WHZ Westsächsische
Hochschule Zwickau
Hochschule für Mobilität

Wissenschaftliche Arbeit

Stream Gatherers mit Java-24

Sanzhar Berdikozhoev

Ziele der Arbeit

- Das Java-Programm zur Erzeugung von n-Gramme und für die Kookkurenz-Analyse zu entwickeln.
- Die zwei Ansätze, nämlich mit und ohne Stream Gatherers, zur Lösung der Aufgabe anzuwenden und sie zu vergleichen.
- Die Ergebnisse darzustellen.

Inhalt

Stream-API: Definition

Stream-API ist eine funktionale Schnittstelle für Datensammlungsverarbeitung, die erst in Java-8 eingeführt wurde. [1]

Stream-API:

- verwendet Lambda-Ausdrücke
- führt das Stream-Objekt ein

Stream-API: Architektur

Stream-API besteht aus zwei Arten von Operationen:

1. Zwischenoperationen - umwandeln die Elemente eines Streams
2. Endoperationen - transformieren ein Stream in Datensammlung (List, Array)

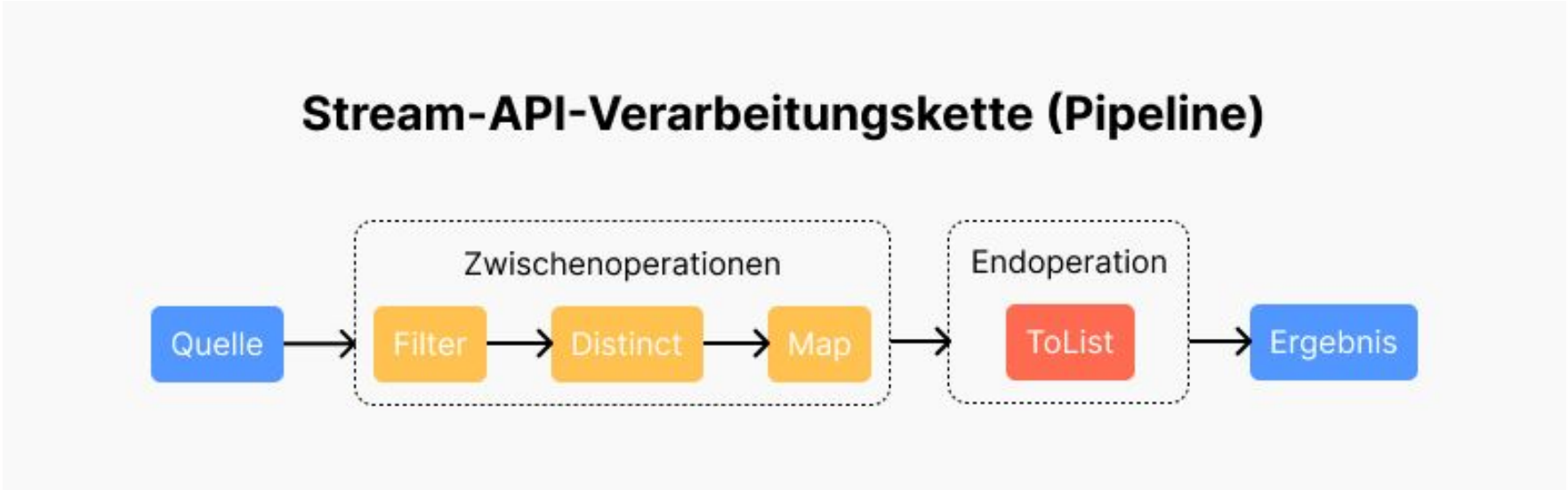


Abbildung 1. Stream-API Pipeline-Modell [2]

(selbst erstellt)

Stream-API: Beispiel

```
var result = Stream.of(5, 2, 8, 3, 1, 4)
    .filter(n -> n % 2 == 0) // gerade Zahlen herausfiltern
    .map(n -> n * n)         // quadrieren
    .sorted()                // sortieren
    .toList();               // Stream in eine Liste
                                umwandeln

    // result => [4, 6, 64]
```

Codebeispiel 1. Anwendung von Stream-API [2] (selbst geschrieben)

Stream-API: Einschränkungen

- fest definierte Operationen (Map, Filtern, Sort)
- ungeeignet für komplexe Aufgaben
- keine Möglichkeit eigene Zwischenoperationen zu erstellen

(Notiz: für benutzerdefinierte Endoperationen die Collector-Schnittstelle schon in Java-8 existiert)

