Westsächsische Hochschule Zwickau

Studiengang: Informatik

Wintersemester 2025/2026

Wissenschaftliche Arbeit

Stream Gatherers mit Java-24

Sanzhar Berdikozhoev

Matrikelnummer: 61270

betreut von

Prof. Dr. Laue

Abgabetermin: 15.12.2025

Abkürzungsverzeichnis

API Application Programming Interface

IDE Integrated Development Environment

JDK Java Development Toolkit

JEP JDK Enhancement Proposal

JIT Just-In-Time (Compiler)

JSON Javascript Object Notation

JVM Java Virtual Machine

NLP Natural Language Processing

Inhaltsverzeichnis

Kapitel 1 Einleitung

Java ist eine der weltweit wichtigsten Programmiersprachen. Sie steht seit vielen Jahren ganz oben in den Rankings für beliebte Sprachen (wie dem TIOBE-Index[1]) und ist der Industriestandard für große Firmen und komplexe Software. Viele Programme für Banken, große Webseiten und Geschäftsanwendungen werden mit Java erstellt, weil die Sprache sehr stabil und zuverlässig ist.

Java entwickelt sich laufend weiter und wurde in der Version Java-24 um eine wichtige Funktion ergänzt: die Stream Gatherers. Diese neue Technologie ist eine Erweiterung der zentralen Java Stream-API und bietet mehr Möglichkeiten bei der Datenverarbeitung.

Ziel dieser Arbeit ist die Untersuchung und praktische Anwendung von Stream Gatherers. Dazu muss ein Java-Programm zum Erzeugen und Zählen von n-Grammen sowie zur Kookkurrenz-Analyse entwickelt werden. Zusätzlich muss die Anwendbarkeit dieses Programms an einem längeren deutschen Eingabetext demonstriert werden.

[1] TIOBE-Index - https://www.tiobe.com/tiobe-index/

Kapitel 2 Stream-API und Stream Gatherers

2.1 Grundlagen der Stream-API

Die Stream-API wurde erstmals in Java-8 eingeführt und bietet einen bequemen, funktionalen Ansatz zur Datenverarbeitung. [1] Ihre Architektur basiert auf dem **Pipeline-Modell** (Verarbeitungskette), in dem die Daten durch aufeinander folgende Operationen verarbeitet werden. Nachfolgendes Beispiel demonstriert Verwendung von Stream-API.

```
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. Datenverarbeitung mit Stream-API

2.1.1 Operationtype

Die Operationen der Stream-API werden in zwei Kategorien unterteilt. Die sind Zwischen- und Endoperationen. [1]

Zwischenoperationen bilden die Verarbeitungskette, verarbeiten den eingehenden Stream und geben immer einen neuen Stream zurück. Sie sind weiter in zustandslose und zustandsbehaftete unterteilt.

- **Zustandslose** Operationen: Sie benötigen zur Durchführung ausschließlich das aktuell bearbeitete Element und speichern keine Informationen über andere Elemente in der Datenkette. Beispiele hierfür sind `map` oder `filter`.

- **Zustandsbehaftete** Operationen: Sie speichern Informationen über frühere Elemente. Die einzigen zustandsbehafteten Operationen sind `sorted` und `distinct`.

Endoperationen sind der letzte Schritt in der Verarbeitungskette. Sie starten die gesamte Verarbeitungskette und produzieren ein endgültiges Ergebnis. Nach einer Endoperation kann der Stream nicht weiterverwendet werden.

```
var result = Stream.of(1, 19, 63, 27, 27, 3, 1, 89, 89, -10)
       .distinct()       // Zwischenoperation
       .sorted()         // Zwischenoperation
       .skip(1)          // Zwischenoperation
       .limit(2)         // Zwischenoperation
       .toList();        // Endoperation

// result ==> [1, 3]
```

Codebeispiel 2. Zwischen- und Endoperationen

2.2 Architektonische Grenzen der Stream-API

Die Stream-API bringt zwar eine große Auswahl an Operationen mit, wie Filtern, Sortieren oder Reduzieren. Jedoch ist diese Liste **fest** und **nicht erweiterbar**. Diese Operationen sind für die Lösung vieler Probleme ausreichend, aber für einige komplexe Aufgaben reichen sie möglicherweise nicht aus oder ihre Anwendung ist völlig unpraktisch.[1]

2.2.1 Die architektonische Starrheit am Beispiel von Distinct-Operationen

Beispielsweise gibt es eine Aufgabe, die darin besteht, in einem Stream von Zeichenfolgen nur eindeutige Elemente zuzulassen. Das Kriterium für die Eindeutigkeit ist hierbei jedoch nicht der Inhalt, sondern die Länge der

Zeichenfolge. Ziel ist es, pro auftretender Zeichenlänge höchstens ein Element im resultierenden Stream zu behalten. [1]

Für diese Aufgabe eine `distinctBy`-Operation geeignet. Aber sie ist in der Standardbibliothek Java Stream-API nicht vorhanden.

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

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

Codebeispiel 3. Hypothetische Verwendung von `distinctBy` [1]

Die ähnlichste Operation `distinct` wählt jedoch die Elemente nach ihrem **Inhalt** aus. Zur Lösung kann eine neue Klasse eingeführt werden, die Zeichenfolgen nach ihrer **Länge** vergleicht. Und dann ist jede Zeichenfolge im Stream in diese Klasse umgewandelt. Diese Ausdrucksweise der Aufgabe ist jedoch nicht intuitiv und führt zu schwer wartbaren Code. [1]

```
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());
    }

}

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

```
                    .toList();

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

Codebeispiel 4. Realistische Lösung der Aufgabe mittels externen Klasse [1]

Die Lösung für das beschriebene Problem liegt in der Schaffung der Möglichkeit, eigene Stream-Operationen zu definieren. Zur Implementierung benutzerdefinierter Endoperationen wurde bereits die Schnittstelle `Collector` in Java-8 eingeführt. Nachfolgendes Beispiel demonstriert Anwendung dieser Schnittstelle.

```
var stringJoinCollector = Collector.of(
       () -> new StringJoiner(delimiter),
       StringJoiner::add,
       StringJoiner::merge,
       StringJoiner::toString
);

var result = Stream.of("Apfel", "Banane", "Kirsche")
.collect(stringJoinCollector(" und "));

