=817, сказав=810, над=798, під=790, мати=777, перед=752, добре=748]
java parallel: run time in ms=415
-----
java parallel: total run time in seconds: 20

```

Bild 12: Java Top10

Ausführliche Tests der Anwendung

In diesem Abschnitt schauen wir genauer auf die Performance von einzelnen Sprachen. Pro Sprache werden drei Abläufe mit Zeitmessung unter Windows durchgeführt. Die Systemeigenschaften sind in Tabelle 1 aufgeführt. Darüber hinaus fassen wir die Anzahl der Inputfiles, deren Größe in Megabyte und die ermittelten Top10 Worte zusammen.

	Systemeigenschaften
OS	Windows 10 Pro 64 Bit
CPU	Intel i5-4250H (2.90GHz x 2)
RAM	12GB DDR3 1600 MHz
Drive	SSD

Tabelle 1: Eigenschaften des Testsystems

a. English

English			<p>Ergebnis</p> <p>-----</p> <p>english</p> <p>Top 10 words=[man=11763, time=9585, good=6752, holmes=5899, long=5765, great=5534, day=5295, men=5018, thought=4828, night=4820]</p>
Input	Kapazität	Laufzeit	
53 files	28.8 MB		
		3.73 Min.	
		3.35 Min.	
		3.80 Min.	

b. Dutch

Dutch			<p>Ergebnis</p> <p>-----</p> <p>dutch</p> <p>Top 10 words=[den=1520, zoo=512, marten=330, baas=303, zien=230, jan=224, vrouw=201, oogen=194, riep=185, goed=178]</p>
Input	Kapazität	Laufzeit	
5 files	685 KB		
		75 Sek.	
		75 Sek..	
		34 Sek.	

c. French

French			Ergebnis
Input	Kapazität	Laufzeit	-----
50 files	32.6 MB		french
		3.29 Min.	Top 10 words=[homme=10266, faire=9207,
		3.71 Min.	monsieur=7084, point=6670, jeune=5891,
		3.28 Min.	femme=5866, fit=5731, yeux=5615, jamais=5572, oui=5526]

d. German

German			Ergebnis
Input	Kapazität	Laufzeit	-----
50 files	32.5 MB		german
		3.69 Min.	Top 10 words=[hand=5610, augen=5172,
		3.69 Min.	herr=4815, leben=3924, sprach=3701,
		3.65 Min.	stand=3630, vater=3553, ließ=3375, mußte=3243, alte=3102]

e. Italian

Italian			Ergebnis
Input	Kapazität	Laufzeit	-----
50 files	19.4 MB		italian
		3.18 Min.	Top 10 words=[disse=8759, rispose=4023,
		3.17 Min.	occhi=3614, donna=3087, mano=3055,
		3.59 Min.	altra=2968, voce=2862, uomini=2851, signor=2710, piú=2626]

f. Russian

Russian			Ergebnis
Input	Kapazität	Laufzeit	-----
85 files	4.06 MB		russian
		4.87 Min.	Top 10 words=[глаза=298, сердце=271,
		4.90 Min.	жизни=234, знаю=224, горский=224,
		4.91 Min.	руки=209, слова=200, вера=192,
			точно=191, руку=183]

g. Spanish

Spanish			Ergebnis
Input	Kapazität	Laufzeit	-----
25 files	8.58 MB		spanish
		1.44 Min.	Top 10 words=[don=3073, casa=2451,
		1.86 Min.	ojos=1816, vida=1599, hombre=1565,
		1.47 Min.	dios=1508, señor=1497, padre=1395,
			mujer=1314, mano=1262]

h. Ukrainian

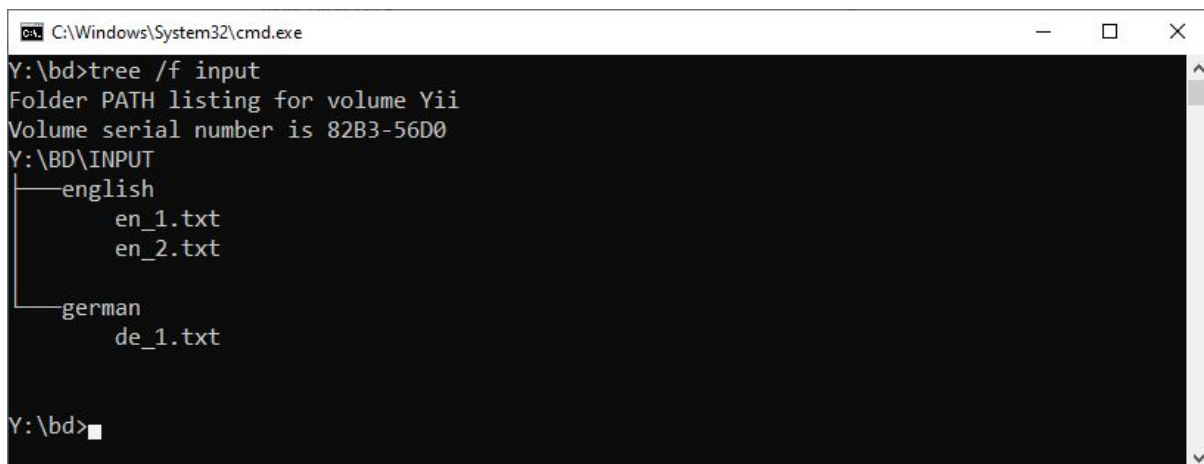
Ukrainian			Ergebnis
Input	Kapazität	Laufzeit	-----
46 files	8.27 MB		ukrainian
		2.91 Min.	Top 10 words=[треба=1029, сказав=810,
		2.92 Min.	над=805, мати=783, перед=758,
		2.49 Min.	добре=749, серце=738, уже=738,
			чіпка=738, усе=731]

Über alle Abläufe hinaus sind zwei Abhängigkeiten zu beobachten. Erste zwischen der Inputdatenanzahl und Laufzeit und zweite zwischen Inputgröße und Laufzeit. Wobei die erste signifikanter erscheint. Niederländisch umfasst zum Beispiel nur ein Zehntel der französischen Werke, aber die Laufzeit sinkt nur um Faktor dreieinhalb. Bei Spanisch und Französisch entspricht die Laufzeitänderung in etwa dem Unterschied in Dateieinanzahl. Da wird es offensichtlich, dass die Kapazität eine untergeordnete Rolle spielt, denn wenn es danach ginge, würde sich die Laufzeit um Faktor vier verkleinert. Das gleiche ist bei Spanisch und Ukrainisch erkennbar, wo sich die Laufzeit nach dem zweifachen Input verdoppelt, während die Kapazität unverändert bleibt.

