Spark
Type-Token-Ratio

Programmierkonzepte und Algorithmen WS25/26

Justin Gebert, Ole Lordieck

Hochschule für Technik und Wirtschaft Berlin
Fachbereich 4
Informatik, Kommunikation und Wirtschaft

Inhaltsverzeichnis

1 Aufgabenstellung

Die Aufgabe besteht darin, gegebene Textdaten mithilfe von Spark verteilt zu verarbeiten, um die sprachliche Vielfalt der enthaltenen Texte zu untersuchen. Dazu sollen die Texte zunächst als normale Textdateien eingelesen. Anschließend soll mit Spark die Type-Token-Ratio (TTR) für jede im Datensatz vorkommende Sprache berechnet werden. Die TTR wird mit der Anzahl der einzigartigen Wörter durch die Gesamtzahl aller Wörter berechnet.

$$\text{TTR} = \frac{\text{Anzahl einzigartiger Wörter}}{\text{Anzahl aller Wörter}}$$

Außerdem sollen vor der Analyse alle Stoppwörter entfernt werden, um aussagekräftigere Ergebnisse zu erhalten. Ziel ist es, die sprachliche Vielfalt zwischen den Sprachen zu vergleichen und gleichzeitig die Vorteile verteilter Big-Data-Verarbeitung sichtbar zu machen, indem der Datensatz bei Bedarf künstlich vergrößert wird.

1.1 Ausführen der Lösung

Der Job muss zunächst gebaut werden:

```
mvn clean package
```

Anschließend wird er mit `spark-submit` gestartet. Die Anzahl der Kerne kann über den Master-Parameter konfiguriert werden.
Befehl für 4 Kerne:

```
spark-submit --class TTRJob --master "local[4]" \
    target/prog-alg-1.0.jar data/text data/stopwords results
```

Automatisierte Benchmarks: Für die Leistungsmessung gibt es ein Python-Script (`scripts/run_benchmarks.py`), das den Job mit 1, 2, 4 und 8 Kernen ausführt und die Laufzeiten protokolliert.
Eine detaillierte Anleitung zur Reproduktion der Benchmarks befindet sich in der `README.md` Datei.

2 Lösungsbeschreibung

2.1 Datenannahmen und Einlesen

- Eingang: Textdateien im Verzeichnis `data/text`
- Sprache wird aus dem Dateipfad mit einer Regex extrahiert (Parent-Ordner = Sprachcode)
- Einlesen mit `wholeTextFiles`: liefert Paare `(Pfad, Inhalt)` für Text und Sprachzuordnung