// result ==> [Apfel und Banane und Kirsche sind Obst]
```

Codebeispiel 5. Verwendung von `Collector` Schnittstelle [1]

`Collector` ermöglichte jedoch keine Erstellung eigener Zwischenoperationen. Diese architektonische Lücke wurde erst mit der Einführung von **Stream Gatherers** in Java-24 geschlossen.

2.3 Grundlagen der Stream Gatherers

Stream Gatherers sind die wichtigste Ergänzung des Stream-API. Sie wurden erstmals in Java-22 als Vorschau-Funktion hinzugefügt und bereits in Java-24 endgültig eingeführt.

Sie dienen als flexible Schnittstelle und ermöglichen die Erstellung **benutzerdefinierter, zustandsbehafteter Zwischenoperationen**. Dies erlaubt die native Implementierung komplexer Verarbeitungsmuster direkt in die Stream-Verarbeitungskette. Um diese flexible Logik zu realisieren, bietet Java-24 die spezielle Schnittstelle `Gatherer`.

2.3.1 Möglichkeiten der Gatherers

Gatherers können Elemente auf eine Weise umwandeln, die eins-zu-eins, eins-zu-viele, viele-zu-eins oder viele-zu-viele ist. Sie können zuvor gesehene Elemente verfolgen, um die Umwandlung späterer Elemente zu beeinflussen. Sie können Kurzschlüsse herstellen, um unendliche Streams in endliche umzuwandeln. Sie können eine parallele Ausführung ermöglichen. [1]

Beispielsweise kann ein Gatherer ein Eingabeelement in ein Ausgabeelement umwandeln, bis eine bestimmte Bedingung erfüllt ist. In diesem Fall beginnt er, ein Eingabeelement in zwei Ausgabeelemente umzuwandeln. [1]

2.3.2 Architektur der Gatherer Schnittstelle

Ein `Gatherer` wird durch vier Funktionen definiert, die zusammenarbeiten, um den Zustandslebenszyklus zu steuern. [1]

1. `initializer` (Optional): Erzeugt das private Zustandsobjekt. Dieses Objekt hält den Zustand über die Verarbeitung der Stream-Elemente hinweg. Beispiel: Speichern des vorherigen Elementes, um es mit dem nächsten zu vergleichen. [1]
2. `integrator`: Verarbeitet jedes neue Element aus dem Eingabestream. Er greift auf das Zustandsobjekt zu, aktualisiert es und kann Elemente an den Ausgabestream weitergeben. Die

Funktion kann die Verarbeitung vorzeitig beenden, falls nötig (Kurzschluss). [1]

3. `combiner` (Optional): Definiert, wie Zustände von parallel verarbeiteten Streams kombiniert werden. Wenn ein Gatherer keine `combiner`-Funktion hat, verarbeitet er die innere Daten aufeinanderfolgend. [1]

4. `finisher` (Optional): Wird einmal am Ende aufgerufen, wenn keine weiteren Eingabeelemente vorhanden sind. Diese Funktion kann den finalen Zustand überprüfen und möglicherweise weitere Ausgabeelemente senden oder einen Fehler melden. [1]

Die Methode `Stream::gather` führt nacheinander die Schritte **Initializer**, dann die iterative Ausführung des **Integrators** für jedes Element und schließlich den **Finisher** aus. [1]

2.3.3 Eingebaute Gatherers

In der Klasse `java.util.stream.Gatherers` werden standardmäßig verschiedene Gatherers bereitgestellt: [1]

- `fold`: Ein zustandsbehafteter Viele-zu-Eins-Gatherer. Er erstellt inkrementell ein Aggregat und gibt dieses als einziges Ergebnis aus, wenn keine weiteren Eingabeelemente vorhanden sind. [1]
- `mapConcurrent`: Ein zustandsbehafteter Eins-zu-Eins-Gatherer. Er ruft für jedes Element eine bereitgestellte Funktion gleichzeitig auf, begrenzt durch eine angegebene Grenze. [1]
- `scan`: Ein zustandsbehafteter Eins-zu-Eins-Gatherer. Er wendet eine Funktion auf den aktuellen Zustand und das aktuelle Element an, um das nächste Element zu erzeugen. Dieses Zwischenergebnis wird direkt an den Ausgabestream weitergegeben (z. B. für die Berechnung eines laufenden Durchschnitts). [1]

- `windowFixed`: Ein zustandsbehafteter Viele-zu-Viele-Gatherer. Er gruppiert Elemente in Listen einer festgelegten Größe. Die fertigen Listen werden an den Ausgabestream gesendet, sobald sie voll sind. [1]
- `windowSliding`: Ein zustandsbehafteter Viele-zu-Viele-Gatherer. Er gruppiert Elemente in Listen einer festgelegten Größe, wobei sich das Fenster Element für Element verschiebt. Nach dem ersten Fenster wird jede Liste erstellt, indem das erste Element des Vorgängers entfernt und das nächste Element aus dem Stream hinzugefügt wird. [1]

2.3.4 Inhalt des Standard-Gatherers windowFixed

Um die praktische Funktionsweise der Gatherer-Schnittstelle zu verdeutlichen, zeigt das folgende Beispiel die interne Implementierung des Standard-Gatherers `windowFixed`, der Elemente eines Streams in feste Gruppen unterteilt.

```
record WindowFixed<TR>(int windowSize)
    implements Gatherer<TR, ArrayList<TR>, List<TR>>
{

    public WindowFixed {
        // Validate input
        if (windowSize < 1) {
            throw new IllegalArgumentException(
                "window size must be positive");
        }
    }

    @Override
    public Supplier<ArrayList<TR>> initializer() {
        // Create an ArrayList to hold the current open window
        return () -> new ArrayList<>(windowSize);
    }

    @Override
```

```
    public Integrator<ArrayList<TR>, TR, List<TR>> integrator() {
        // The integrator is invoked for each element consumed
        return Gatherer.Integrator.ofGreedy(
        (window, element, downstream) -> {

            // Add the element to the current open window
            window.add(element);

            // Until we reach our desired window size,
            // return true to signal that more elements are desired
            if (window.size() < windowSize)
                return true;

            // When the window is full, close it by creating a copy
            var result = new ArrayList<TR>(window);

            // Clear the window so the next can be started
            window.clear();

            // Send the closed window downstream
            return downstream.push(result);

        });
    }
    // The combiner is omitted since this operation is intrinsically
       sequential, and thus cannot be parallelized

    @Override
    public BiConsumer<ArrayList<TR>, Downstream<? super List<TR>>>
        finisher() {
        // The finisher runs when there are no more elements to pass
        // from the upstream
        return (window, downstream) -> {
            // If the downstream still accepts more elements and the
            // current

            // open window is non-empty, then send a copy of it
            // downstream
            if(!downstream.isRejecting() && !window.isEmpty()) {
                downstream.push(new ArrayList<TR>(window));
                window.clear();
            }
        };
    }
}
```

```
// Example of usage:
var result = Stream.of(1,2,3,4,5,6,7,8,9)
.gather(new WindowFixed(3))
.toList()
// result ==> [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Codebeispiel 5. Realisation von Standard-Gatherer `windowFixed` [1]

Kapitel 3 n-Gramme

3.1 Grundlagen der n-Gramme

n-Gramme sind die geordnete n-Tupel von n Elementen. Im Kontext von natürlicher Sprachverarbeitung (NLP) sind n-Gramme in der Regel Wörter und die Zahl n definiert die Länge der Reihenfolge und kann variieren. [2]

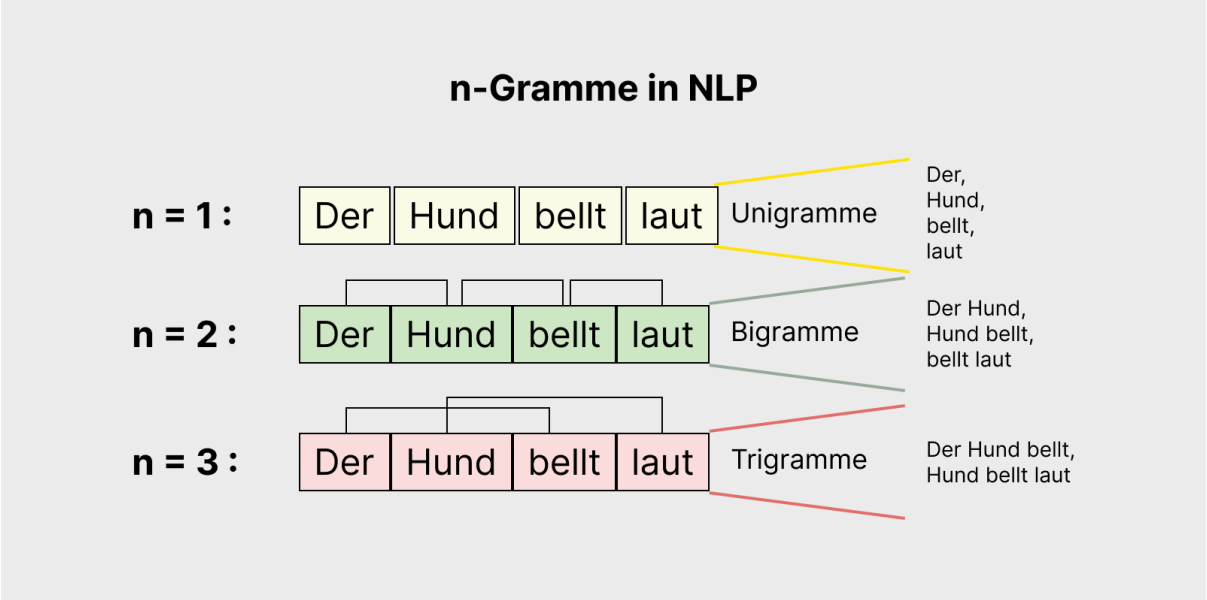


Abbildung 1. n-Gramme in natürlicher Sprachverarbeitung

3.1.1 Typen von n-Grammen

Je nach der Anzahl der Elemente (n) werden n-Gramme wie folgt unterschieden:

