# Funktionale Programmierung III

Processing Order
(Stream Pipeline, Lazy Evaluation, terminal und intermediate operations)
Immutability und Parallelisierung
Exception Testing mit assertThrows (JUnit)
Checked Exception Handling

# Processing Order

Intermediate and Terminal Operations

Stream Pipeline

Lazy Evaluation

Durch mehrere Operationen auf Streams wird eine sogenannte Stream Pipeline aufgebaut.

intermediate operation

```
filter
b -> b.kampfsportBuecher.size()>=2
```

intermediate operation

```
map
b -> b.name
```

terminal operation

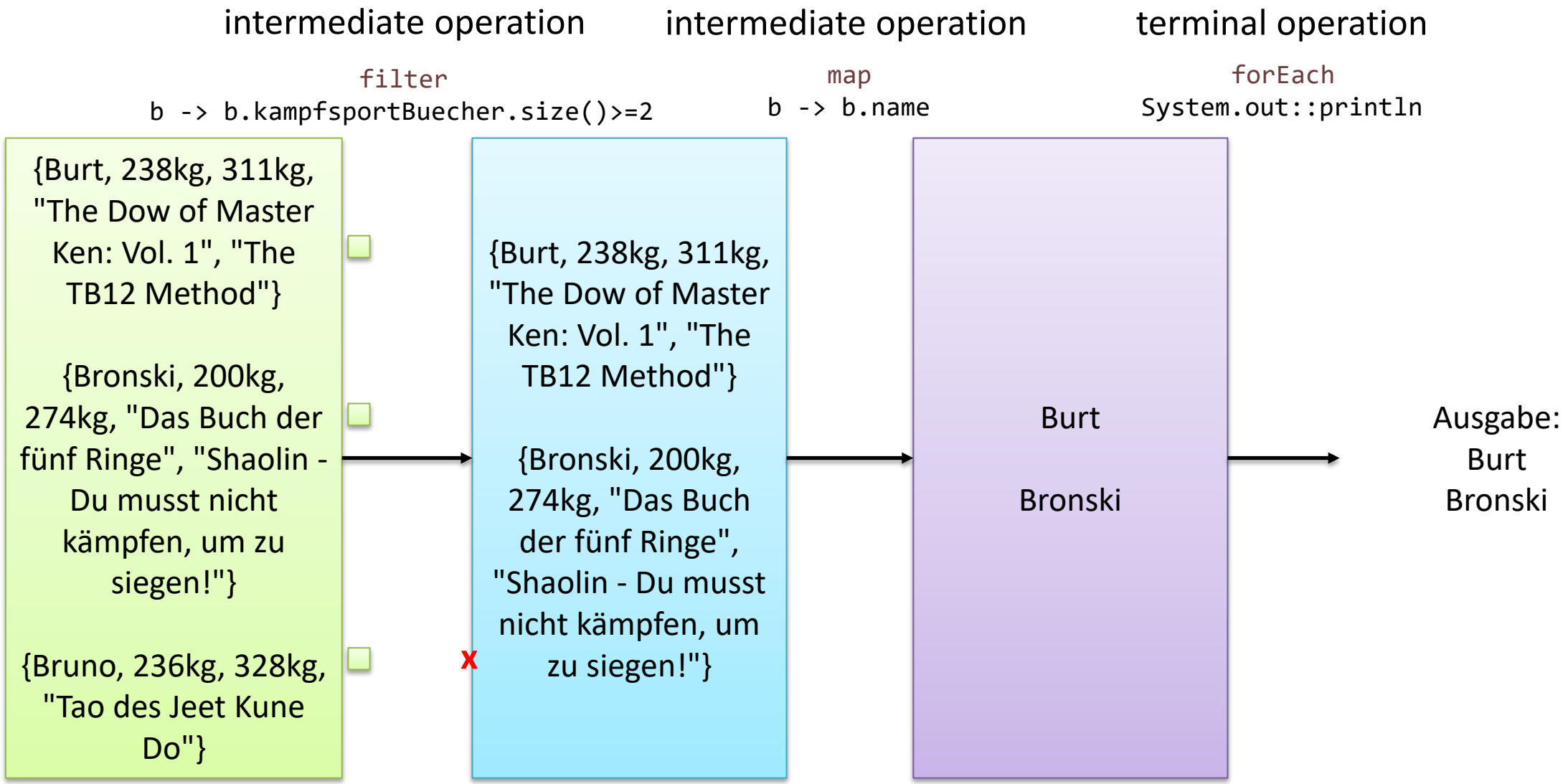
```
forEach
System.out::println
```

Eine intermediate operation hat einen Stream als Ergebnis, auf dem eine weitere intermediate oder terminal operation aufgerufen werden kann.

