Hadoop Sprachvergleich

Release 1.0.0

25.06.2019

Inhalt:

KAPITEL 1

Aufgabenstellung

Dies ist eine Belegaufgabe im Rahmen unseres Studiums an der HTW Berlin im Master Angewandte Informatik. Im Kurs Programmierkonzepte & Algorithmen arbeiten wir an Aufgabe 11. Es liegt ein Datensatz von 502 Textdateien vor (Beletristik) - unterteilt in 8 Sprachen. Ziel ist es, mit Apache Hadoop diese Texte zu analysieren und dabei drei Aufgaben zu erfüllen:

1. Zählen der Wörterlänge (pro Sprache)
2. Sortieren der Wörter der Länge nach (pro Sprache)
3. Zusammenfassen der Ergebnisse

Dabei soll eine Ausgabe in der Form: „Sprache – Längstes Wort – Länge“ generiert werden.

1.1 Dokumentationsanforderungen

Anforderung an Dokumentation sind mindestens 10 Seiten und sie muss mindestens die folgenden Bestandteile enthalten:

- Kurze Beschreibung und Erklärung der Aufgabe
- Detaillierte Lösungsbeschreibung
- Code-Fragmente mit einer Textbeschreibung
- Screenshots für die Ergebnisse und/oder Zwischenergebnisse
- Ausführliche Tests der Anwendung
- Tabellen, Graphen und Diagramme für die Leistung und vergleichende Laufzeit
- Kurzes Fazit

1.2 Lösungsansatz

Hadoop stellt vorrangig vier Dinge bereit:

1. Hadoop distributed file system (HDFS) - ein Verteiltes Dateisystem
2. Hadoop MapReduce - ein System für parallele Datenverarbeitung
3. Hadoop YARN - ein Framework um zur Job-Ablaufplanung (scheduling) und Cluster Ressourcen Management
4. Hadoop Common - alle Werkzeuge die das Kommunizieren dieser drei untereinander ermöglichen

Wir arbeiten dabei besonders eng mit HDFS und MapReduce zusammen.

Mittels Docker starten wir einen Single-Node Cluster. Auf diesen können wir durch HDFS (Hadoop distributed file system) automatisch alle Textdateien (.txt-Format) verteilt speichern. Außerdem können Jobs ausgeführt werden, um die Dateien zu verarbeiten.

Wir starten pro Sprache einen Job der den MapReduce Prozess ausführt. Dieser findet für eine Sprache das längste Wort und speichert dieses in einer Part-Datei.

KAPITEL 2

Installation

Das Aufsetzen von Hadoop gestaltete sich als sehr schwierig. Sowohl auf Windows, als auch auf Linux und Mac gab es unterschiedliche Probleme. Verschiedene Anleitungen & Hadoop-Versionen wurden ausprobiert. Auch Stackoverflow und andere Foren konnten die Probleme nicht vollständig beseitigen.

Auf dem Mac entstanden viele Fehler durch Berechtigungsprobleme beim SSH Zugriff auf localhost. Zu einem Zeitpunkt funktionierte der Hadoop-Cluster teilweise. Es gibt wohl öfter Probleme beim Starten und Beenden von YARN.

Auf Linux konnte Hadoop zwar vermeintlich installiert werden, die Beispiel-JAR's erreichen beim Testen dann jedoch nicht alle Nodes.

Aus diesen Gründen wurden sich gegen eine einfache Lokale installation und für eine Containerisierungslösung entschieden. So können auch Sie schneller und verlässlicher Testen.

2.1 Quickstart

Note: In dieser Dokumentation markieren wir jeden Komandozeilenausschnitt mit **Local** oder **Docker** um zu verdeutlichen ob die Befehle für das Hostsystem oder innerhalb des Docker Containers ausgeführt werden.

Unser Dockerfile basiert auf dem Hadoop Docker Image von sequenceiq, befindet sich in `./Docker/Dockerfile` und kann wie folgt gebaut und ausgeführt werden:

Local:

```
docker build -t sv .
docker run -it sv /etc/bootstrap.sh -bash --name docker_hadoop
```

Um den Docker Container zu testen kann das mitgelieferte Beispiel wie ausgeführt und dessen Ergebnisse ausgelesen werden.