Wir glauben, dass dieses Phänomen mit der Blockgröße in Hadoop fest verbunden ist. Wie bereits in der Einleitung erwähnt, wurde Hadoop hauptsächlich für die Bearbeitung von Big Data entwickelt und basiert auf einem virtuellen Dateisystem namens HDFS. HDFS besteht aus einer Unmenge von Speicherblöcken, vergleichbar mit RAM, aber wegen dem Anwendungsfall mit einer deutlich größeren Kapazität von 64MB (default Einstellung) mit der Erwartung, dass die Dateien über mehrere Blöcke verteilt werden. In unserer Aufgabe, sind die Dateien im Gegensatz dazu sehr klein, werden aber trotzdem in jeweils einen Block platziert, denn Hadoop kann die Dateien nur blockweise zwischen Mapper und Reducer transferieren. Außerdem werden die Zwischenergebnisse auch noch auf die Festplatte abgelegt, bevor sie weiter verarbeitet werden. Als Resultat haben wir die oben aufgeführten Laufzeiten und starte Korrelationen zu Dateieinanzahl.

Stoppwörter

Für den Stoppworttest, wurden manuell überschaubare Texte in englischer und deutschen Sprachen erstellt und analysiert. Bild 13 illustriert die Dokumentenstruktur. Die Texte wurden dann durch Hadoop bearbeitet und das Endergebnis mit dem Erwartungswert abgeglichen.



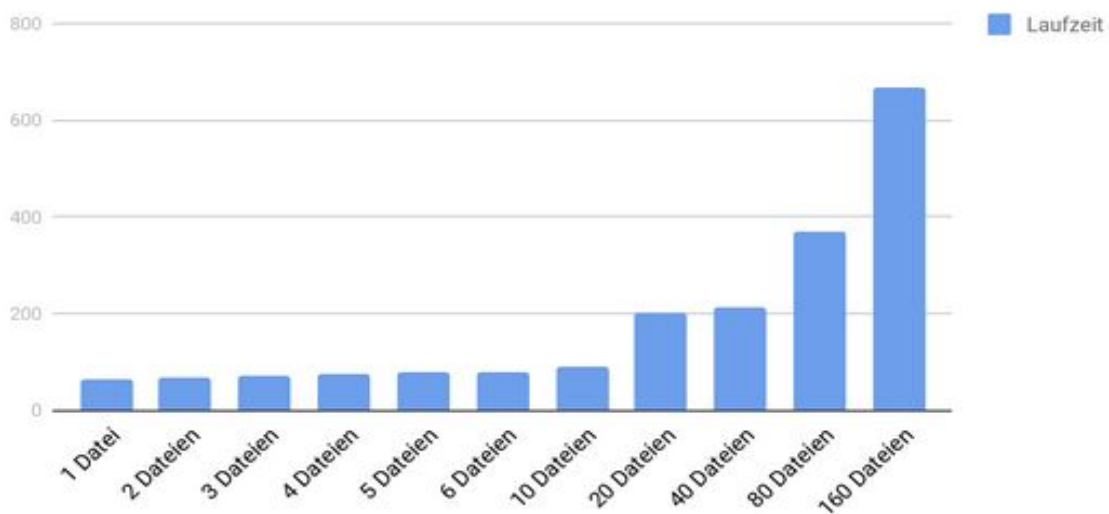
```
C:\Windows\System32\cmd.exe
Y:\bd>tree /f input
Folder PATH listing for volume Yii
Volume serial number is 82B3-56D0
Y:\BD\INPUT
├── english
│   ├── en_1.txt
│   └── en_2.txt
└── german
    └── de_1.txt
Y:\bd>
```

Bild 13: Dokumentenstruktur

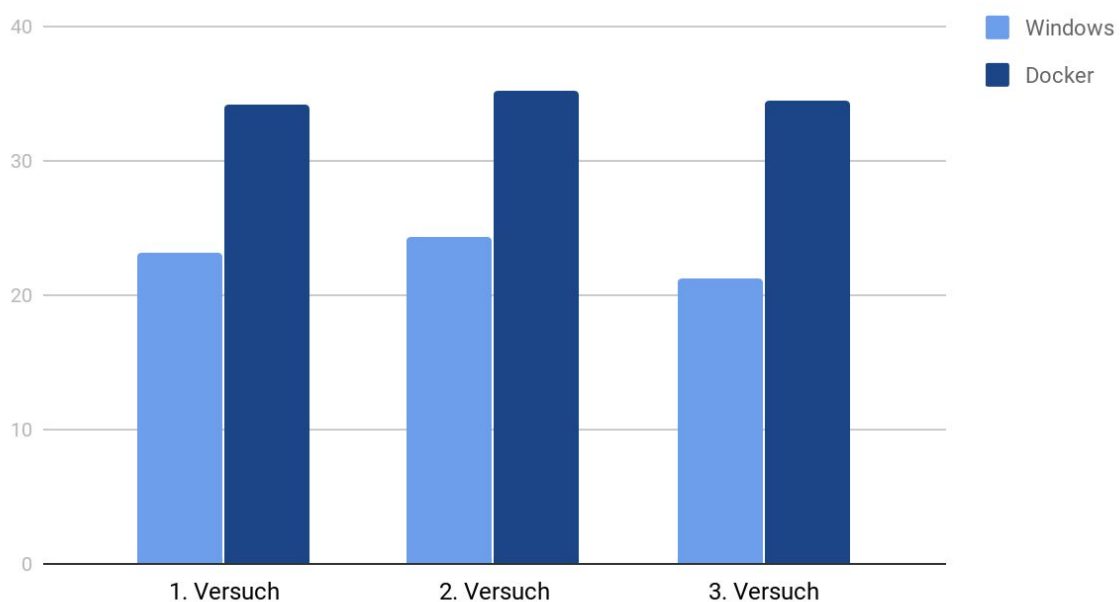
Diagramme für die Laufzeit

Um das Problem mit den leeren Speicherblöcken zu beseitigen, experimentierten wir mit Zusammenschluss von mehreren kleinen Dateien in eine große, um die Blöcke vollständig füllen zu können. Bei allen zur Verfügung gestellten Dateien, das sind 364 Stück, in ihrer Originalstruktur mit Sprachordnern, braucht Hadoop etwa 20 Minuten. Je wenigen Dateien, desto schneller wird die Bearbeitung. Bei einer Datei mit allen Texten und Sprachen drin (135MB), konnte die Laufzeit auf etwa 60 Sekunden gesenkt werden. Es ist immer noch länger als bei parallelisierten Javaversion (20 Sekunden), aber immerhin eine 20-fache Leistungssteigerung gegenüber der Anfangsvariante.

(Einheit : Sekunde)



(Einheit: Minute)