Stream-API: distinctByLength

```
var result = Stream.of("foo", "bar", "baz", "quux")
        .distinctBy(String::length)
        .toList();                          // hypothetisch

var result = Stream.of("foo", "bar", "baz", "quux")
        .map(DistinctByLength::new)
        .distinct()
        .map(DistinctByLength::str)
        .toList();

       // result ==> [foo, quux]
```

Codebeispiel 2. distinctByLength-Operation [1]

Stream-API: distinctByLength

```
record DistinctByLength(String str) {
        @Override
        public boolean equals(Object obj) {
            return obj instanceof DistinctByLength(String other)
                    && str.length() == other.length();
        }

        @Override
        public int hashCode() {
            return str == null ? 0 : Integer.hashCode(str.length());
        }
}
```

Codebeispiel 3. distinctByLength-Hilfsobjekt [1]

Stream Gatherers: Definition

Stream Gatherers ist eine neue Schnittstelle zur Implementierung benutzerdefinierte Zwischenoperationen, die in Java-24 eingeführt wurde. [1]

Stream Gatherers führt ein:

- die gather-Methode für Integration von Gatherer in Stream-Kette
- das Gatherer-Interface

Stream Gatherers: Struktur

- Initializer: erzeugt das Ausgangszustandsobjekt.
- Integrator (obligatorisch): verarbeitet jedes neue Element und aktualisiert den Zustand.
- Combiner: definiert Logik für parallele Verarbeitung.
- Finisher: letzte Aktion nach dem Ende des Upstreams.

Stream Gatherers: Beisp.


```java
public static Gatherer<String, Set<Integer>, String> distinctByLength()
{
    return Gatherer.of(
            HashSet::new, // Initializer: Zustand
            (state, element, downstream) -> {
                int length = element.length();
                if (state.add(length)) {
                    return downstream.push(element);
                }
                return true;
            }, // Integrator: Verarbeitungslogik
            Gatherer.defaultCombiner(), // Combiner: omitted
            Gatherer.defaultFinisher()  // Finisher: omitted
    );
}
```

Codebeispiel 4. Gatherer für distinctByLength-Operation [2]

(selbst geschrieben)

Stream Gatherers: Beisp.


```java
// Stream-API
var result = Stream.of("foo", "bar", "baz", "quux")
        .map(DistinctByLength::new)
        .distinct()
        .map(DistinctByLength::str)
        .toList();

        // result ==> [foo, quux]

// Stream Gatherers
var result = Stream.of("foo", "bar", "baz", "quux")
        .gather(distinctByLength())
        .toList();

        // result ==> [foo, quux]
```

Codebeispiel 4. distinctByLength-Operation ohne und mit Gatherers [2]

(selbst geschrieben)

n-Gramme

n-Gramme sind die geordnete n-Tupel von Elementen. Im Kontext von NLP (natürliche Sprachverarbeitung) sind sie meistens die Wörter. Die Zahl n definiert die Länge der Reihenfolge und kann variieren. [3]