- Unigramme (1-Gramme): Hierbei handelt es sich um einzelne, isolierte Wörter. Unigramme sind die Basis für einfache Häufigkeitsanalysen. [2]

- Bigramme (2-Gramme): Bestehen aus Paaren von direkt aufeinanderfolgenden Wörtern. Sie erfassen einfache Wortassoziationen ("rotes Auto"). [2]

- Trigramme (3-Gramme): Sind Abfolgen von drei Wörtern. Sie bieten einen tieferen Kontext und mehr Informationen über die syntaktischen Beziehungen ("der rote Teppich"). [2]

- n-Gramme höherer Ordnung: Folgen mit $n > 3$. Mit steigender n steigt die Detailgenauigkeit des Kontextes. Das liegt daran, dass mehr Wörter in die Betrachtung einbezogen werden. Jedoch nehmen gleichzeitig die Komplexität und der benötigte Rechenaufwand stark zu. Dies ist darauf zurückzuführen, dass die Anzahl der möglichen Kombinationen exponentiell wächst. [2]

n-Gramme sind für NLP-Aufgaben unverzichtbar. Sie helfen dabei zu verstehen, wie Wörter zusammenhängen und wie die Wortreihenfolge die Bedeutung beeinflusst. [2]

3.2 Anwendung in der Textanalyse

n-Gramme spielen eine zentrale Rolle bei der Gewinnung eines tieferen Verständnisses von sprachlichen Mustern, Kontext und Bedeutung. Ihre Fähigkeit, Wortfolgen zu erfassen, macht sie zu einem vielseitigen Werkzeug in verschiedenen NLP-Anwendungen: [2]

- Stimmungsanalyse (Sentiment-Analyse): Durch die Analyse von n-Grammen können Modelle gefühlstragende Phrasen identifizieren ("nicht gut", "sehr zufrieden") und so den emotionalen Ton eines Textes bestimmen. [2]

- Texterstellung und Autovervollständigung: n-Gramme nutzen Wahrscheinlichkeiten von Wortfolgen, um kohärente (logisch zusammenhängende und gut verständliche) und kontextuell relevante

Texte zu generieren (z.B. in Chatbots oder Auto-Suggest-Funktionen). [2]

- Text-Klassifizierung: Sie dienen als leistungsstarke Merkmale zur Darstellung von Dokumenten. Durch die Zählung von n-Grammen können Klassifikatoren Dokumente präziser in Kategorien einteilen. [2]

- Extraktion von Phrasen und Kollokationen: n-Gramme sind hervorragend geeignet, um häufig auftretende Wortkombinationen (Kollokationen) zu entdecken. Solche Wortkombinationen sind für die Extraktion von Schlüsselwörtern oder die Zusammenfassung von Inhalten relevant. [2]

- Informationsbeschaffung: Suchmaschinen verwenden n-Gramme, um die Relevanz von Suchergebnissen zu verbessern, indem sie die Anfrage des Benutzers präziser mit Dokumenten abgleichen. [2]

Kapitel 4 Kookkurrenz-Analyse

4.1 Grundlagen der Kookkurrenz

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. [3]

Die grundlegende Idee basiert auf der Beobachtung der statistischen Abweichung von der reinen Zufallswahrscheinlichkeit: Wenn zwei Wörter (X und Z) viel häufiger zusammen auftreten, als es aufgrund ihrer individuellen Häufigkeit erwartet werden müsste, dann existiert eine besonders enge sprachliche Verbindung zwischen ihnen. [3]

4.1.1 Das Kontextfenster

Das Kontextfenster ist der wichtigste Parameter in der Kookkurrenz-Analyse. Das Fenster legt den maximal zulässigen Abstand zwischen zwei Wörtern fest, damit ihr gemeinsames Auftreten noch als Kookkurrenz gezählt wird.

Beispiel: Bei einer Fenstergröße von $k = 2$ werden alle Wörter gezählt, die maximal zwei Positionen links oder zwei Positionen rechts von einem Zentrumswort liegen.

4.1.2 Die Kookkurrenz-Matrix

Die Kookkurrenz-Matrix ist eine der Formen zur Darstellung der Kookkurrenzen. Nachfolgend ist ein Anwendungsbeispiel für den einfachen Satz *“Der Hund bellt laut”*.

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 2. Kookkurrenz-Matrix

4.2 Anwendung der Kookkurrenz-Analyse

Im Rahmen lexikalischer Analysen wird die Kookkurrenz-Analyse für folgende Zwecke benötigt:

- Identifikation von Kollokationen und festen Wendungen [3]: Dies dient zur Erkennung nicht-zufälliger Wortverbindungen, bei denen Wörter signifikant häufiger gemeinsam auftreten, als es der Zufall erwarten lässt. (z. B. "eine Entscheidung treffen").
- Bestimmung der relevanten Lesarten des Lexems [3]: Die Wörter, die mit einem bestimmten Wort kookkurrieren, helfen dabei, seine genaue Bedeutung (Lesart) in diesem Kontext zu bestimmen (z. B. die Lesart der *Bank* in Kombination mit *Geld* und mit *sitzen*).
- Bestimmung von semantischen Relationen [3]:
    - Kohyponymie: Wörter, die sich denselben Oberbegriff teilen. (z. B. *Hund* und *Katze*).

    - Hyperonymie/Hyponymie: Über- und Unterbegriffe (z. B. *Tier* und *Hund*).
    - Synonymie/Antonymie: Ähnliche oder gegenteilige Wörter.
    - Meronymie: Teil-Ganzes-Beziehungen (z. B. *Rad* und *Auto*).
- Bestimmung der Valenz [3]: Die Kookkurrenz hilft zu bestimmen, welche syntaktischen Ergänzungen (z. B. Präpositionen, Objekte) ein Verb typischerweise erfordert (z. B. *bestehen* aus).

Kapitel 5 Implementierung

Dieses Kapitel widmet sich der praktischen Anwendung und Demonstration der theoretischen Konzepte.

Die zentralen Aufgaben sind:

1. Die effiziente Erzeugung und Zählung von n-Grammen.
2. Die Durchführung der Kookkurrenz-Analyse.
3. Die anschließende Extraktion der **k** häufigsten Kookkurrenzen, um die relevantesten Wortbeziehungen zu identifizieren.

Die Implementierung dieser Aufgaben erfolgt unter Verwendung zweier unterschiedlicher Ansätze des Java Stream-API: der klassischen Stream-API (Java-8) und der neuen Gatherers-Technologie (Java-24).

Im ersten Schritt wird der Korpus in **Tokens** unterteilt – also in die kleinsten Analyseeinheiten wie Wörter, Zahlen oder Satzzeichen. Dabei nehmen alle Funktionen außer den Hilfsfunktionen als Eingabe eine Liste von Tokens und den entsprechenden Parameter (n für n-Gramme oder die Fenstergröße für die Kookkurrenz-Analyse) entgegen.

Zusätzlich wurde die Bibliothek *GSON*[2] (Version 2.11.0) angewendet, um die Ergebnisse des Programms als JSON-Dateien zu speichern.

5.1 Implementierung n-Gramm-Erzeugung

Die unten beschriebenen Methoden speichern n-Gramme als Liste von Zeichenfolgen (`List<String>`).