Im zweiten Diagramm wurde die Laufzeit direkt unter Windows mit der Ausführung in Docker verglichen. Dabei ist zu beachten, dass Docker nur einen Prozessor installationsbedingt nutzen konnte. Nichtsdestotrotz verdoppelte sich die Laufzeit nicht. Bei drei Durchläufen brauchte Hadoop unter Docker im Durchschnitt 33 Minuten, während Windows 22 lief. Daraus ergibt sich eine Laufzeitsteigerung um Faktor eineinhalb.

Fazit

Nach diesem Projekt fühlen wir uns mehr vertraut mit Hadoop und Map-Reduce. Auch wenn wir viel herumexperimentieren konnten, war es doch nicht genügend Zeit da, um noch den ersten Ansatz aus der Planungsphase zu implementieren. Das würde einen noch mehr detaillierten Einblick in die Hadoop-Verarbeitung geben.

Der von uns ausgewählte Ansatz, Sprache und Worte in einem einzigen String zu vereinen funktionierte jedoch einwandfrei. Auch eine direkte Ausgabe der Ergebnisse in Hadoop-Context konnte mit Textdatentypen für Wort und Zählerwert umgesetzt werden. Eine einfache Consolenausgabe würde natürlich ebenfalls gehen, wäre aber nicht so elegant gewesen. Das Endergebnis wurde in der Cleanup-Methode aufbereitet. Uns ist später eingefallen, dass zum Stopworte-Einlese eine ähnliche Setup-Methode ganz gut geeignet ist.

In der Auswertung wurde eine direkte Abhängigkeit zwischen Laufzeit und Inputgröße, bzw. Anzahl der zu verarbeitenden Dateien festgestellt. Wir konnten die Laufzeit maßgeblich durch einen Zusammenschluss der Dateien, was zu einer effizienteren Speicherblockausnutzung führt, zwanzigfach verkleinern. Das Endergebnis wurde dabei nicht beeinflusst.

Das zur Vergleichszwecken entwickelte Java-Programm war mit der Verarbeitung in 20 Sekunden der Hadoop-Version eindeutig überlegen, aber leistete keine vergleichbare Ausfallsicherheit. Ein weiterer, für Java günstiger Grund war die Inputgröße, die keinesfalls im Big Data Bereich lag. Bei einer Inputgrößenverschiebung in Tera- oder Petabytebereich ist eine andere Konstellation zu erwarten. Ungeachtet dessen, konnten sich Java-Streams mit entsprechenden Bearbeitungsfunktionen wie Map und Filter ganz gut behaupten. Damit lässt sich der Code ganz knapp halten, man muss aber die Denkweise etwas umstellen.

Im Dokument ist es nicht reflektiert, aber die Hadoop-Installation unter Windows ist aufwendig. Besonders die Cluster-Konfiguration erfordert viel Geduld und Aufmerksamkeit, denn ein falsch gesetzter Flag kann ganz schnell zu unerwarteten Komplikationen führen. Das Ganze wird mit Docker sehr vereinfacht: Image laden, starten und es kann los gehen. Um uns in der Zukunft besser an das Deployment zu erinnern, fassen wir in Anhang die wichtigsten Schritte zusammen.