Docker:

```
cd $HADOOP_PREFIX
bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.0.jar grep input␣
↪output 'dfs[a-z.]+'
bin/hdfs dfs -cat output/*
```

2.1.1 Ausführung vorbereiten

Quick & Dirty commands on Linux & Mac:

Local:

` docker ps cd <Project> ./createAndCopyJAR.sh <container_id> docker exec -it <container_id> /bin/bash `

Docker: ` cd /hadoop_sv/ ./createEnvironment.sh runhadoop `

Resourcen auf den Container bringen

Die Textdateien müssen in das DFS Dateisystem kopiert werden. Dabei sind folgende Schritte nötig:

- Hostmaschine Docker (dieser Schritt wird automatisch durch das Dockerfile ausgeführt)
- Docker DFS (dieser Schritt muss wie unten beschrieben ausgeführt werden)

In `./Docker/` befindet sich die Input datei (`textfiles.zip`) und eine kleinere Testdatei (`textfiles_mini.zip`). Diese enthalten Beispiel-Text-Dateien, die analysiert werden sollen. Diese Dateien werden Automatisch durch das Dockerfile in den Docker Container kopiert. Um andere Dateien zu testen kann das Dockerfile bearbeitet werden.

Falls man sich noch nicht in bash des Containers befindet holt man sich die container-id mit `docker ps`, und kopiert sie in folgenden Befehl:

Local:

```
docker exec -it <docker container_id> /bin/bash
```

Danach kann das Archiv auf das verteilte Hadoop-Dateisystem (HDFS) hochgeladen werden:

Docker:

```
$HADOOP_PREFIX/bin/hdfs dfs -mkdir /hadoop_sv
$HADOOP_PREFIX/bin/hdfs dfs -put /hadoop_sv/textfiles /hadoop_sv/
```

JAR-File Kompilieren und in Container kopieren

Da wir ein Maven-Projekt benutzen, muss nach der Implementierung das ‚.jar' file kompiliert, ggf. umbenannt und auf den Docker-Container kopiert werden. Dafür wurde das Skript `create_and_copyJAR.sh` geschrieben:

1.a: Mit Hilfe von script

Local:

```
create_and_copyJAR.sh <containerId>
```

1.b: anuell: ***Alternativ*** **kann die Maven .jar manuell erzeugt und in den** Container kopiert werden:

Local:

```
cd hadoop_sv
mvn clean package
mv target/Hadoop_sv-1.0-SNAPSHOT.jar target/hadoop_sv.jar
docker cp target/hadoop_sv.jar <containerId>:/hadoop_sv
```

2. In jedem Fall muss die .jar Datei danach vom Docker Container auf das HDFS System kopiert werden:

Docker:

```
$HADOOP_PREFIX/bin/hdfs dfs -put /hadoop_sv/hadoop_sv.jar /hadoop_sv
```

2.1.2 Hadoop-Job ausführen

Um den Hadoop-Job zu starten wird folgender Befehl ausgeführt (beim Starten des Docker-Containers sollte auch ein Alias angelegt worden sein. Damit lässt sich der lange Befehl auch mit `runhadoop` ausführen.):

Docker:

```
$HADOOP_PREFIX/bin/hadoop jar /hadoop_sv/hadoop_sv.jar de.berlin.htw.Hadoop_sv /
↪hadoop_sv/textfiles /hadoop_sv/output/ /hadoop_sv/results/
```

2.1.3 Ergebnisse sichten

Die Ergebnisse liegen jetzt in `hadoop_sv/output` und können direkt angezeigt werden:

```
$HADOOP_PREFIX/bin/hdfs dfs -cat /hadoop_sv/output/part-r-00000
```

Oder die Dateien können in zwei Schritten auf das Hostsystem kopiert werden:

1. HDFS –> Docker

Um die Ergebnisse vom HDFS auf den Container zu kopieren kann auch der alias `copyresults` verwendet werden.