Beispiel für Bigramme:

```
[„Der Hund“, „Hund bellt“, „bellt laut“]
```

[2] GSON-Bibliothek - https://github.com/google/gson

5.1.1 Stream-API Methode

```
public static List<String> produceNGramsWithoutGatherer(int n,
List<String> tokens) {
    // Randfallprüfung: Ungültige n-Größe.
    if (n < 1 || n > tokens.size()) {
        return List.of();
    }

    // Sonderfall für Unigramme (n=1).
    if (n == 1) {
        return tokens;
    }

    // IntStream: Erzeugt einen Stream über alle möglichen Startindizes.
    // mapToObj: Holt die Subliste [i, i+n] und verbindet sie zum
    // n-Gramm-String.
    return IntStream.range(0, tokens.size() - n + 1)
            .mapToObj(i -> String.join(" ", tokens.subList(i, i + n)))
            .toList();
}
```

Codebeispiel 7. n-Gramm-Erzeugung ohne Stream Gatherers

Die Funktion `produceNGramsWithoutGatherer` bildet die n-Gramm-Erzeugung unter Verwendung der vorhandenen Stream-API ab. Da die Stream-API selbst keine eigene Funktion für das Verschieben von Fenstern bietet, wird eine Index-basierte Umgehungslösung verwendet.

Dabei wird ein Stream über die möglichen Anfangindizes des n-Gramms erzeugt `(IntStream.range)`. Für jeden Index `(i)` wird dann die Methode `tokens.subList(i, i + n)` aufgerufen.

5.1.2 Gatherers Methode

```
public static List<String> produceNGramsWithGatherer(
   int n, List<String> tokens) {

   // Randfallprüfung: Ungültige n-Größe.
   if (n < 1 || n > tokens.size()) {
       return List.of();
   }

   // Sonderfall für Unigramme (n=1).
   if (n == 1) {
       return tokens;
   }

   // map: Verbindet die Liste von Wörtern in jedem Fenster zu einem
      String.
   return tokens.stream()
           .gather(Gatherers.windowSliding(n))
           .map(w -> String.join(" ", w))
           .toList();
}
```

Codebeispiel 8. n-Gramm-Erzeugung mit Stream Gatherers

Die Funktion `produceNGramsWithGatherer` demonstriert die elegante und deklarative Lösung. Sie nutzt den in Java-24 eingeführten Standard-Gatherer `Gatherers.windowSliding(n)`. Die gesamte Logik der Fensterbildung wird vom `Gatherer` gekapselt.

Der Stream wird mit der Zwischenoperation `gather(Gatherers.windowSliding(n))` erweitert. Der Gatherer sammelt die Elemente und emittiert eine Liste von Zeichenketten für jedes verschobene Fenster.

Anschließend wird lediglich eine `map`-Operation ausgeführt, um die erzeugte Liste von Wörtern in eine einzelne Zeichenkette über `String.join` Operation umzuwandeln.

5.2 Kookkurrenz-Analyse

Das Ergebnis von dieser Implementation wird in einer verschachtelten Datenstruktur, der `Map<String, Map<String, Long>>`, akkumuliert werden, um die Kookkurrenz-Frequenzen effizient zu speichern:

- Die **äußere Map** verwendet jedes distinkte Wort des Korpus als Schlüssel (`String`).
- Die **innere Map** speichert als Schlüssel die Wörter, die im definierten Kontextfenster des zentralen Wortes vorkommen, und als Wert (`Long`) die Häufigkeit, mit der diese Kookkurrenz aufgetreten ist.

5.2.1 Stream-API-Methode

```
public static Map<String, Map<String, Long>>
performCooccurrenceAnalysisWithoutGatherer(
       int window, List<String> tokens) {

   // Randfallprüfung: Ungültige window-Größe.
   if (window < 1 || window > tokens.size()) {
       return Map.of();
   }

   // boxed: Konvertiert primitive IntStream zu Stream<Integer>
   return IntStream.range(0, tokens.size())
           .boxed()
           .collect(toMap(
                   // Schlüssel Mapper: Der Index i liefert
                      das Zentrumswort.
                   i -> tokens.get(i),
                   // Werte Mapper: Erstellt die Kookkurrenz-Map für das
                      aktuelle Zentrumswort
                   i -> {
                       Map<String, Long> map = new HashMap<>();
                       for (int j = i - window; j <= i + window; j++) {
                           // Prüfung der Array-Grenzen und Ausschließen
                              des Zentrumswortes (j == i)
```

```
                        if (j < 0 || j >= tokens.size() || j == i) {
                            continue;
                        }
                        // Zählt die Häufigkeit der Kookkurrenz
                        map.merge(tokens.get(j), 1L, Long::sum);
                    }
                    return map;
                },
                // Merge Function: Kombiniert Werte-Maps für
                   identische Schlüsseln (Zentrumswörter)
                (map1, map2) -> {
                    map2.forEach((k, v) -> map1.merge(k, v,
                        Long::sum));

                    return map1;
                }
            ));
}
```

Codebeispiel 9. Kookkurrenz-Analyse mit Stream-API

Die Funktion `performCooccurrenceAnalysisWithoutGatherer` demonstriert den Versuch, die Kookkurrenz-Analyse mit der Standard Stream-API zu implementieren. Da die Stream-API keine eingebaute Operation für ein gleitendes, zentriertes Fenster bietet, muss der Entwickler auf den Index-Ansatz zurückgreifen:

1. Index-Stream: Es wird ein Stream über alle Indizes der Token-Mengen (`IntStream.range(0, tokens.size())`) erzeugt, um den Zugriff auf benachbarte Elemente zu ermöglichen.
2. Die Endoperation `collect(Collectors.toMap(...))` wird verwendet, um die verschachtelte Map zu akkumulieren.

5.2.3 Gatherers-Methode

```
public class CooccurrenceAnalysisGatherer implements
        Gatherer<String, Map<String, Map<String, Long>>, Map<String,
Map<String, Long>>> {
    private final List<String> tokens;
    private final int window;
    private boolean done = false;

    // Speichert Tokens und Fenstergröße für die Analyse.
    public CooccurrenceAnalysisGatherer(List<String> tokens, int window)
{
        this.tokens = tokens;
        this.window = window;
    }

    // Liefert den initialen Zustand (leere verschachtelte HashMap).
    @Override
    public Supplier<Map<String, Map<String, Long>>> initializer() {
        return HashMap::new;
    }

    // Liefert den Integrator, der die gesamte Kookkurrenz-Analyse
ausführt.
    @Override
    public Integrator<Map<String, Map<String, Long>>, String, Map<String,
        Map<String, Long>>> integrator() {

        return (state, _, downstream) -> {

            // Führt die Analyse nur einmal aus (global Aggregation).
            if (!done) {
                done = true;

                // Äußere Schleife: Iteriert über jedes Wort als
                   Zentrumswort.
                for (int i = 0; i < tokens.size(); i++) {
                    String center = tokens.get(i);

                    // Stellt sicher, dass die innere Map für das
                       Zentrumswort existiert.
                    Map<String, Long> centerMap = state.computeIfAbsent(
```

```
                        center, k -> new HashMap<>());

                    // Innere Schleife: Iteriert über das Kontextfenster.
                    for (int j = i - window; j <= i + window; j++) {

                        // Prüft Randbedingungen und schließt das
                           Zentrumswort aus (j == i).
                        if (j < 0 || j >= tokens.size() || j == i) {
                            continue;
                        }

                        String neighbor = tokens.get(j);

                        // Erhöht den Zähler für das Nachbarwort.
                        centerMap.merge(neighbor, 1L, Long::sum);
                    }
                }

                // Schiebt das Endergebnis in den Ausgabestream.
                downstream.push(state);
            }

            // Beendet den Stream nach dem ersten Element, da das
               Endergebnis gesendet wurde.
            return false;
        };
    }

    // Finisher/Combiner sind für diese globale, aufeinanderfolgende
       Analyse optional.
}
```

Codebeispiel 10. Benutzerdefinierte Gatherer für Kookkurrenz-Analyse

Die Klasse `CooccurrenceAnalysisGatherer` implementiert die `Gatherer`-Schnittstelle für eine Kookkurrenz-Analyse.

Der Konstruktor dieser Klasse akzeptiert und speichert die vollständigen Tokens (`tokens`) und die Fenstergröße (`window`).

Die Methode `initializer` erzeugt eine leere Map als globalen Akkumulator.