Eine terminal operation hat ein Nicht-Stream-Objekt als Ergebnis (i.d.R. eine Collection, void, ein Einzelwert, …).

Erst wenn eine terminal operation definiert wurde, wird die Stream Pipeline gestartet.

Achtung: Die Abarbeitung der kompletten Stream Pipeline erfolgt **pro Datensatz** des Streams.



```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
  new BBruder("Bronski", 200, 274).add("Das Buch der fünf Ringe",
                                       "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
  new BBruder("Bruno", 236, 328).add("Tao des Jeet Kune Do"));

bBrothers.stream()
  .filter(b ->
  {
    System.out.println("filter: " + b);
    return b.kampfsportBuecher.size()>=2;
  })
  .map(b ->
  {
    System.out.println("map: " + b);
    return b.name;
  })
  .forEach(n -> System.out.println("forEach: " + n));
```

Ausgabe:

```
filter: Name: Burt, Bankdrücken: 238kg, Kniebeugen 311kg
map: Name: Burt, Bankdrücken: 238kg, Kniebeugen 311kg
forEach: Burt
filter: Name: Bronski, Bankdrücken: 200kg, Kniebeugen 274kg
map: Name: Bronski, Bankdrücken: 200kg, Kniebeugen 274kg
forEach: Bronski
filter: Name: Bruno, Bankdrücken: 236kg, Kniebeugen 328kg
```

Die Auswertung erfolgt erst, sobald eine terminal operation auf der Stream Pipeline aufgerufen wurde.

Man sagt auch die Auswertung erfolgt **lazy**.

```
List<String> phoneBook = List.of("Albrecht", "Bauer", "Clemens", "Arndt", "Beck", "Christ",
  "Ackermann", "Berger", "Claus", "Adam", "Becker", "Conrad");

phoneBook.stream()
  .map(n -> {
    System.out.println("map: " + n);
    return n.toUpperCase();
  })
  .filter(n -> {
    System.out.println("filter: " + n);
    return n.startsWith("A");
  })
  .forEach(n -> {
    System.out.println("forEach: " + n);
   });
```

Was kann man an obigen Code verbessern?

Ausgabe:

```
map: Albrecht
filter: ALBRECHT
forEach: ALBRECHT
map: Bauer
filter: BAUER
map: Clemens
filter: CLEMENS
map: Arndt
filter: ARNDT
forEach: ARNDT
map: Beck
filter: BECK
map: Christ
filter: CHRIST
map: Ackermann
filter: ACKERMANN
forEach: ACKERMANN
map: Berger
filter: BERGER
map: Claus
filter: CLAUS
```

```
List<String> phoneBook = List.of("Albrecht", "Bauer", "Clemens", "Arndt", "Beck", "Christ",
  "Ackermann", "Berger", "Claus", "Adam", "Becker", "Conrad");

phoneBook.stream()
  .filter(n -> {
    System.out.println("filter: " + n);
    return n.startsWith("A");
  })
  .map(n -> {
    System.out.println("map: " + n);
    return n.toUpperCase();
  })
  .forEach(n -> {
    System.out.println("forEach: " + n);
  });
```

Wenn möglich verwendet man filter vor anderen Verarbeitungsschritten.

Dann müssen weitere Arbeitsschritte nicht mehr auf allen Elementen ausgeführt werden.

Ausgabe:

```
filter: Albrecht
map: Albrecht
forEach: ALBRECHT
filter: Bauer
filter: Clemens
filter: Arndt
map: Arndt
forEach: ARNDT
filter: Beck
filter: Christ
filter: Ackermann
map: Ackermann
forEach: ACKERMANN
filter: Berger
filter: Claus
filter: Adam
map: Adam
forEach: ADAM
filter: Becker
filter: Conrad
```

```
List<String> phoneBook = List.of("Albrecht", "Bauer", "Clemens", "Arndt", "Beck", "Christ",
  "Ackermann", "Berger", "Claus", "Adam", "Becker", "Conrad");

phoneBook.stream()
  .filter(n -> {
    System.out.println("filter: " + n);
    return n.startsWith("A");
  })
  .map(n -> {
    System.out.println("map: " + n);
    return n.toUpperCase();
  })
  .sorted()
  .forEach(n -> {
    System.out.println("forEach: " + n);
  });
```

Ausgabe:

```
filter: Albrecht
map: Albrecht
filter: Bauer
filter: Clemens
filter: Arndt
map: Arndt
filter: Beck
filter: Christ
filter: Ackermann
map: Ackermann
filter: Berger
filter: Claus
filter: Adam
map: Adam
filter: Becker
filter: Conrad
forEach: ACKERMANN
forEach: ADAM
forEach: ALBRECHT
forEach: ARNDT
```

Bei sorted (wie auch bei distinct) wird eine Ausführung der Stream Pipeline erzwungen, da zum Sortieren alle Elemente zur Verfügung stehen müssen. Daher sind sorted und distinct auch sog. **stateful intermediate operations.**

# Immutability
# Parallelisierung

Wir haben gesehen:

Funktionale Programmierung verzichtet auf Zustandsänderungen.