2.2 Preprocessing

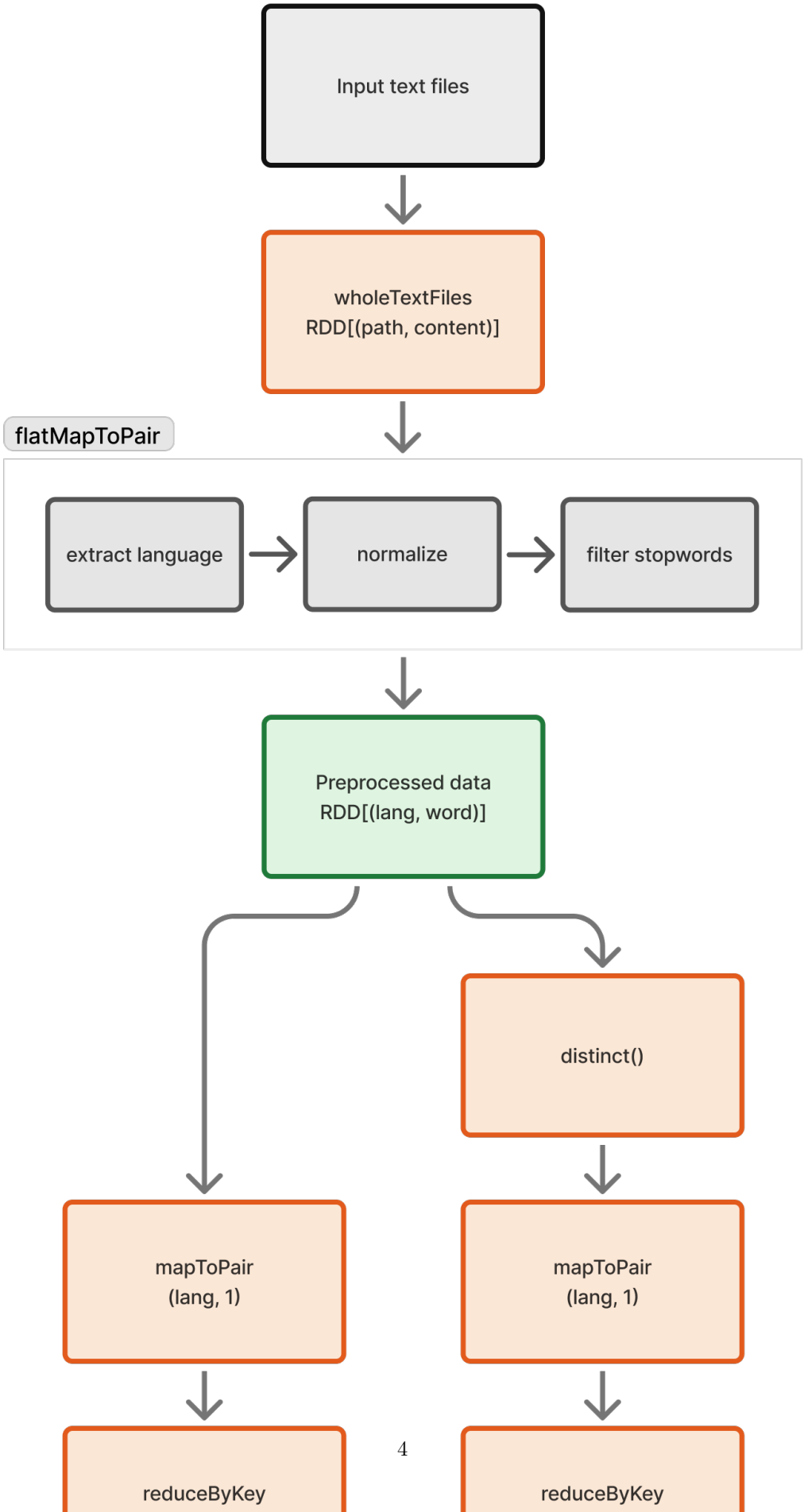
- Extraktion von Wörtern (`\p{L}+`), um auch nicht-lateinische Sprachen unterstützt werden
- Normalisierung und Kleinschreibung zur Reduktion von Duplikaten
- Stoppwort-Filter

2.3 Verteilte Berechnung in Spark

- Erzeugung eines RDD mit Paaren `(language, word)`
- **Gesamtzahl Tokens**: `mapToPair((lang,1))` + `reduceByKey(sum)`
- **Einzigartige Tokens**: `distinct()` auf `(lang,word)` + `reduceByKey`
- Join der beiden Ergebnisse pro Sprache und Berechnung von `ttr = unique/total`

2.4 Speicherung der Ergebnisse

- Ausgabe als **CSV**
- Ausgabe auf Konsole zur schnellen Überprüfung
- Zeit messungen in der Konsole


Input text files
wholeTextFiles
RDD[(path, content)]
flatMapToPair
extract language
normalize
filter stopwords
Preprocessed data
RDD[(lang, word)]
distinct()
mapToPair
(lang, 1)
mapToPair
(lang, 1)
reduceByKey
reduceByKey

3 Beschreibung des Codes

Der Code befindet sich in der Klasse `TTRJob.java`. Im Folgenden werden die wichtigsten Code-Fragmente erläutert.