Die Methode `integrator` enkapsuliert ganze Kernlogik:

- Führt die gesamte Analyse **nur einmal** aus (kontrolliert durch das `done`-Flag).
- Iteriert über *alle* Zentrumswörter und deren Nachbarn im definierten Fenster.
- Akkumuliert die Häufigkeiten der Nachbarwörter in der Zustands-Map (`state`).
- Nach Abschluss der Iteration wird das fertige Ergebnis (`state`) über `downstream.push(state)` in den Ausgabestream geschoben.
- Der Rückgabewert `false` beendet die Stream-Verarbeitung nach der einmaligen Ausführung, da das Endergebnis bereits vorliegt.

Dieser Gatherer ermöglicht es, die Verarbeitungskette zu bereinigen, indem er die gesamte Arbeitslogik und den erforderlichen Status in sich selbst kapselt.

```
public static Map<String, Map<String, Long>>
performCooccurrenceAnalysisWithGatherer(
       int window, List<String> tokens) {

   // Randfallprüfung: Ungültige window-Größe.
   if (window < 1 || window > tokens.size()) {
       return Map.of();
   }

   // Der Gatherer kapselt die gesamte globale Analyse und gibt nur das
Endergebnis (die Map) aus.
   return tokens.stream()
           .gather(new CooccurrenceAnalysisGatherer(tokens, window))
           .findFirst()
           .orElseThrow(() -> new IllegalStateException(
               "Gatherer sollte ein Ergebnis liefern."));
}
```

Codebeispiel 11. Kookkurrenz-Analyse mit Stream Gatherers

Die Funktion `performCooccurrenceAnalysisWithGatherer` nutzt den oben erwähnten Gatherer `(CooccurrenceAnalysisGatherer)`. Dieser Gatherer kapselt die gesamte Komplexität der Fensteranalyse und des Akkumulationsprozesses funktional.

Das Ergebnis wird in einem einzigen Element am Ende des Streams ausgegeben, weshalb die Verarbeitungskette mit `.findFirst().orElseThrow()` abgeschlossen wird, um die resultierende Map abzurufen.

```
public static Map<String, Long> findTopKCooccurrences(String centerWord,
int k, Map<String, Map<String, Long>> cooccurrenceMap) {
   // Überprüft, ob das Schlüsselwort (centerWord) in der
      Kookkurrenz-Map vorhanden ist.
   if (!cooccurrenceMap.containsKey(centerWord.toLowerCase())) {
       return Map.of();
   }

   Map<String, Long> cooccurrences =
       cooccurrenceMap.get(centerWord.toLowerCase());

   return cooccurrences.entrySet().stream()
           // Absteigend nach Häufigkeit (Wert) sortieren
           .sorted(Collections.reverseOrder(comparingByValue()))
           .limit(k) // Nur die K häufigsten behalten
           // Sammle in einer Map, die die Sortierreihenfolge beibehält
           .collect(toMap(
                   Map.Entry::getKey,
                   Map.Entry::getValue,
                   (e1, e2) -> e1,
                   LinkedHashMap::new));
}
```

Codebeispiel 12. Funktion zur Bestimmung der-k-häufigsten Kookkurrenzen

Die Funktion `findTopKCooccurrences` findet die `k` häufigste Wörter, die am häufigsten mit einem bestimmten Schlüsselwort (`centerWord`) zusammen auftreten. Sie nimmt alle Kookkurrenz-Wörter des Schlüsselworts, sortiert diese nach ihrer Häufigkeit (wie oft sie zusammen vorkommen) und gibt dann nur die `k` häufigsten Wörter in dieser absteigenden Reihenfolge zurück. Ist das Schlüsselwort unbekannt, liefert die Methode eine leere Map.

5.3 Hilfsfunktionen

5.3.1 Tokenisierung und Normalisierung

In der Computerlinguistik bilden die Tokenisierung und die Normalisierung die essenziellen Vorverarbeitungsschritte (Preprocessing), um unstrukturierten Text in ein für Algorithmen verarbeitbares Format zu überführen.

Die **Tokenisierung** bezeichnet den Prozess der Segmentierung eines kontinuierlichen Textstroms in Tokens**.**

Die **Normalisierung** umfasst die Standardisierung des Textkorpus, um die Varianz in den Daten zu reduzieren und die Vergleichbarkeit der Token zu erhöhen. Dies beinhaltet primär die Bereinigung des Textes von irrelevanten Elementen (z. B. Interpunktionszeichen oder Sonderzeichen) sowie die Vereinheitlichung der Schreibweise (z. B. die Umwandlung aller Buchstaben in Kleinschreibung).

```
public static List<String> produceTokensFromText(String text) {
   // 1. Text in Kleinbuchstaben umwandeln
   text = text.toLowerCase();

   // 2. Explizite Definition der Wort-Zeichenklasse
   // [a-z] = Basis-Alphabet, [äöüß] = Deutsche Sonderzeichen
```

```
    Pattern wordPattern = Pattern.compile("[a-zäöüß]+");

    // 3. Alle passenden Wörter suchen und als Tokens extrahieren
    return wordPattern.matcher(text)
            .results() // Stream der gefundenen Matches
            .map(match -> match.group())
            .toList();
}
```

Codebeispiel 13. Hilfsfunktion für Tokenisierung

Die Hilfsfunktion `produceTokensFromText` führt die grundlegende Vorverarbeitung des Eingabetextes durch. Zuerst wird der gesamte Text in Kleinbuchstaben umgewandelt. Anschließend wird er in einzelne Wörter zerlegt, wobei Satzzeichen und Zahlen (durch regulären Ausdruck: `[a-zäöüß]+`) entfernt werden. Zuletzt werden die leere Tokens herausgefiltert. Das Ergebnis ist eine saubere `List<String>`, die für die weitere lexikalische Analyse bereitsteht.

5.3.2 Laufzeitmessung

```
public static void measureTimeToExecute(
        String operationName, Runnable operation,
        int warmupRuns, int measureRuns) {

    // 1. Aufwärmen: Code 'warmupRuns'-mal ausführen, um JIT-Effekte zu
          reduzieren.
    for (int i = 0; i < warmupRuns; i++) {
        operation.run();
    }

    // 2. Messung
    long sumOfNanos = 0;

    for (int i = 0; i < measureRuns; i++) {
        // Startzeit in Nanosekunden erfassen
        long start = System.nanoTime();

        // Code ausführen
```

```
        operation.run();

        // Dauer zu Summe hinzufügen
        sumOfNanos += (System.nanoTime() - start);
    }

    // 3. Ergebnisberechnung und -ausgabe
    // Berechnung der durchschnittlichen Zeit in Nanosekunden
    double averageNanos = (double) sumOfNanos / measureRuns;

    // Umrechnung in Millisekunden (1 ms = 1.000.000 ns)
    double averageMillis = averageNanos / 1_000_000.0;

    // Ausgabe mit Formatierung für bessere Lesbarkeit
    System.out.printf("%s: Mittelzeit = %.4f ms (Über %d Messläufe)%n",
            operationName, averageMillis, measureRuns);
}
```

Codebeispiel 14. Hilfsfunktion für Laufzeitmessung

Die Hilfsfunktion `measureTimeToExecute` dient zur parametrisierbaren Leistungsanalyse einer Operation. Zunächst führt sie die Operation `warmupRuns`-mal aus, um den Java JIT-Compiler zu aktivieren und Optimierungen zu ermöglichen. Anschließend wird die durchschnittliche Ausführungszeit über `measureRuns` Messdurchläufe ermittelt, wobei die Zeit hochpräzise in Nanosekunden erfasst wird. Das Endergebnis wird für eine bessere Lesbarkeit in Millisekunden (ms) ausgegeben und dient als Grundlage für den Vergleich der Leistung verschiedener Implementierungen.