Anhang

JAR Erstellen mit Eclipse

- Klicke mit der rechten Maustaste auf den Projekt und auf „Exportieren“

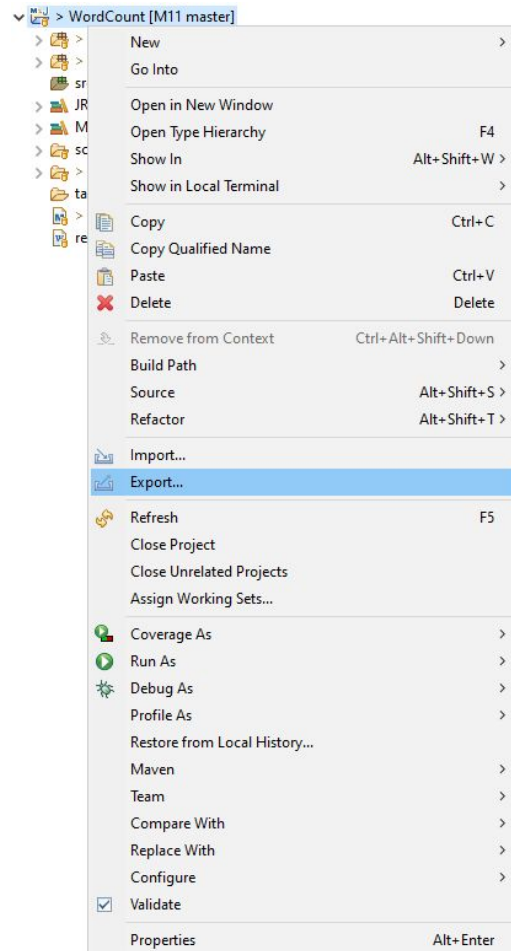


Abbildung 15: aus Eclipse exportieren

- Erweitere den „Java“-Ordner und mache einen Doppelklick auf die Option „Ausführbare Jar-Datei“. (Abbildung 16)

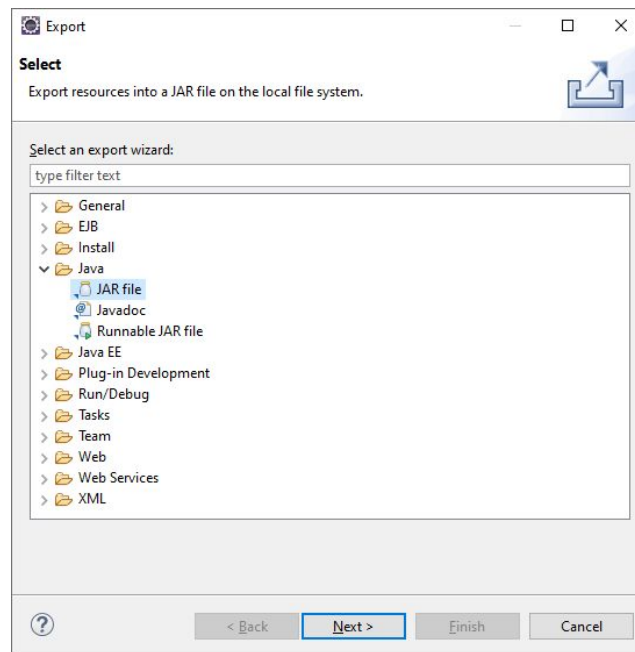


Abbildung 16: Die Option „Ausführbare Jar-Datei“.

- Wähle die folgenden Optionen für den Export und den Speicherordner.(Abbildung 17)

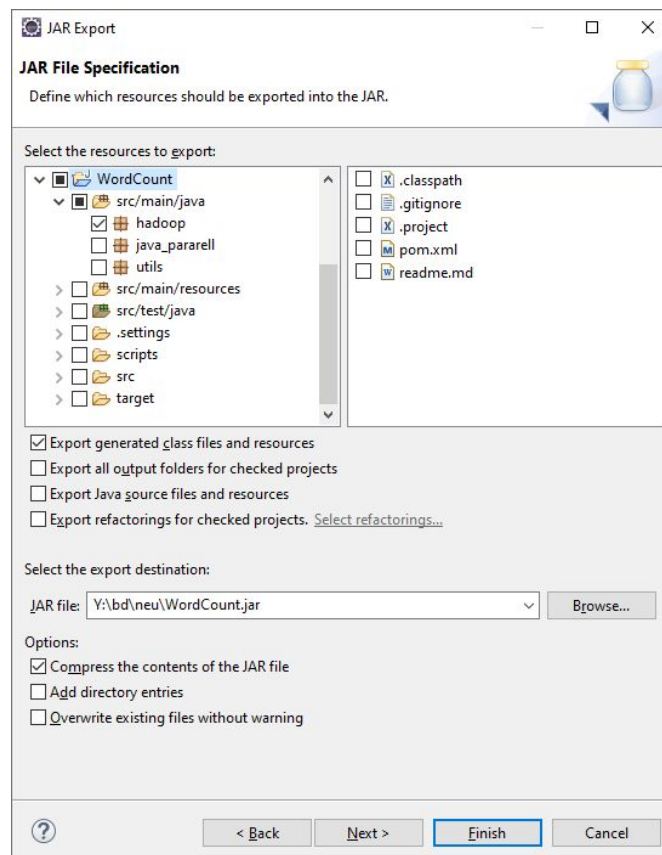


Abbildung 17: Wähle die folgenden Optionen für den Export und den Speicherordner

- Klicke „Weiter“, wähle die Hauptklasse und klicke „Finish“.