3.1 Einlesen der Dateien

Die Textdateien werden mit `wholeTextFiles` eingelesen, um den Pfad für die Sprachextraktion zu erhalten:

```
JavaPairRDD<String, String> filesRDD = sc.wholeTextFiles(textDir + "/*/*.txt", 8);
```

3.2 Tokenisierung und Stoppwort-Filter

Für die Tokenisierung wird ein Unicode-Pattern verwendet (`\p{L}+`), das alle Unicode-Buchstaben erkennt. Die Stoppwörter werden per Broadcast verteilt:

```
Broadcast<Map<String, Set<String>>> broadcastStopwords =
    sc.broadcast(stopwordsMap);

Matcher matcher = WORD_PATTERN.matcher(content);
while (matcher.find()) {
    String word = normalizeWord(matcher.group());
    if (!stopwords.contains(word)) {
        pairs.add(new Tuple2<>(language, word));
    }
}
```

3.3 TTR-Berechnung

Die verteilte Berechnung erfolgt in zwei Schritten mit anschließendem Join:

```
// Total tokens per language
JavaPairRDD<String, Long> totalTokensPerLang = langWordPairs
    .mapToPair(t -> new Tuple2<>(t._1(), 1L))
    .reduceByKey(Long::sum);

// Unique tokens per language
JavaPairRDD<String, Long> uniqueTokensPerLang = langWordPairs
    .distinct()
    .mapToPair(t -> new Tuple2<>(t._1(), 1L))
    .reduceByKey(Long::sum);

// Join and calculate TTR
JavaPairRDD<String, Tuple2<Long, Long>> joinedCounts =
    totalTokensPerLang.join(uniqueTokensPerLang);
double ttr = (double) unique / total;
```

4 Ergebnisse

Die folgende Tabelle zeigt die Ergebnisse der TTR-Berechnung für alle acht Sprachen im Datensatz (134 MB, 363 Dateien):

Sprache	Total Tokens	Unique Tokens	TTR
dutch	60.739	12.222	0,2012
english	1.984.198	69.223	0,0349
french	2.305.570	78.002	0,0338
german	2.174.079	157.107	0,0723
italian	1.454.801	98.773	0,0679
russian	201.710	59.296	0,2940
spanish	622.341	66.591	0,1070
ukrainian	473.116	94.795	0,2004

Tabelle 1: TTR-Ergebnisse pro Sprache

Russisch zeigt die höchste sprachliche Vielfalt (TTR = 0,29). Französisch und Englisch haben niedrigere TTR-Werte (ca. 0,03).

5 Tests von Leistung und Laufzeiten

5.1 Messaufbau

- **Hardware**: MacBook mit Apple Silicon, 18GB RAM, Spark Local Mode
- **Datensatz**: Original (134 MB, 363 Dateien), vergrößert (672 MB, 1815 Dateien) und huge (2,6 GB, 6045 Dateien)
- **Baseline**: Serielle Implementierung
- **Parallelisierung**: Tests mit 1, 2, 4 und 8 Kernen

5.2 Ergebnisse und Analyse

Die Performance-Tests wurden auf drei Datensätzen durchgeführt, um das Verhalten bei unterschiedlichen Datenmengen zu untersuchen:

1. **Small** (134 MB)
2. **Medium** (672 MB)
3. **Huge** (2,6 GB)

5.2.1 Laufzeit-Vergleich: Serial vs. Spark

Die folgende Grafik zeigt den Vergleich zwischen der seriellen Java-Implementierung und Spark (mit verschiedenen Kern-Anzahlen).