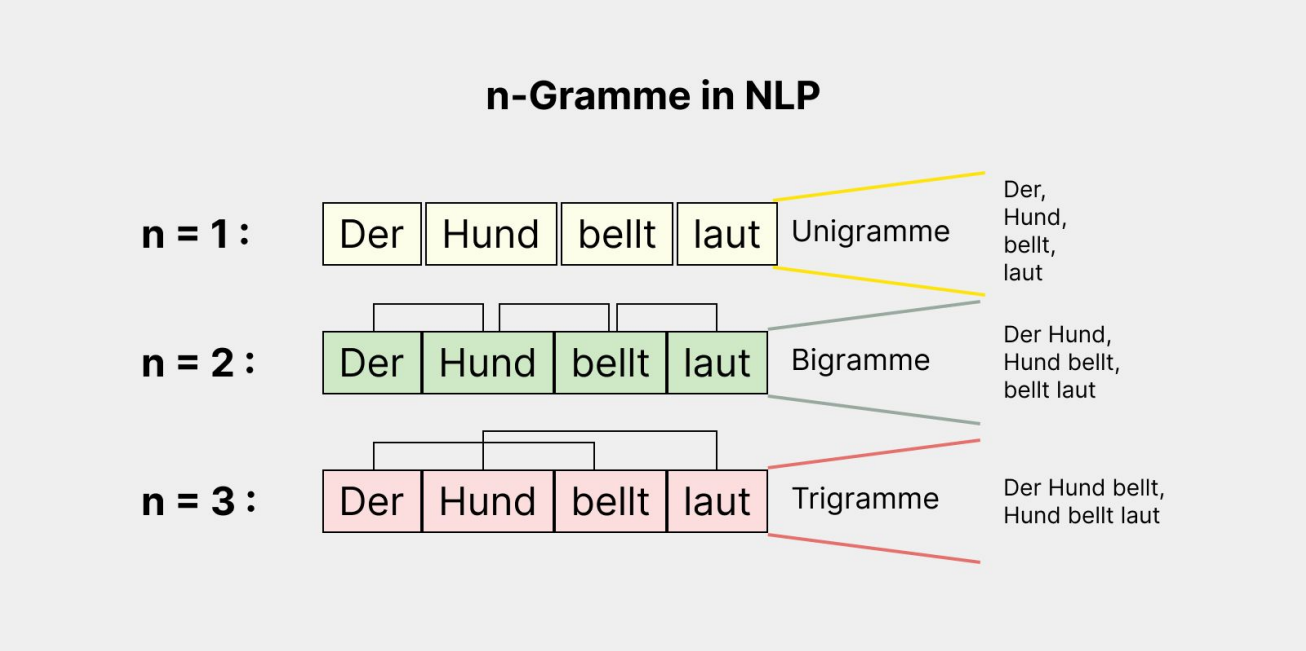


Abbildung 2. n-Gramme in natürlicher Sprachverarbeitung [2]

(selbst erstellt)

n-Gramme: Lösungsstrategie

1. Startpunkt in der Datensammlung setzen.
2. n-mal darüber iterieren und die Tokens (Wörter) sammeln.
3. n-Gram aus der gesammelte Tokens zusammenstellen.
4. Startpunkt um eine Position nach vorne verschieben.
5. Schritte 1 bis 4 wiederholen, bis die Bedingung (n <= Sätzlange - n) erfüllt ist.

Kookkurrenz-Analyse

Die Kookkurrenz-Analyse ist eine grundlegende Methode der Computerlinguistik, die das gemeinsame Auftreten von Wörtern (Kookkurrenz) im Text untersucht. Sie bildet die Basis für das Verständnis lexikalischer Beziehungen. [4]

Kookkurrenz-Matrix

Fenstergröße = 2

	Der	Hund	bellt	laut
Der	0	1	1	0
Hund	1	0	1	1
bellt	1	1	0	1
laut	0	1	1	0

Abbildung 3. Kookkurrenz-Matrix [2]

(selbst erstellt)