5.3.3 Lesen einer Textdatei

```
public static String readTextFileAndSetToString(String filePath) {
    Path path = Paths.get(filePath);
    try {
        // Gibt den Inhalt der Textdatei zurück
        return Files.readString(path, StandardCharsets.UTF_8);
    } catch (IOException e) {
      // Wenn die Datei nicht gefunden wird oder ein anderer I/O-Fehler
```

```
auftritt,
     // wird der Fehler behandelt und ein leere String zurückgegeben
       System.err.println("Fehler beim Lesen der Datei "
           + filePath + ": " + e.getMessage());

       return "";
   }
}
```

Codebeispiel 15. Hilfsfunktion zum Lesen einer Textdatei

Die folgende Hilfsfunktion `readTextFileAndSetToString` wurde entwickelt, um den gesamten Inhalt einer Textdatei in eine einzige Zeichenfolge zu laden. Bei einem Fehler (z.B. wenn die Datei nicht existiert) wird der Fehler in der Konsole gemeldet und eine leere Zeichenkette zurückgegeben.

5.3.4 Speichern in JSON-Datei

```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;

public static void writeToJsonFile(Object data, String fileName) throws
IOException {
   Gson gson = new GsonBuilder()
           .setPrettyPrinting()
           .create();

   String json = gson.toJson(data);
   Files.writeString(Path.of(fileName), json);
}
```

Codebeispiel 16. Hilfsfunktion zum Speichern die Ergebnisse in JSON-Datei

Diese Funktion formatiert beliebige Java-Daten (`Objekt`) in das JSON-Format und speichert das Ergebnis in der angegebenen Datei (`fileName`). Dazu wurde die beliebte Java-Bibliothek *GSON* angewendet.

Kapitel 6 Experiment und Auswertung

Um die Funktionsweise des realisierten Programms zu demonstrieren, wurde die deutsche Übersetzung des Romans *"Zwanzigtausend Meilen unter dem Meer"* des bekannten französischen Schriftstellers Jules Verne[3] ausgewählt. Der Text stammt aus dem *Corpus of German-Language Fiction* [4].

Vor der Anwendung des Programms wurden die Daten manuell bearbeitet. Konkret wurden zwei Teile des Buches (die getrennt waren) zu einem zusammengefügt und unnötige Wörter wie Titel und Kapitelnummern entfernt.

6.1 Messumgebung: Hard- und Software

Die Entwicklung und die Durchführung der Leistungstests erfolgten innerhalb der *IntelliJ IDEA Community Edition*[4] (IDE) in Kombination mit dem *Amazon Corretto 24.0.2 JDK*[5].

Zur Gewährleistung der Nachvollziehbarkeit wurden die Messungen auf der nachfolgend aufgeführten Hardware-Umgebung durchgeführt:

Komponente	**Spezifikation**
Prozessor	AMD Ryzen 5 7500U (8 Kerne, 1,80 GHz)
Arbeitsspeicher	16 GB DDR4
Betriebssystem	Endeavour OS x86_64

Tabelle 1. Hardware-Spezifikationen

[3] Jules Verne (Biographie) - https://de.wikipedia.org/wiki/Jules_Verne
[4] *Intellij IDEA* - https://www.jetbrains.com/idea
[5] *Amazon Coretto JDK*- https://github.com/corretto/corretto-24

6.2 Beispiel für die Verwendung

Als Beispiel für die Anwendung des Programms wurden die folgenden Parameter für die Ausführung der Aufgaben (n-Gramme-Erzeugung und Kookkurrenz-Analyse) festgelegt:

- $n = 3$ zur - Trigramme-Erzeugung
- $windowSize = 2$ - zur Kookkurrenz-Analyse
- $centerWord = "See"$ und $k = 5$ - um fünf Kookkurrenzen von diesem Wort zu finden.

```
public static void main(String[] args) {
   String text = readTextFileAndSetToString(
       "src/data/Jules_Verne_Zwanzigtausend_Meilen_unter_dem_Meer.txt");

   // Tokenisierung
   List<String> tokens = produceTokensFromText(text);

   // Parameter
   int n = 3;
   int windowSize = 2;
   String centerWord = "See";
   int k = 5;

   List<String> nGramsWithoutGatherer = produceNGramsWithoutGatherer(n,
       tokens);
   List<String> nGramsWithGatherer = produceNGramsWithGatherer(n,
       tokens);

   // Überprüfen die Ergebnisse auf Ähnlichkeit.
   System.out.println("n-Gramme-Erzeugung Identität: " +
       nGramsWithoutGatherer.equals(nGramsWithGatherer));

   Map<String, Map<String, Long>> cooccurrencesWithoutGatherer =
       performCooccurrenceAnalysisWithoutGatherer(windowSize, tokens);

   Map<String, Map<String, Long>> cooccurrencesWithGatherer =
       performCooccurrenceAnalysisWithGatherer(windowSize, tokens);

   // Überprüfen die Ergebnisse auf Identität.
   System.out.println("Kookkurrenz-Analyse Identität: " +
cooccurrencesWithoutGatherer.equals(cooccurrencesWithGatherer)  + "\n");
```

```
    // Finden k (k = 5) häufigste Kookkurrenzen für das Wort "See"
    Map<String, Long> topCooccurrencies =
    findTopKCooccurrences(centerWord, k, cooccurrencesWithoutGatherer);

    System.out.println(k + " häufigste Kookkurrenzen für das Wort " +
        centerWord + ": " + topCooccurrencies);

    try {
        // Speichern der Ergebnisse in Json-Dateien
        writeToJsonFile(nGramsWithoutGatherer,
            "src/result/n-Gramme-mit-Gatherer.json");
        writeToJsonFile(nGramsWithGatherer,
            "src/result/n-Gramme-ohne-Gatherer.json");

        writeToJsonFile(cooccurrencesWithoutGatherer,
            "src/result/Cooccurrences-mit-Gatherer.json");
        writeToJsonFile(cooccurrencesWithGatherer,
            "src/result/Cooccurrences-Gramme-ohne-Gatherer.json");

    } catch (IOException e) {
        // (I/O) Fehlerbehandlung
        System.out.println("Fehler wurde passiert: " + e.getMessage());
    }
}
```

Codebeispiel 17. Demonstration der Anwendung der implementierten Funktionen

Dieses Programm:

- liest den vorbereitenden Text ein und tokenisiert ihn
- führt die Trigramm-Erzeugung und die Kookkurrenz-Analyse sowohl mit der herkömmlichen Stream-API als auch mit den Stream Gatherers durch.
- Prüft die Ergebnisse beider Methoden auf **Identität**
- Findet die fünf häufigsten Kookkurrenzen für das Zentrumswort "See".
- Speichert alle generierten Listen und Maps in JSON-Dateien.

```
n-Gramme-Erzeugung Identität: true
Kookkurrenz-Analyse Identität: true

5 häufigste Kookkurrenzen für das Wort See: {auf=12, in=10, hoher=10, der=8, ein=6}

Process finished with exit code 0
```

Abbildung 3. Ergebnis des Programms in Konsole (siehe Codebeispiel 17)

Anmerkung: Beide Ergebnisse für beide Aufgaben sind identisch.

```
[
  "eine schweifende klippe",
  "schweifende klippe ein",
  "klippe ein seltsames",
  "ein seltsames ereigniß",
  "seltsames ereigniß ein",
  "ereigniß ein unerklärtes",
  "ein unerklärtes und",
  "unerklärtes und eine",
  "und eine unerklärbare",
  "eine unerklärbare naturerscheinung",
```

Abbildung 4. Die ersten zehn n-Gramme aus der resultierenden JSON-Datei (siehe Codebeispiel 17)

```
{
  "skelette": {
    "polypen": 1,
    "von": 1,
    "meter": 1,
    "große": 1
  },
  "merkwürdigsten": {
    "der": 2,
    "sondirungen": 1,
    "die": 1,
    "angetroffen": 1,
    "im": 1,
    "einer": 1,
    "sind": 1,
    "und": 1,
    "fische": 1,
    "schönsten": 1,
    "verzeichnet": 1
  },
  "seeschlangen": {
    "der": 1,
    "kraken": 1,
    "des": 1,
    "moby": 1
  },
```

Abbildung 5. Einige Kookkurrenzen aus der resultierenden JSON-Datei (siehe Codebeispiel 17)

6.3 Leistungsanalyse

Die Leistungsanalyse zielt darauf ab, die Laufzeiteffizienz der Implementierungen mit Stream Gatherers im Vergleich zur herkömmlichen Stream-API zu bewerten. Hierfür wurden Skalierbarkeitstests anhand variierender Schlüsselparameter durchgeführt:

- **n-Gramm-Erzeugung:** Getestet wurden die Längen: (n) 2, 3, 4, 5.
- **Kookkurrenz-Analyse:** Getestet wurden die Fenstergrößen: 2, 3, 4, 5.