Docker:

```
$HADOOP_PREFIX/bin/hdfs dfs -get /hadoop_sv/output /hadoop_sv/
$HADOOP_PREFIX/bin/hdfs dfs -get /hadoop_sv/results /hadoop_sv/
```

2. Docker –> Hostmaschine

Local:

```
docker cp <containerId>:/hadoop_sv/output ~/Desktop/
docker cp <containerId>:/hadoop_sv/results ~/Desktop/
```

KAPITEL 3

Nützliche Befehle

Mit HDFS cluster interagieren: `$HADOOP_PREFIX/bin/hdfs dfs -ls /`

outout löschen: `$HADOOP_PREFIX/bin/hdfs dfs -rm -r /hadoop_sv/output`

output anzeigen `$HADOOP_PREFIX/bin/hdfs dfs -cat /hadoop_sv/output/part-r-00000`

get files (HDFS to Docker): `$HADOOP_PREFIX/bin/hdfs dfs -get /hadoop_sv/output / hadoop_sv/`

get files (Docker to Local): `docker cp <containerId>:/hadoop_sv/output ~/Desktop/`

KAPITEL 4

Umsetzung / Implementierung

4.1 Entwicklerumgebung

Mittels IDE oder Konsole kann ein Maven-Projekt erstellt werden, in welchem die Dependencies `hadoop-core` und `hadoop-mapreduce-client-core` eingebunden werden müssen.

4.1.1 Struktur

```
├── java
│   └── de
│       └── berlin
│           └── htw
│               ├── Hadoop_sv.java
│               ├── SV_Reducer.java
│               └── mappers
│                   ├── AggregationMapper.java
│                   ├── RegexMapper.java
│                   └── ReplaceMapper.java
├── python
│   └── python-sprachvergleich.py
└── resources
```

Hauptklasse: **Hadoop_sv.java**

- Umfasst Konfiguration (Drei Variablen als Argumente: Input, Output & Results)
- Ausführen der verschiedenen Jobmethoden
- Konsolenausgaben zum nachvollziehen des Fortschritts
- Stoppt die Zeit zum Ausführen der verschiedenen Jobs

Mapper:

RegexMapper/ReplaceMapper:

- Findet in jeder Datei das längste Wort

AggregationMapper

- Findet das längste Wort von jeder Sprache

Reducer: **SV_Reducer**

- Fässt alle langen Worte einer Sprache zusammen

KAPITEL 5

Quellcodebeschreibung

In diesem Dokument soll der Quellcode genau beschrieben werden.

5.1 Hadoop_sv

In dieser Datei startet Hadoop.

5.1.1 Main

Dies ist der Einstiegspunkt in das Programm.

Zum messen der Ausführungszeit nutzen nutzen wir `System.currentTimeMillis()`.

```
long totalStart = System.currentTimeMillis();
```

Zuerst wird die Angabe der nötigen argumente überprüft und - wenn erfolgreich - gespeichert.

```
if(args.length != 3)
{
    System.out.println("Please provide the following paths as arguments (3): 'input-
↪path', 'output-path' & 'result-path'.");
    System.exit(1);
}
if (args[0].equals("") || args[1].equals("") || args[2].equals(""))
    System.exit(1);

// input folder should have the following structure:
// <rootPath>/<languageDirectories>/TXT/<txt-files>
String rootPath = args[0];
String destinationPath = args[1];
String resultPath = args[2];
File[] languageDirectories = new File(rootPath).listFiles();
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
int languageProgress = 0;
int languages = languageDirectories.length;
```

Danach erstellen wir eine Konfiguration für hadoop

```
Configuration conf = new Configuration();
```

Jetzt kann man entweder einen Job für alle Sprachpakete ausführen:

```
startJobToCountWordLengthForAFolder(conf, new File(rootPath), destinationPath);
```

oder für jede Sprache einen eigenen Job Ausführen (für Multi-Node Systeme sinnvoll). Wichtig ist die for-Schleife mit der Funktion `startJobToCountWordLengthForAFolder`. Währendessen werden stets Debug informationen ausgegeben.

```
debugMessage(String.format("Looping through subdirectories of rootfolder: '%s'",␣
↪rootPath), "DEBUG");
for (File language : languageDirectories) {
    long start = System.currentTimeMillis();

    debugMessage(String.format("Current state: '%s/%s' languages counted.",␣
↪languageProgress, languages), "DEBUG");
    startJobToCountWordLengthForAFolder(conf, language, destinationPath);
    languageProgress += 1;

    long timeElapsed = (System.currentTimeMillis() - start)/1000;
    debugMessage(String.format("Time Elapsed for this language: %ds", timeElapsed),
↪"TIMER");
}

long totalTimeElapsed = (System.currentTimeMillis() - totalStart)/1000;
debugMessage(String.format("Time Elapsed for all Jobs: %ds", totalTimeElapsed), "TIMER
↪");

// use output of first jobs and save to resultPath
debugMessage(String.format("Aggregating results to %s", resultPath), "DEBUG");

//data is located at: /hadoop_sprachvergleich/output/<language>/part-r-00000
```