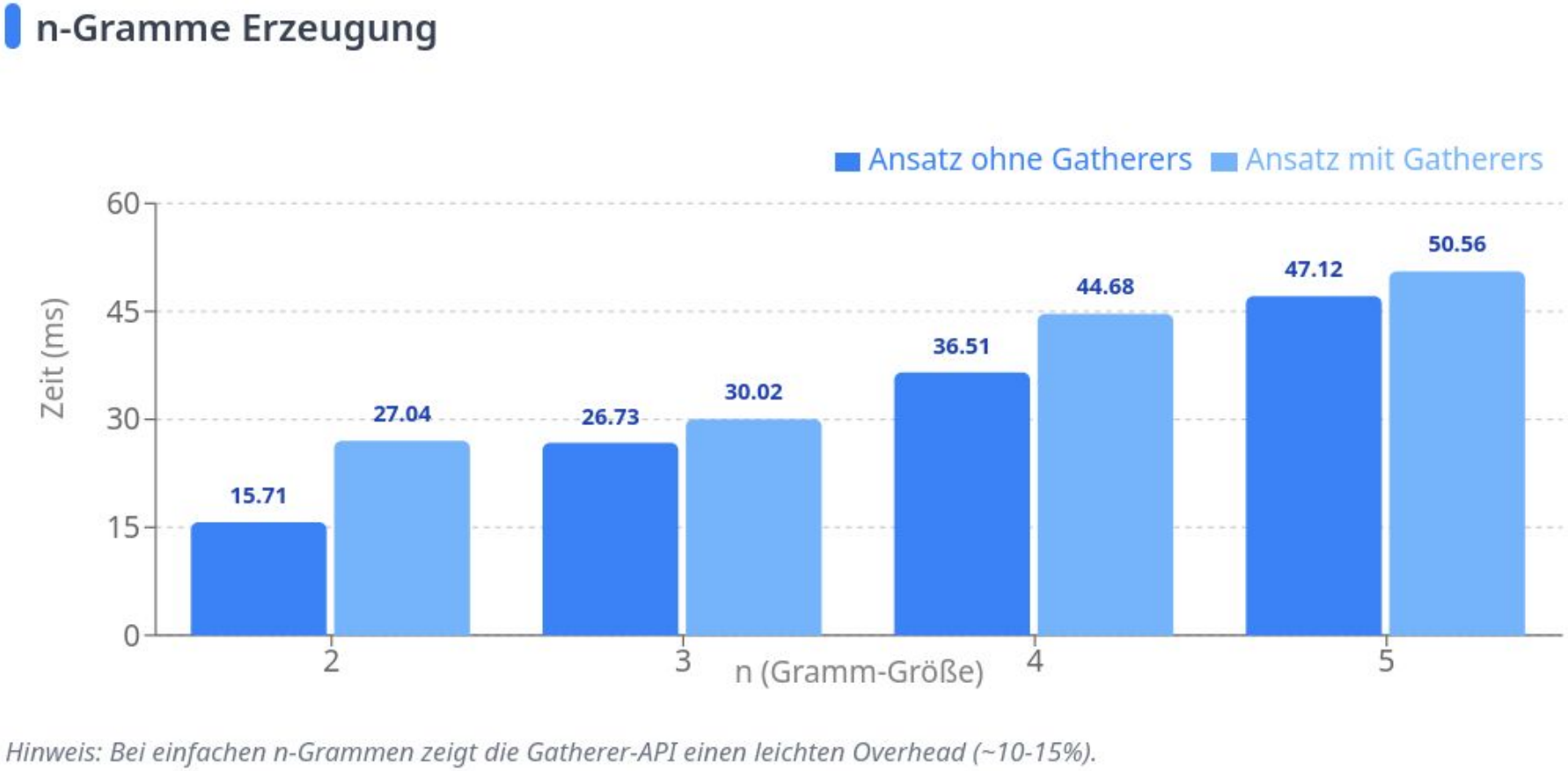
Kookkurrenz-Analyse: Strategie

1. Startpunkt (Zentrumswort) in der Datensammlung setzen.
2. w Positionen links und rechts vom Zentrumswort befinden.
3. Ein Paar (Zentrumswort, Nachbarwort) für jedes Wort
4. Frequenz von Wörtern aktualisieren.
   - (dafür wird eine verschachtelte Mappe verwendet Map<Zentrumswort, Map<Nachbarwort, Frequenz>>)
5. Startpunkt um eine Position nach vorne verschieben.
6. Schritte 1 bis 4 wiederholen, bis die Bedingung (n <= Sätzlange - n) erfüllt ist.

Demonstration des Programms

im Prozess!!!