Jede Messung wurde mit fünf **Warm-up-Läufen** zur JIT-Kompilierung und **zehn Messläufen** zur Mittelwertbildung durchgeführt, um präzise und stabile Ergebnisse zu erzielen. Dazu wurde die Hilfsfunktion `measureTimeToExecute` verwendet (siehe Codebeispiel 14).

```
public static void main(String[] args) {
   String text = readTextFileAndSetToString(
       "src/data/Jules_Verne_Zwanzigtausend_Meilen_unter_dem_Meer.txt");

   List<String> tokens = produceTokensFromText(text);

   int warmUpRuns = 5;
   int measureRuns = 10;
   String nGramOp = "n-Gramme";
   String cooccurrenceOp = "Kookkurrenz-Analyse";

   String[] opMethods = new String[]{" (ohne Gatherers) ", " (mit
                                       Gatherers) "};
   int[] nParams = new int[]{2, 3, 4, 5};
   int[] windowParams = new int[]{2, 3, 4, 5};

   for (int i = 0; i < nParams.length; i++) {
       int n = nParams[i];

       measureTimeToExecute(
               nGramOp + opMethods[0] + "(n = " + n + ")",
               () -> produceNGramsWithoutGatherer(n, tokens),
               warmUpRuns, measureRuns);
```

```
        measureTimeToExecute(
                nGramOp + opMethods[1] + "(n = " + n + ")",
                () -> produceNGramsWithGatherer(n, tokens),
                warmUpRuns, measureRuns);

        System.out.println();
    }

    for (int i = 0; i < windowParams.length; i++) {
        int window = windowParams[i];

        measureTimeToExecute(
                cooccurrenceOp + opMethods[0] + "(window = " + window +
                    ")",
                () -> performCooccurrenceAnalysisWithoutGatherer(window,
                    tokens),
                warmUpRuns, measureRuns);

        measureTimeToExecute(
                cooccurrenceOp + opMethods[1] + "(window = " + window +
                    ")",
                () -> performCooccurrenceAnalysisWithGatherer(window,
                    tokens),
                warmUpRuns, measureRuns);

        System.out.println();
    }
}
```

Codebeispiel 18. Leistungsanalyse der implementierten Funktionen

```
n-Gramme (ohne Gatherers) (n = 2): Mittelzeit = 15.7108 ms (Über 10 Messläufe)
n-Gramme (mit Gatherers) (n = 2): Mittelzeit = 27.0373 ms (Über 10 Messläufe)

n-Gramme (ohne Gatherers) (n = 3): Mittelzeit = 26.7332 ms (Über 10 Messläufe)
n-Gramme (mit Gatherers) (n = 3): Mittelzeit = 30.0218 ms (Über 10 Messläufe)

n-Gramme (ohne Gatherers) (n = 4): Mittelzeit = 36.5091 ms (Über 10 Messläufe)
n-Gramme (mit Gatherers) (n = 4): Mittelzeit = 44.6771 ms (Über 10 Messläufe)

n-Gramme (ohne Gatherers) (n = 5): Mittelzeit = 47.1205 ms (Über 10 Messläufe)
n-Gramme (mit Gatherers) (n = 5): Mittelzeit = 50.5620 ms (Über 10 Messläufe)

Kookkurrenz-Analyse (ohne Gatherers) (window = 2): Mittelzeit = 132.0531 ms (Über 10 Messläufe)
Kookkurrenz-Analyse (mit Gatherers) (window = 2): Mittelzeit = 88.0586 ms (Über 10 Messläufe)

Kookkurrenz-Analyse (ohne Gatherers) (window = 3): Mittelzeit = 180.9425 ms (Über 10 Messläufe)
Kookkurrenz-Analyse (mit Gatherers) (window = 3): Mittelzeit = 133.7324 ms (Über 10 Messläufe)

Kookkurrenz-Analyse (ohne Gatherers) (window = 4): Mittelzeit = 237.6129 ms (Über 10 Messläufe)
Kookkurrenz-Analyse (mit Gatherers) (window = 4): Mittelzeit = 181.9133 ms (Über 10 Messläufe)

Kookkurrenz-Analyse (ohne Gatherers) (window = 5): Mittelzeit = 289.8968 ms (Über 10 Messläufe)
Kookkurrenz-Analyse (mit Gatherers) (window = 5): Mittelzeit = 227.6008 ms (Über 10 Messläufe)
```

Abbildung 6. Ergebnisse der Leistungsanalyse in Konsole

6.4 Bewertungen

6.4.1 Bewertungskriterien

Die abschließende Bewertung der Implementierungen mit und ohne Stream Gatherers basiert auf zwei zentralen Kriterien, die sowohl die technische Effizienz als auch die praktische Anwendbarkeit beleuchten:

1. **Leistung:** Ermittlung der schnellsten Methode zur Verarbeitung großer Datenmengen.
2. **Entwicklungskomplexität:** Bewertung der Implementierungshärte, des Wartungsaufwands und der Lesbarkeit der jeweiligen Lösung.

6.4.2 Leistung

n–Gramme-Erzeugung

n	Ansatz ohne Gatherers	Ansatz mit Gatherers
2	15.7108 ms	27.0373 ms
3	26.7332 ms	30.0218 ms
4	36.5091 ms	44.6771 ms
5	47.1205 ms	50.5620 ms

Tabelle 2. Ergebnisse der Leistungsanalyse für n-Gramm-Erzeugung

Die Erzeugung von n-Grammen mittels der klassischen Stream-API (Index-Methode) ist **deutlich schneller** als die Verwendung des Standard-Gatherers (`windowSliding`).

Der Grund dafür ist, dass der klassische Ansatz die Daten nur einmal durchläuft und der JIT-Compiler von der JVM diesen Code extrem effizient optimieren kann. Der Gatherers-Ansatz fügt hier unnötigen Mehraufwand (Overhead) hinzu.

Kookkurrenz-Analyse

window	Ansatz ohne Gatherers	Ansatz mit Gatherers
2	132.0531 ms	88.0586 ms
3	180.9425 ms	133.7324 ms
4	237.6129 ms	181.9133 ms
5	289.8968 ms	227.6008 ms

Tabelle 3. Ergebnisse der Leistungsanalyse für Kookkurrenz-Analyse

Bei der komplexeren Aufgabe der Kookkurrenz-Analyse, die einen globalen Zustand über ein großes Fenster speichern muss, sind die Gatherers **deutlich überlegen**.

Für die komplexe Aufgabe, bei der **zustandsbehaftete Logik** über mehrere Elemente hinweg nötig ist, sind die Gatherers schneller. Die Gatherers-Architektur ermöglicht es, den Zustand effizient zu verwalten und so den Mehraufwand (Overhead) der klassischen Index-basierten Logik zu vermeiden.

6.4.3 Entwicklungskomplexität

Die Bewertung konzentriert sich darauf, wie einfach der Code zu schreiben und zu warten ist (Lesbarkeit).

Stream-API (ohne Gatherers)

- Einfache Operationen (n-Gramme): Für einfache Aufgaben ist die alte Stream-API sehr leicht zu implementieren. Die Lösung ist kurz und schnell zu verstehen.
- Komplexe Operationen (Kookkurrenz): Für komplexe, zustandsbehaftete Aufgaben (wie Kookkurrenz-Analyse) ist der Ansatz ohne Gatherers schwierig. Man muss oft auf Lösungen wie `IntStream.range` zurückgreifen und die Elemente im Stream über Indizes suchen. Dies führt zu unklarem Code innerhalb der ansonsten deklarativen Stream-Kette.