Zuletzt werden die PartFiles (Je Sprache ein Part-File mit einem Eintrag für jede .txt Datei dieser Sprache) aggregiert mit `aggregatePartFiles`

```
aggregatePartFiles(conf, destinationPath, resultPath);
debugMessage(String.format("Finished aggregating longest words for all languages.␣
↪Results are under: %s", resultPath), "DEBUG");
```

5.1.2 startJobToCountWordLengthForAFolder

Diese Funktion Startet die Haupt-Jobs mit dem MapReducer um die größten Wörter zu sammeln.

Zuerst kommen ein paar Debug Nachrichten:

```
String currentLanguage = languageFolderFile.getName();
debugMessage(String.format("Start counting words for language folder: '%s''",␣
↪currentLanguage), "DEBUG");
```

Dann wird der Job erstellt und die Konfigurationen eingestellt. Es wird unsere Mapper- und Combiner-Class und die Reduce Menge auf 1 gesetzt. Da das Key-Value Pair ein „long-Text“ darstellt setzen wir `OutputKey` und `OutputValue` entsprechend.

```
Job job = Job.getInstance(conf, "Word length counting for language: ");
job.setJarByClass(Hadoop_sv.class);
job.setMapperClass(RegexMapper.class);
job.setCombinerClass(SV_Reducer.class);
job.setReducerClass(SV_Reducer.class);
job.setNumReduceTasks(1);
job.setOutputKeyClass(LongWritable.class);
job.setOutputValueClass(Text.class);
```

Um die Keys dann absteigend zu sortieren, sodass der Länge Eintrag oben zu finden ist, setzen wir noch folgenden Wert:

```
// sort keys
job.setSortComparatorClass(LongWritable.DecreasingComparator.class);
```

Wir geben an, die Verzeichnisse Rekursiv zu durchsuchen (sodass auch alle Unterverzeichnisse nach .txt Dateien untersucht werden)

:: // loop through all subdirectories recursively FileInputFormat.setInputDirRecursive(job, true);

Wir setzen die In- und Outputpfade unseres Jobs:

```
// set input & output paths
FileInputFormat.addInputPath(job, new Path(languageFolderFile.getAbsolutePath()));
FileOutputFormat.setOutputPath(job, new Path(destinationPath + currentLanguage));
```

Gegebenenfalls muss noch der alte Output ordner gelöscht werden.

```
// Delete output filepath if already exists
FileSystem fs = FileSystem.get(conf);
if (fs.exists(new Path(destinationPath + currentLanguage))) {
    fs.delete(new Path(destinationPath + currentLanguage), true);
}
```

Der Job wird gestartet:

```
job.waitForCompletion(true);

debugMessage(String.format("DEBUG: Completed counting for language folder: '%s'.",␣
↪currentLanguage), "DEBUG");
```

5.1.3 aggregatePartFiles

Hier wird ein Job angelegt um die PartFiles zu aggregieren. Je Sprache gibt es ein PartFile. In jedem Part file befindet sich je eine Zeile für jede .txt Datei mit dem jeweils längsten Wort dieser Datei. Diese werden in der durch den in aggregatePartFiles aufgesetzten Job zusammengefasst.

Wir holen uns die Configurierte Instanz und setzen für unseren Aggregierungsjob unseren Mapper `AggregationMapper`. Für den Reducer reicht der `Reducer.class` (Oberklasse). Wir möchten dabei nur einen Reduce Task ausführen.

```
Job aggregateJob = Job.getInstance(conf, "Aggregating longest words of different␣
↪languages");
aggregateJob.setJarByClass(Hadoop_sv.class);
aggregateJob.setMapperClass(AggregationMapper.class);
aggregateJob.setReducerClass(Reducer.class);
aggregateJob.setNumReduceTasks(1);
```

Da wir die Key-Value Paare für die Ausgabe als jeweils als Text erwarten setzen wir die entsprechende Werte.

```
aggregateJob.setOutputKeyClass(Text.class);
aggregateJob.setOutputValueClass(Text.class);
```