Vergleich: Leistung

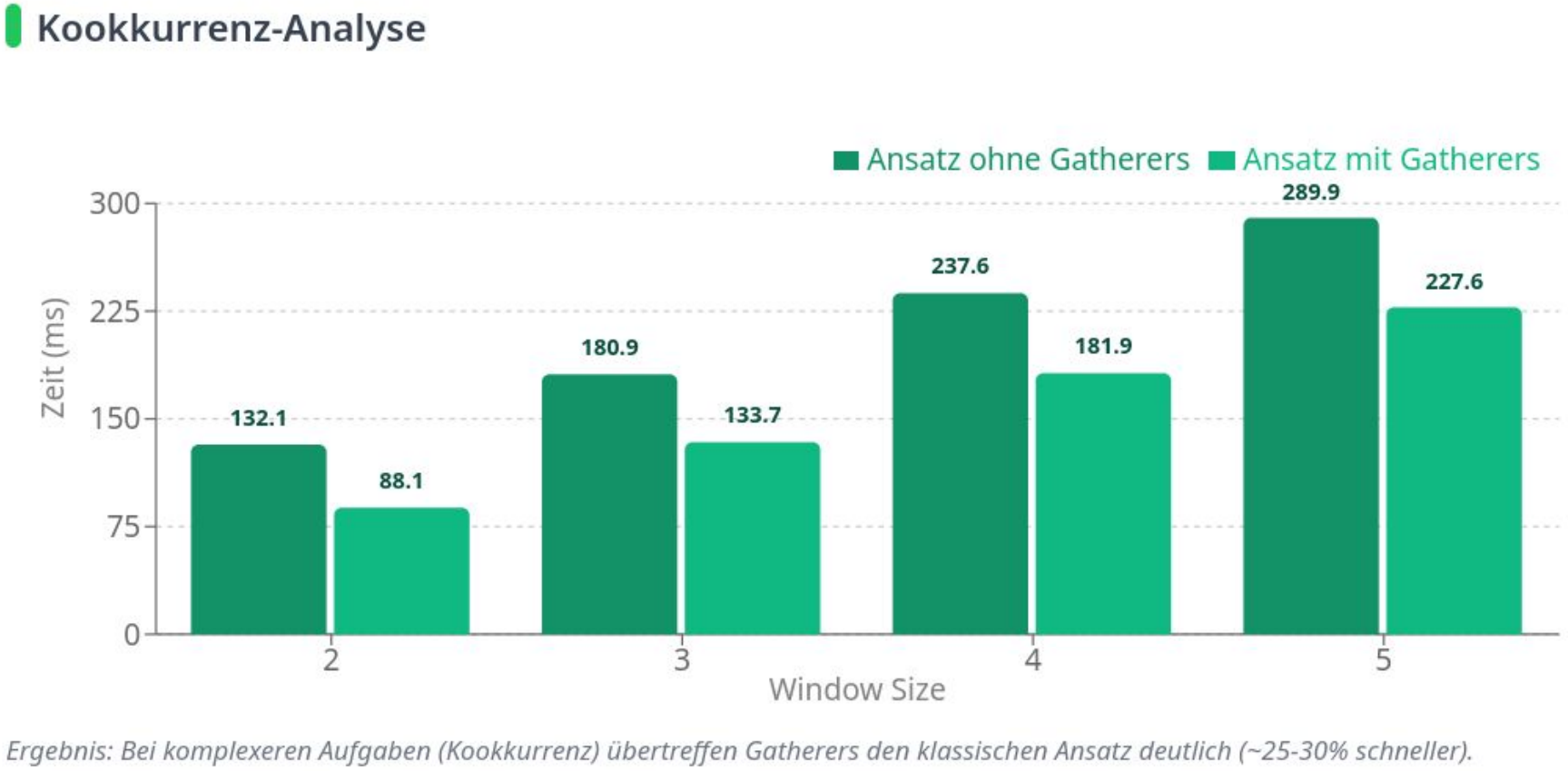


Grafik 2. Ergebnisse: n-Gramme-Erzeugung [2]

(durch Gemini generiert)

Vergleich: Leistung

WHZ Westsächsische Hochschule Zwickau
Hochschule für Mobilität



Grafik 2. Ergebnisse: Kookkurrenz-Analyse [2]

(durch Gemini generiert)

Vergleich: Entwicklungskomplexität

Stream-API (keine Gatherers)

- leichter für die einfache Aufgaben zu verwenden.
- nicht geeignet für die komplexe Aufgaben.
- funktioniert seit Java-8 und ist ziemlich beliebt.

Stream Gatherers

- erhöhen die anfängliche Implementierungshärte
- verbessern die Wartbarkeit und die Lesbarkeit
- funktioniert nur in Java-24 und höher.

Zusammenfassung

- Stream Gatherers erweitern die Stream-API.
- Stream Gatherers ermöglichen die komplexe Logik in einer separaten Klasse zu enkapsulieren.
- Stream Gatherers eignen sich gut für die komplexe Aufgaben (wie z.B. Kookkurrenz-Analyse).

Quellenverzeichnis

[1] Stream Gatherers JEP - https://openjdk.org/jeps/485

[2] Ergebnisse wissenschaftlicher Arbeiten (einschließlich Screenshots, JSON-Dateien und WA selbst) - https://github.com/SanzharBerdikozhoev/wa-stream-gatherers

[3] Daniel Jurafsky & James H. Martin (2026): n-gram Language Models - https://web.stanford.edu/~jurafsky/slp3/3.pdf

[4] Stefan Evert (Osnabrück, 2007): Corpora and collocations - https://lexically.net/downloads/corpus_linguistics/Evert2008.pdf

[5] Corpus of German Language Fiction - https://figshare.com/articles/dataset/Corpus_of_German-Language_Fiction_txt_/4524680?file=7320866

WHZ Westsächsische
Hochschule Zwickau
Hochschule für Mobilität

https://github.com/SanzharBerdikozhoev/wa-stream-gatherers

Wissenschaftliche Arbeit

Stream Gatherers mit Java-24

Sanzhar Berdikozhoev