Gesamtzusammenfassung Leistung: Der Ansatz unter Verwendung der klassischen Stream-API ist hinsichtlich der Leistungsfähigkeit bei der Ermittlung einfacher **n-Gramme überlegen**. Gleichzeitig unterliegt er dem Ansatz unter Verwendung neuer Gatherers bei der komplexeren und **zustandsbehafteten Aufgabe der Kookkurrenz-Analyse**.

Stream Gatherers

- Implementierungskomplexität: Die Entwicklung eines eigenen Gatherers erfordert ein tiefes Verständnis seiner vier Konzepte (Initializer, Integrator, Combiner, Finisher; siehe Absatz 2.3.2). Diese anfängliche Komplexität ist nötig, um hochflexible, wiederverwendbare und klar strukturierte Verarbeitungsschritte in der Stream-Verarbeitungskette zu erstellen.

- Lesbarkeit der Verarbeitungskette: Ist der Gatherer einmal definiert, ist die eigentliche Stream-Verarbeitungskette sauber und deklarativ. Der komplexe Zustand ist vollständig in die `Gatherer`-Klasse gekapselt. Die Lesbarkeit der gesamten Verarbeitungskette steigt dadurch stark an.

Gesamtzusammenfassung Entwicklungskomplexität: Die Gatherers erhöhen die anfängliche **Implementierungshärte** (man muss eine neue Schnittstelle lernen), aber sie verbessern die **Wartbarkeit und Lesbarkeit** der Stream-Verarbeitungskette für alle komplexen Operationen deutlich. Sie bieten die einzig architektonisch saubere Lösung für zustandsbehaftete Zwischenoperationen.

Kapitel 7 Zusammenfassung

Durch diese Arbeit wurde die Funktionsweise der Stream Gatherers in Java untersucht. Es wurde gezeigt, wie Stream Gatherers benutzerdefinierte, zustandsbehaftete Zwischenoperationen zu erstellen ermöglichen, und wie sie die Datenverarbeitung in Stream-Verarbeitungskette verbessern. Außerdem wurden zwei Ansätze zur Datenverarbeitung (ohne und mit Stream Gatherers) hinsichtlich ihrer Leistungsfähigkeit und Entwicklungskomplexität bewertet.

Dank dieser Arbeit konnte ich mich besser in die Stream-API-Technologie vertiefen, habe die neue Stream Gatherers-Technologie kennengelernt und sie in der Praxis angewendet.

Zusammenfassend lässt sich sagen, dass die Stream Gatherer eine große Lücke schließen und eine bedeutende Funktionalität für die moderne, effiziente Datenverarbeitung in Java darstellen.

Quellenverzeichnis

[1] Stream Gatherers JEP - https://openjdk.org/jeps/485; letzter Besuch am 04.12.2025

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

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

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

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

Anhang: Werkzeugnutzung

DeepL (Übersetzer) - war das Hauptwerkzeug für die Übersetzung von Texten

Figma (Grafikeditor) - in dem die Bilder (Abb. 1 und Abb. 2) manuell erstellt wurden, da es keine passenden gab.

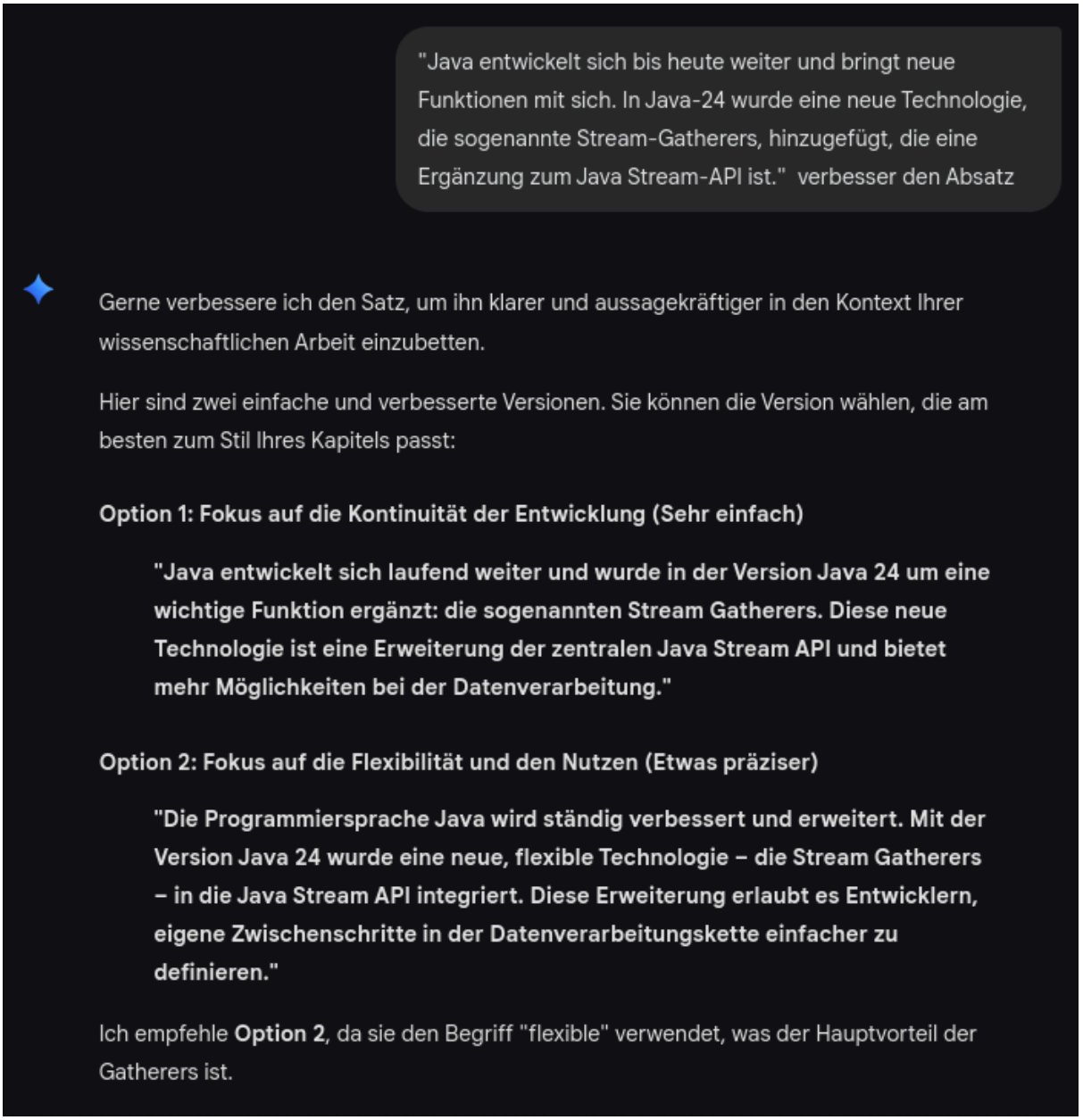
Google Docs (Texteditor) - Texteditor mit Korrekturfunktion, mit dem die wissenschaftliche Arbeit geschrieben wurde.

Code Blocks - Plugin für Google Docs, mit dem Code formatiert und ansprechend als Text dargestellt werden kann.

Auto-LaTeX equations - Plugin für Google Docs, mit dem mathematische Ausdrücke formatiert und ansprechend dargestellt werden können

Gemini (KI) - wurde zur Bereicherung des bereits geschriebenen Textes verwendet.

Nachfolgend ist ein Beispiel von KI-Verwendung:



Anmerkung: In diesem Fall wurde die erste Option ausgewählt, da es einfacher ist. Außerdem wurden Fehler, die mit dem fehlenden Bindestrich zwischen Java und 24 sowie Stream und API zusammenhingen, manuell korrigiert.