Wir setzen die durch die Argumente bereitgestellten Input und Output Pfade für unseren `aggregateJob` und geben an, die Verzeichnisse Rekursiv zu durchsuchen (sodass auch alle Unterverzeichnisse nach .txt Dateien untersucht werden)

```
FileInputFormat.addInputPath(aggregateJob, new Path(inputPath));
FileOutputFormat.setOutputPath(aggregateJob, new Path(outputPath));
FileInputFormat.setInputDirRecursive(aggregateJob, true);
```

Zum schluss sorgen wir dafür, dass das Ausgabeverzeichniss gelöscht wird sofern dieses bereits existiert und wir starten unseren Job.

```
// Delete output filepath if already exists
FileSystem fs = FileSystem.get(conf);
if (fs.exists(new Path(outputPath))) {
    fs.delete(new Path(outputPath), true);
}

aggregateJob.waitForCompletion(true);
```

5.1.4 debugMessage

Dies Funktion formatiert lediglich Debug Nachrrichten, damit sie eindeutig von den Hadoop-Internen Nachrrichten zu unterscheiden sind.

```
String spacing = "------------------------------------";
System.out.println(String.format("%s %s: %s %s", spacing, type, msg, spacing));
```

5.2 SV_Reducer

Der SV_Reducer wird für jeden Job ein mal Ausgeführt und fasst die Key-Value-Pairs aller Mapper dieses Jobs zusammen.

5.2.1 reduce

Wir prüfen ob ein Wort gefunden wurde. Dann iterieren wir über alle Key-Value-Paare und schreiben diese in den Kontext.

```
if (!maximumFound) {
    for (Text t : values) {
        context.write(key, t);
    }
    maximumFound = true;
}
```

5.3 RegexMapper

Der Regex Mapper durchsucht, wie auch der Replace Mapper, eine ganze Datei nach dem längsten Wort.

5.3.1 map

Hier wird mittels der Regular Expression `\w` jedes Wort erfasst. Wichtig ist das Pattern auf `Pattern.UNICODE_CHARACTER_CLASS` zu setzen.

```
Matcher m = Pattern.compile("\\w+", Pattern.UNICODE_CHARACTER_CLASS).matcher(value.
↪toString());
```

Danach können wir die Matches durchsuchen und bei jedem Hit überprüfen ob wir hiermit ein längeres gefunden haben.

```
while (m.find()) {
    String hit = m.group(0);

    if (hit.length() > maxLength) {
        maxLength = hit.length();
        longestWord = hit;
    }
}
```

5.3.2 cleanup

Beim Cleanup schreiben wir nurnoch das Key-Value-Pair in den Entsprechenden Kontext.

```
context.write(new LongWritable(maxLength), new Text(longestWord));
```

5.4 ReplaceMapper

Der Replace Mapper durchsucht, wie auch der Regex Mapper, eine ganze Datei nach dem längsten Wort.

5.4.1 map

Der Ansatz ist anders als bei RegexMapper, da wir nicht alle Wörter mit `\w` finden, sondern mit dem StringTokenizer alle Wörter in dem Dokument trennen.

```
StringTokenizer itr = new StringTokenizer(value.toString());
```

Danach können wir über alle diese Wörter iterieren. Wir entfernen noch jegliche Sonderzeichen und suchen dann das längste wie gewohnt heraus.

```
while (itr.hasMoreTokens()) {
    String currentToken = itr.nextToken().replaceAll("([\\p{Punct}])", "").trim().
↪toLowerCase();
    word.set(currentToken);

    if (word.getLength() > maxLength) {
        maxLength = word.getLength();
        longestWord = word.toString();
    }
}
```

5.4.2 cleanup

Beim Cleanup schreiben wir nurnoch das Key-Value-Pair in den Entsprechenden Kontext.

```
context.write(new LongWritable(maxLength), new Text(longestWord));
```

5.5 AggregationMapper

Der AggregationMapper wird zuletzt ausgeführt und fasst alle Part-Dateien zusammen.