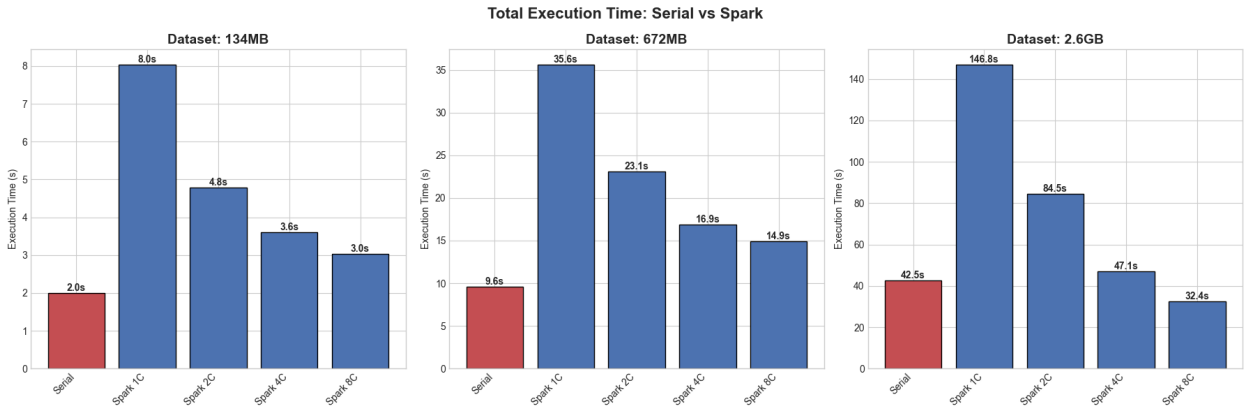


Abbildung 2: Laufzeitvergleich über verschiedene Datensatzgrößen

Ausführung	Small (134 MB)	Medium (672 MB)	Huge (2,6 GB)
Serial (Java)	**1,99 s**	**9,25 s**	42,47 s
Spark (1 Core)	8,31 s	34,40 s	146,85 s
Spark (8 Cores)	3,12 s	14,58 s	**32,36 s**

Tabelle 2: Kern-Ergebnisse der Benchmarks

Beobachtung:

- **Small/Medium Data**: Hier dominiert die **Serial**-Implementierung. Der Overhead von Spark ist im Verhältnis zur Rechenzeit zu hoch.
- **Big Data (2,6 GB)**: Hier zeigt sich der **Wendepunkt**. Spark (8 Cores) überholt die serielle Lösung (32s vs 42s). Die Parallelisierung zahlt sich nun aus, da die reine Rechenlast den Verwaltungs-Overhead übersteigt.

5.2.2 Skalierbarkeit (Speedup)

Die Speedup-Analyse zeigt, wie effizient Spark zusätzliche Kerne nutzt (Referenz: Spark 1 Core).

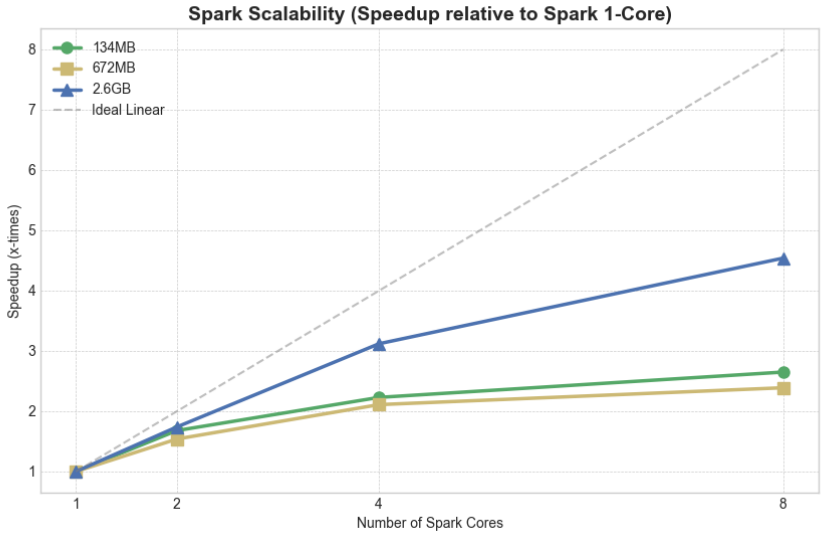


Abbildung 3: Skalierung (Speedup) relativ zu Spark 1-Core

Fazit: Spark zeigt auf dem größtem Datensatz einen Speedup von ca. **4,5x** auf 8 Kernen. Dies bestätigt, dass Spark für größere Datenmengen skaliert, während die serielle Lösung linear an die Hardware-Grenzen gebunden ist.

6 Fazit

- Die Type-Token-Ratio konnte erfolgreich für acht Sprachen mit Apache Spark berechnet werden.
- **Höchste sprachliche Vielfalt**: Russisch (TTR = 0,29).
- **Niedrigste Vielfalt**: Französisch und Englisch (TTR = 0,03).
- Die Parallelisierung zeigt mit 8 Kernen einen Speedup von **4,5x** gegenüber sequentieller Verarbeitung auf dem 2,6 GB Datensatz.
- Der Broadcast der Stoppwörter und das Caching der RDDs optimieren die Performance.
- Der `distinct()`-Operator ist die rechenintensivste Operation (Shuffle).