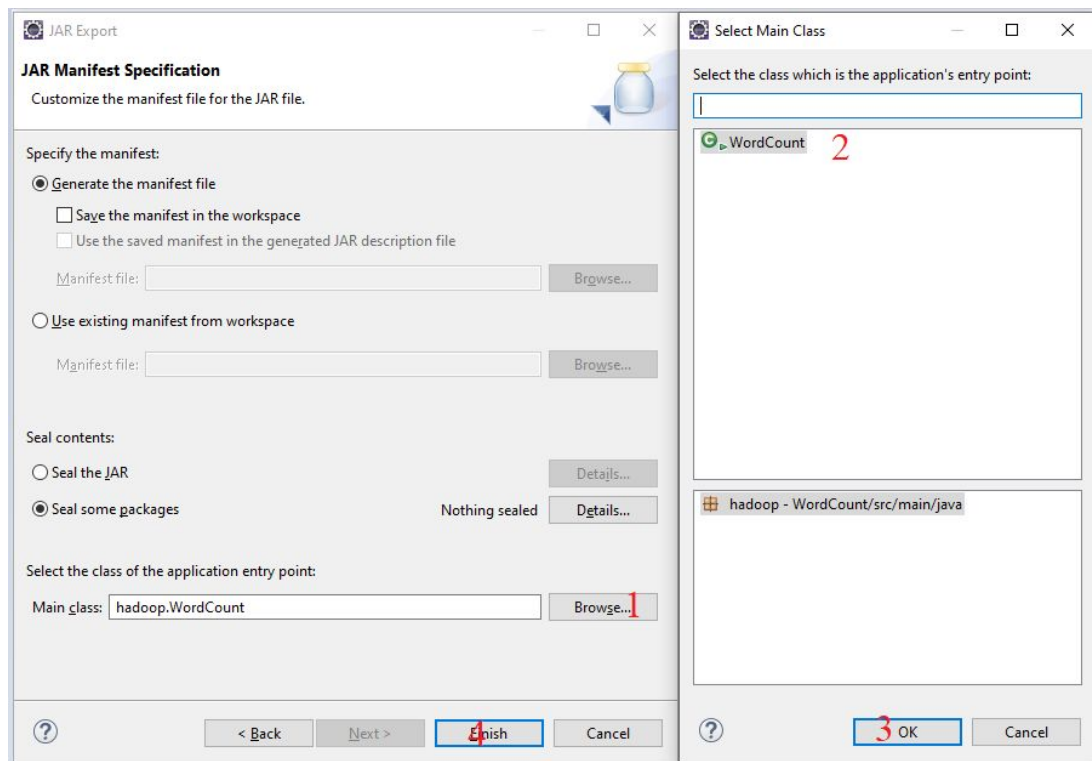


Abbildung 18: Wähle der Hauptklasse.

Wir könnten das Programm entweder unter Docker oder unter Windows durchführen.

Docker

Zum Installieren von Docker unter Windows/Mac/Linux können Sie hier <https://www.docker.com/get-started> zugreifen und dann Docker-Anwendung herunterladen (Abbildung 19). Danach installieren Sie es.

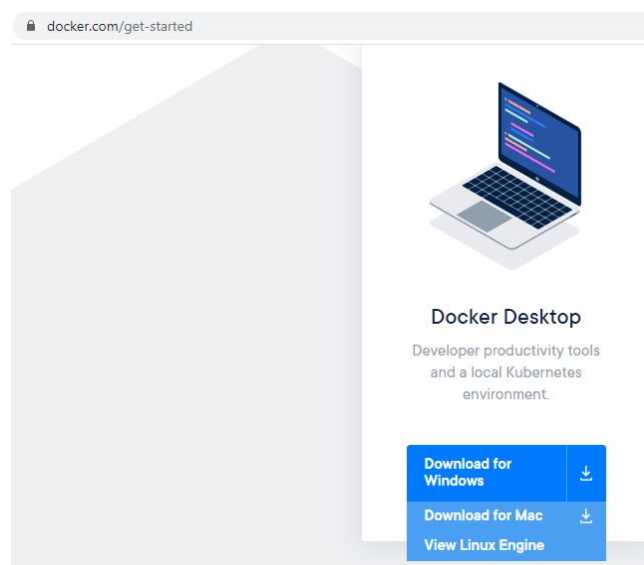


Abbildung 19: Docker-Download Seite

Nach der Installation starten wir Docker-Anwendung. Die Abbildung 20 zeigt, dass Docker schon fertig gestartet wurde.

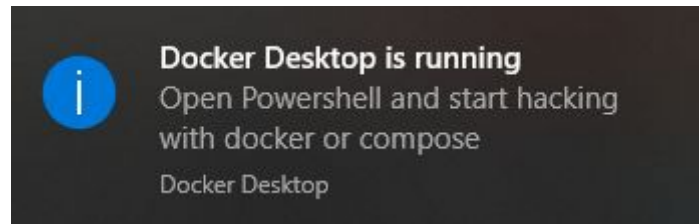


Abbildung 20: Docker ist fertig gestartet.

Greifen wir in die Einrichtung zu. (Abbildung 21)

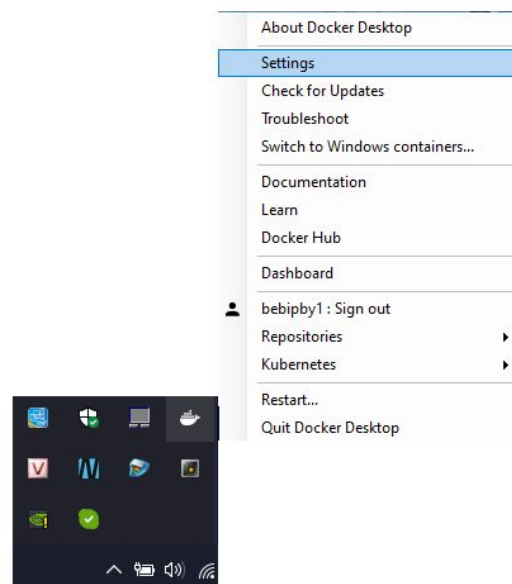


Abbildung 21: Zugriff der Einrichtung von Docker.