5.5.1 map

In der Map Funktion iterieren wir über alle Zeilen (im Key-Value-Pair) und findet darin das längste Wort pro Sprache.

```
String lines[] = value.toString().split("\\r?\\n");

for (String line : lines) {
    String number = line.split("\\t")[0];
    int length = Integer.parseInt(number);

    if (length > max) {
        max = length;
        longestWord = line;
    }
}
```

5.5.2 cleanup

In dieser Funktion schreiben wir in den Context die Sprache und das Längste Wort welches ebenfalls die Anzahl an Zeichen enthält. Davor holen wir uns noch die Sprache aus dem Dateipfad des aktuellen Kontextes.

```
String[] filepath = context.getInputSplit().toString().split("/");
language = filepath[filepath.length- 2];
context.write(new Text(language), new Text(longestWord));
```

KAPITEL 6

Performance

Wir haben verschiedene Test aufgestellt um die Performance unseres Codes und die von Hadoop zu evaluieren.

6.1 Replace vs Regex

Für das Heraussuchen der längsten Wörter haben wir zwei verschiedene Mapper implementiert und deren Performance verglichen.

Der Replace-Algorithmus schaut sich jedes Wort an, entfernt (die meisten) unerwünschten Symbole und prüft, ob bereits ein längeres Wort gefunden wurde. Durch den RegEx-Algorithmus werden alle Wörter mithilfe des regulären Ausdrucks `\w+` gesucht und auf Länge überprüft.

Die Laufzeit der Algorithmen wurde mit den Testdaten überprüft. Es ist ersichtlich dass der RegexMapper wenige Sekunden schneller ist. Darüber hinaus liefert er allerdings auch sauberere Ergebnisse, da er auch Bindestriche filtert. Deswegen haben wir uns für den RegexMapper in der finalen Implementierung entschieden.

JOB No.	replaceMapper	regexMapper
1	20s	19s
2	27s	26s
3	21s	21s
4	28s	27s
5	25s	25s
6	25s	24s
7	37s	37s
8	25s	25s
Total	**213s**	**209s**

Aufgrund dieser Ergebnisse wird fortan nur noch der RegexMapper genutzt.

6.2 Ausführungsdauer des Sprachvergleichs

Der Sprachvergleich wurde mit zwei verschiedenen Datensätzen getestet. `textfiles.zip` enthält den Vollen Datensatz aller 502 Textdateien, verteilt über die acht Sprachen. `textfiles_mini.zip` ist eine Reduzierte Version, angelehnt an `textfiles.zip`, speziell für Tests ausgelegt. Sie enthält nur 44 Textdateien.

Außerdem wurden die Tests auf folgenden zwei Maschinen durchgeführt:

Feature	**Mac**	**Windows PC**
OS	Mac OS	Windows 10
Prozessor	Intel Core i7-4850HQ	Intel Core i5-4460
RAM	16 GB Dual Channel	16 GB Dual Channel
Festplatte	AHCI SSD	AHCI SSD

Da die Hardware sehr ähnlich ist, werden Testergebnisse erwartet, die nah beieinander liegen.

In den unten stehenden Tabellen ist die Ausführungsdauer der jeweiligen Schritte und der Gesammtdauer dargestellt.

textfiles_mini.zip			
Language	Amount of files	**MacBookPro**	**Windows PC**
Ukrajinska	13	52s	39s
Deutsch	2	33s	21s
Francais	7	40s	28s
Russkyj	2	27s	21s
Espanol	5	33s	26s
Italiano	5	39s	26s
Nederlands	6	40s	27s
English	4	29s	27s
Total	**44**	**297s**	**219s**

textfiles.zip			
Language	Amount of files	**MacBookPro**	**Windows PC**
Ukrajinska	46	116s	99s
Deutsch	50	102s	102s
Francais	50	104s	125s
Russkyj	223	396s	468s
Espanol	25	62s	70s
Italiano	50	123s	108s
Nederlands	5	31s	27s
English	52	148s	119s
Total	**502**	**1085s**	**1126s**

KAPITEL 7

Tests

Um die Funktionalität der „jar“-Datei zu testen, wurde das Shell-Skript `testHadoopSv.sh` geschrieben. Da die Hadoop-Jobs aus einem Docker-Container heraus gestartet werden, war ein Shell-Skript die einfachste Lösung. Dieses Skript wird beim Erstellen des Containers direkt in den Container kopiert. Es benutzt einen Testdatensatz mit nur zwei Sprachen, führt dafür den Hadoop-Sprachvergleich aus und vergleicht die Resultate mit den bekannten Lösungen.

Das Skript wird mit dem Argument ‚0‘ oder ‚1‘ gestartet, um den Sprachvergleich zu starten und danach die Ergebnisse auszuwerten (0), oder nur die Ergebnisse auszuwerten (1).

Demnach wird dann der Sprachvergleich ausgeführt oder nicht. Die Ergebnisse werden dann automatisch vom HDFS in den Container kopiert: `/hadoop_sv/test_output` & `/hadoop_sv/test_results`

Die Auswertung erfolgt nun, indem die zu vergleichenden Werte erst für den Anwender gezeigt werden und danach mithilfe der „assert()“-Methode vergleichen werden. Falls der Vergleich nicht erfolgreich ist, beendet sich das Programm.

Es werden die folgenden Fälle untersucht:

- Anzahl der Eingangsdateien bei „Deutsch“ müssen mit der deutschen Wörteranzahl übereinstimmen (pro Datei wird ein längstes Wort gespeichert).
- Anzahl der Ordner (Sprachen) soll mit der Anzahl der Wörter in den Resultaten übereinstimmen (Zwei Ordner resultieren in zwei längsten Wörtern, pro Sprache eins).
- Das deutsche & spanische Wort werden konrekt mit den Wörtern `himmelherrgottssakkermentische` & `circunstanciadamente` verglichen, da dies die längsten Wörter in den Beispieldateien sind.

KAPITEL 8

Vergleich

Die Ergebnisse sollen hier mit anderen Ausführungen verglichen werden.

8.1 Hadoop - Sprachvergleich ohne Sprachseparierung

Zuerst war die Idee, den Job nicht auf Sprachen aufzuteilen, sondern alle Sprachen zusammen als Input zu wählen. Damit wurden auch etwas schneller Ergebnisse erzielt (996 Sekunden). Hier ist allerdings das Problem, dass uns dadurch die Information verloren geht, zu welcher Sprache ein Wort gehört. Dadurch werden die Wörter direkt vermischt und sind nicht mehr übersichtlich in einer Datei abrufbar.

Die etwas schnellere Ausführung lässt sich dadurch erklären, da das Programm nur einen Job anlegen muss und nicht pro Sprache einen. Im Live-Betrieb würde das Setup auf ein Multi-Node Cluster umgelegt werden. Hier müsste eine deutliche Performance-verbesserung sichtbar werden.

8.2 Python - Sprachvergleich

Ein Python-Skript sollte die gleiche Aufgabe bewältigen:

- Alle Textdateien eines Verzeichnisses finden
- Textdateien durchgehen und längstes Wort herausfinden
- Pro Sprache das längste Wort finden und alle Sprachen zusammenfassen

Das Skript erledigt diese Aufgabe in 22 Sekunden. Die Ausgabe des Skriptes:

```
INFO: Counting words for all txt-files from root-folder: '/Users/d064467/Projects/
↪Hadoop_Sprachvergleich/Docker/textfiles'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Espanol'
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
INFO: Found 25 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Espanol/TXT'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Russkyj'
INFO: Found 223 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Russkyj/TXT'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Francais'
INFO: Found 50 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Francais/TXT'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/English'
INFO: Found 52 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/English/TXT'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Ukrajinska'
INFO: Found 47 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Ukrajinska/TXT'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Italiano'
INFO: Found 50 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Italiano/TXT'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Nederlands'
INFO: Found 5 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Nederlands/TXT'
INFO: Found 0 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Deutsch'
INFO: Found 50 amount of files in subfolder: '/Users/d064467/Projects/Hadoop_
↪Sprachvergleich/Docker/textfiles/Deutsch/TXT'
FINISHED: Needed 22.09651803970337 seconds to count words for 502 files.
INFO: Longest word for each language:
    Espanol - 20 - circunstanciadamente
    Russkyj - 25 -
    Francais - 21 - constitutionnellement
    English - 70 -␣
↪Mekkamuselmannenmassenmenchenmoerdermohrenmuttermarmormonumentenmacher
    Ukrajinska - 20 -
    Italiano - 27 - quattrocentoquarantatremila
    Nederlands - 22 - landbouwgereedschappen
    Deutsch - 34 - eindusendsöbenhunnertuneiunsösstig
```