Wir passen Docker an unsere Maschinenkonfiguration an. Achten Sie darauf, dass die folgende Abbildung 22 minimale Einrichtung ist.

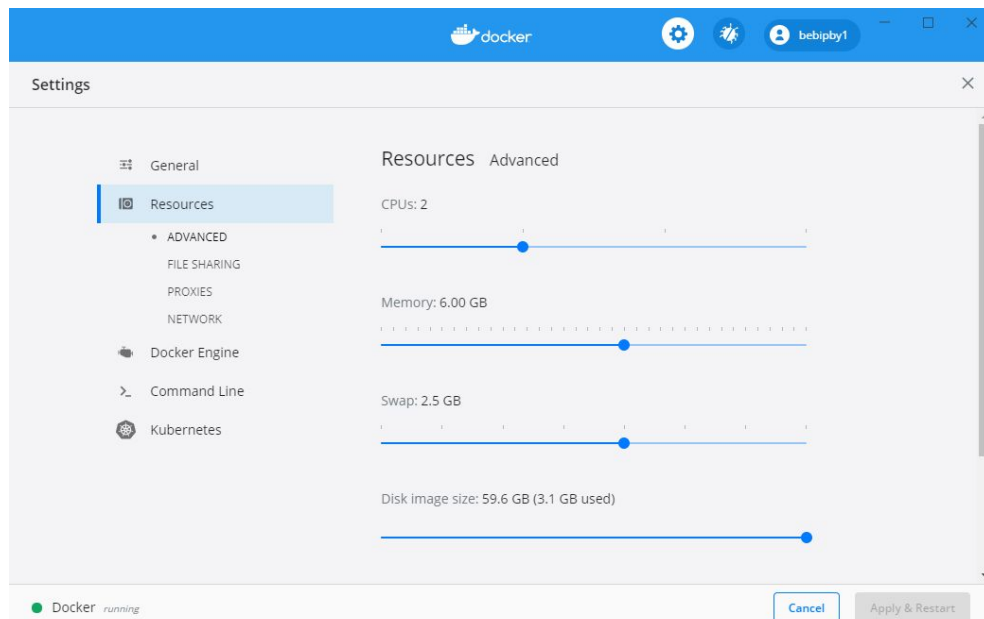


Abbildung 22: Einrichtung der Maschinenkonfiguration von Docker.

Wir benutzen die Eingabeaufforderung (command prompt), um Hadoop unter Docker zu installieren.

- Wir benutzen Git, um Hadoop-Docker herunterzuladen.
[git clone https://github.com/big-data-europe/docker-hadoop.git](https://github.com/big-data-europe/docker-hadoop.git)
- Nach der fertigen Herunterladung gehen wir zum Ordner „docker-hadoop“ (Abbildung 23).

```

C:\Users\nguye>cd docker-hadoop

C:\Users\nguye\docker-hadoop>dir
Volume in drive C: has no label.
Volume Serial Number is 7834-9871

Directory of C:\Users\nguye\docker-hadoop

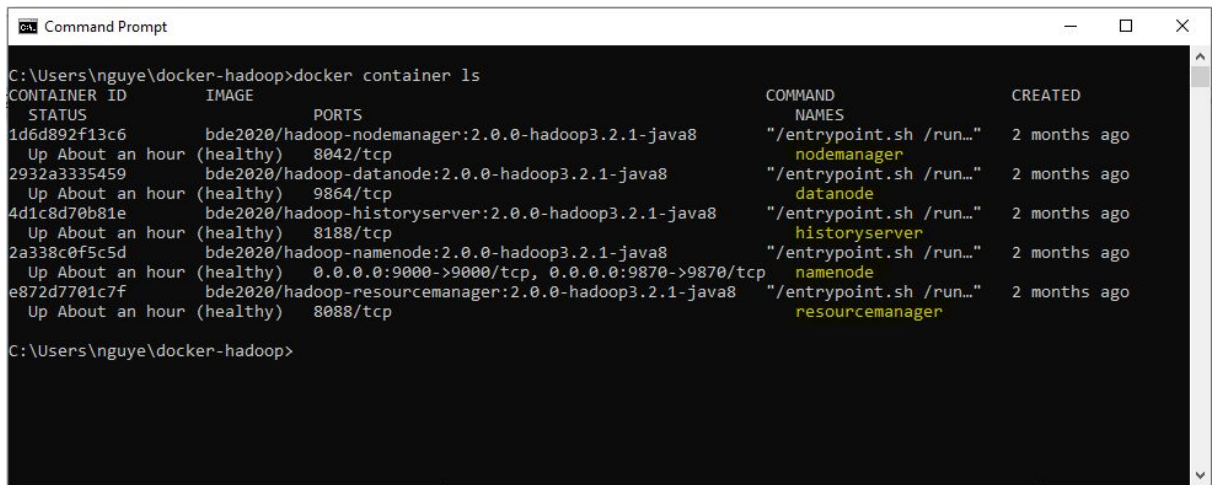
26/04/2020  04:36  <DIR>          .
26/04/2020  04:36  <DIR>          ..
26/04/2020  04:36              7 .gitignore
26/04/2020  04:36  <DIR>          base
26/04/2020  04:36  <DIR>          datanode
26/04/2020  04:36              2.632 docker-compose-v3.yml
26/04/2020  04:36              1.620 docker-compose.yml
26/04/2020  04:36              2.550 hadoop.env
26/04/2020  04:36  <DIR>          historyserver
26/04/2020  04:36              1.457 Makefile
26/04/2020  04:36  <DIR>          namenode
26/04/2020  04:36  <DIR>          nginx
26/04/2020  04:36  <DIR>          nodemanager
26/04/2020  04:36              2.234 README.md
26/04/2020  04:36  <DIR>          resourcemanager
26/04/2020  04:36  <DIR>          submit
                6 File(s)          10.500 bytes
                10 Dir(s)  1.162.264.576 bytes free

C:\Users\nguye\docker-hadoop>

```

Abbildung 23: Der Ordner von docker-hadoop

- Dann benutzen wir den Code „[docker-compose up -d](#)“, um Hadoop zu installieren.
- Zum checken, ob Hadoop abgeschlossen installiert ist oder nicht, benutzen wir den Code „[docker container ls](#)“. Die Abbildung 24 zeigt, dass alles in Ordnung war.



```
C:\Users\nguye\docker-hadoop>docker container ls
CONTAINER ID        IMAGE                                     PORTS                COMMAND NAMES          CREATED
1d6d892f13c6       bde2020/hadoop-nodemanager:2.0.0-hadoop3.2.1-java8 8042/tcp             "/entrypoint.sh /run..." 2 months ago
Up About an hour   (healthy)
2932a3335459       bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8    9864/tcp             "/entrypoint.sh /run..." 2 months ago
Up About an hour   (healthy)
4d1c8d70b81e       bde2020/hadoop-historyserver:2.0.0-hadoop3.2.1-java8 8188/tcp             "/entrypoint.sh /run..." 2 months ago
Up About an hour   (healthy)
2a338c0f5c5d       bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8    0.0.0.0:9000->9000/tcp, 0.0.0.0:9870->9870/tcp "/entrypoint.sh /run..." 2 months ago
Up About an hour   (healthy)
e872d7701c7f       bde2020/hadoop-resourcemanager:2.0.0-hadoop3.2.1-java8 "/entrypoint.sh /run..." 2 months ago
Up About an hour   (healthy)

C:\Users\nguye\docker-hadoop>
```

Abbildung 24: Hadoop wurde abgeschlossen installiert.

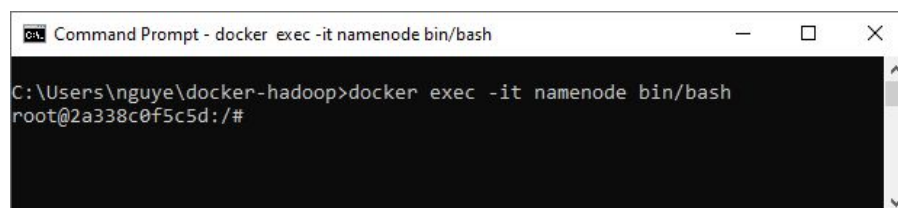
Zum Installieren von Hadoop unter Windows ist es verfügbar im Moodle :
https://www.dropbox.com/s/0oiamvxgxdjl89n/Setting_up_on_Windows.pdf

Ausführung des Programms

1. Docker

Vorbereitung der Eingaben.

- + Kopiere den Ordner von Textdateien „input“ in den Ordner tmp.
`docker cp ./input namenode:/tmp`
- + Kopiere den Ordners von Textdateien „stopwords“ in den Ordner tmp.
`docker cp ./stopwords namenode:/tmp`
- + Kopiere die „WordCount.jar“ Datei in den Ordner tmp.
`docker cp ./WordCount.jar namenode:/tmp`
- + Zugriffe in Hadoop.
`docker exec -it namenode bin/bash`



```
C:\Users\nguye\docker-hadoop>docker exec -it namenode bin/bash
root@2a338c0f5c5d:/#
```

Abbildung 25: Zugriffe in Hadoop.

- + Wir fügen einen Ordner root hinzu und kopieren den Daten in den HDFS.
`cd tmp`
`hdfs dfs -mkdir -p /user/root`

```
hdfs dfs -put input ./
hdfs dfs -put stopwords ./
```

- Führe das Programm aus.
`hadoop jar WordCount.jar input output`

2. Windows

- Wir benutzen die als Administrator geöffnete Eingabeaufforderung (command prompt), um Hadoop unter Windows auszuführen.

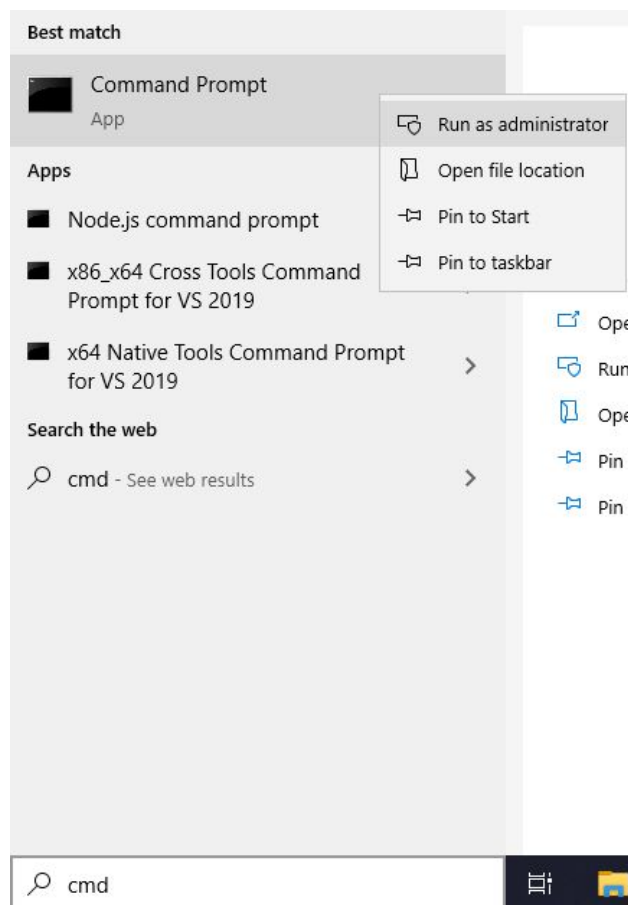


Abbildung 26: die als Administrator geöffnete Eingabeaufforderung.

- Gebe den Code „`start-all.cmd`“ auf die Eingabeaufforderung an, um Hadoop zu starten.
- Vorbereitung der Eingaben.
- + Kopiere den Ordner von Textdateien „input“ in den Ordner tmp.
`docker cp ./input namenode:/tmp`
- + Kopiere den Ordner von Textdateien „stopwords“ in den Ordner tmp. Achte darauf, dass **nguyen** den Name meines Kontos ist.
`hdfs dfs -put ./stopwords \user\nguyen\`

- Führe das Programm aus
`hadoop jar ./WordCount.jar /input /output`

3. Java

- Erweitere das „java_pararell“-Paket und wähle die „Main.java“ Datei (Abbildung 26), um das Programm auszuführen (Abbildung 27).

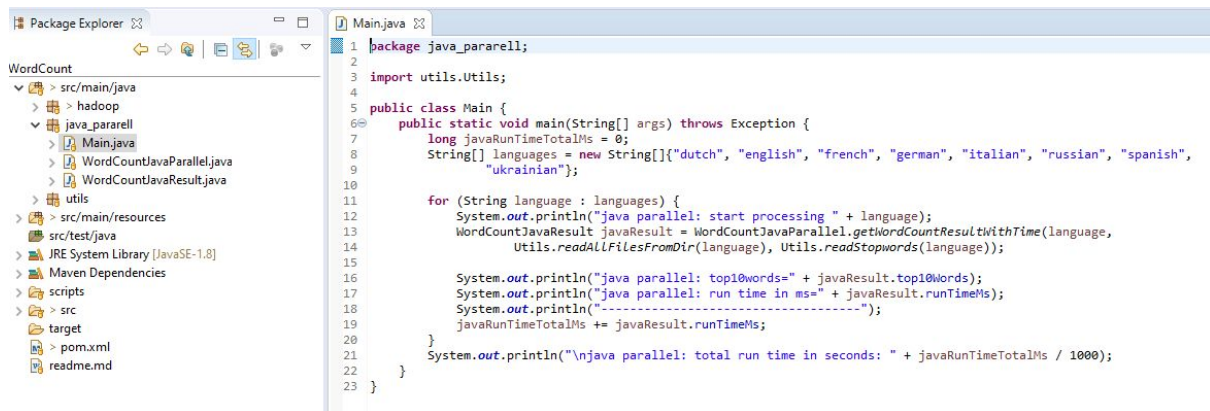


Abbildung 27: Die „Main.java“ Datei.

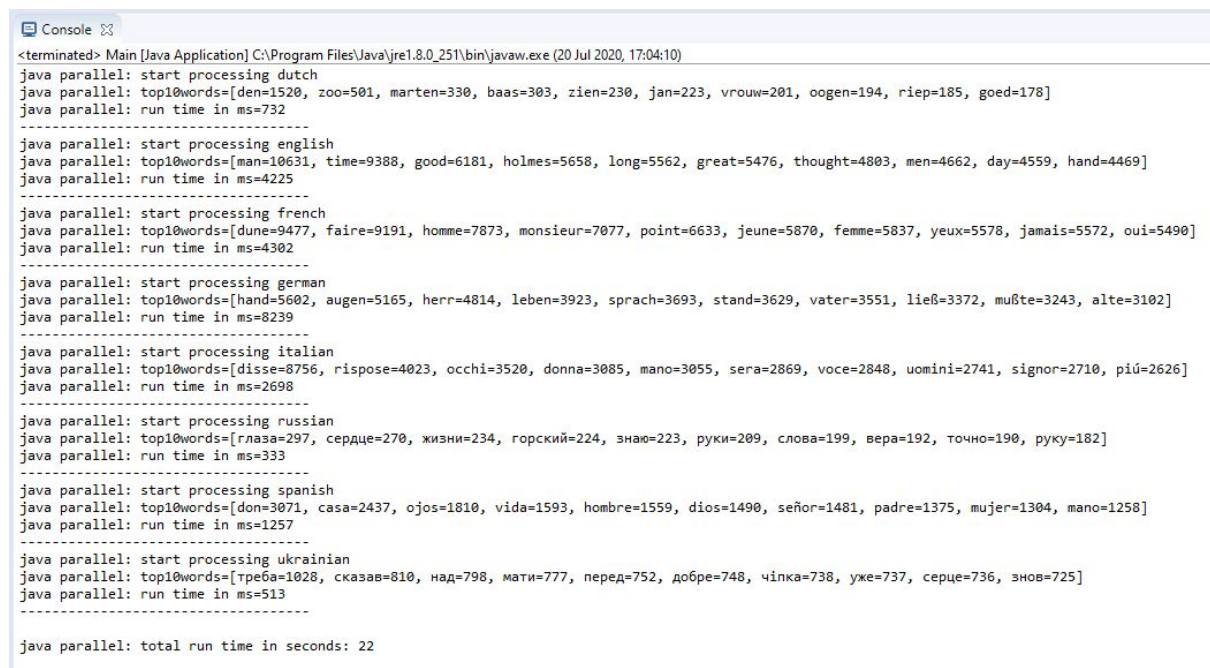


Abbildung 28: Das Ergebnis des Programms.