In unseren Programmen wurde nach jedem filter- oder map-Aufruf ein neuer Stream erstellt, die Daten des bestehenden Streams aber nicht verändert.

Die Daten **ändern** sich nicht (Immutability).

# Das einzige Beispiel, bei dem wir Daten außerhalb des Stream geändert haben:

```
List<BBruder> bBrothers = List.of(
  new BBruder("Burt", 238, 311).add("The Dow of Master Ken: Vol. 1", "The TB12 Method"),
  new BBruder("Bronski", 200, 274)
    .add("Das Buch der fünf Ringe", "Shaolin - Du musst nicht kämpfen, um zu siegen!"),
  new BBruder("Bruno", 236, 328).add("Tao des Jeet Kune Do"));

List<String> buecher = new ArrayList<>();
bBrothers.stream()
  .map(b -> b.kampfsportBuecher)
  .forEach(k -> buecher.addAll(k));
buecher.stream().forEach(System.out::println);
```

Diese Änderung konnten wir mit flatMap umgehen.

# Was sind Vorteile von Immutability?

Einfacheres Testen
Parallelisierung

Egal wann in einem zustandslosen Programm eine Funktion aufgerufen wird,
sie liefert bei gleicher Eingabe immer das gleiche Ergebnis.

Was heißt das für Tests?

assertEquals(f(x),f(x)) ist immer true.

Wenn dagegen eine Funktion Seiteneffekte hat (d.h. Zustandsänderungen vornimmt),
kann der wiederholte Aufruf von f(x) unterschiedliche Ergebnisse liefern.

Parallelisierung auf nicht veränderlichen Daten ist einfach.

Jeder Computer, jedes mobile Endgerät hat heute i.d.R. mehr als einen Prozessor.

Funktionale Programmierung erlaubt im Zusammenhang mit Collections eine einfache Parallelisierung.

```
List<String> bBrothers = Arrays.asList("Burt", "Bronski", … , "Peter");

List<String> collect = bBrothers.parallelStream()
  .filter(name -> name.startsWith("B"))
  .collect(Collectors.toList());
System.out.println(collect);
```

Nur sinnvoll, wenn die Collections groß
und die Operationen aufwendig genug sind.

Die Parallelisierung (Erzeugung von Threads, etc.) kostet Zeit.

Die Dokumentation beachten, welche Methoden für eine parallele Ausführung geeignet sind.

# Zum Ausprobieren:

```
public class ParallelHelloWorld
{
  public static void main(String[] args)
  {
    "Hello World".chars().parallel().forEach(c -> System.out.print((char) c));
  }
}
```

# Exception Testing in JUnit 5

```java
public class OberflaechenBerechnung
{
  public static String berechneOberflaeche(int a, int b, int c)
  {
    if (a < 0 || b < 0 || c < 0) throw new RuntimeException("Ungültiges Argument");
    return "Die Oberfläche des Quaders beträgt: " + (2*a*b+2*a*c+2*b*c);
  }
}
public class OberflaechenBerechnungTest
{
  @Test
  public void testFehlerfall()
  {
    try
    {
      berechneOberflaeche(1, 1, -1);
      fail("Runtime Exception erwartet");
    }
    catch(RuntimeException e)
    {
      String errorMessage = e.getMessage();
      assertEquals("Ungültiges Argument", errorMessage);
    }
  }
```

Unser bisheriger JUnit Test,
um das korrekte Auslösen einer Exception zu testen.

# Die Klasse Assertions stellt für Exception-Handling einige Methoden zur Verfügung.

org.junit.jupiter.api

**Assertions**

```
public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)
public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable,
                                                   String message)
public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable,
                                                   Supplier<String> messageSupplier)

public static void assertDoesNotThrow(Executable executable)
public static void assertDoesNotThrow(Executable executable, String message)
public static void assertDoesNotThrow(Executable executable, Supplier<String> messageSupplier)
public static <T> T assertDoesNotThrow(ThrowingSupplier<T> supplier)
public static <T> T assertDoesNotThrow(ThrowingSupplier<T> supplier, String message)
public static <T> T assertDoesNotThrow(ThrowingSupplier<T> supplier,
                                       Supplier<String> messageSupplier)
```

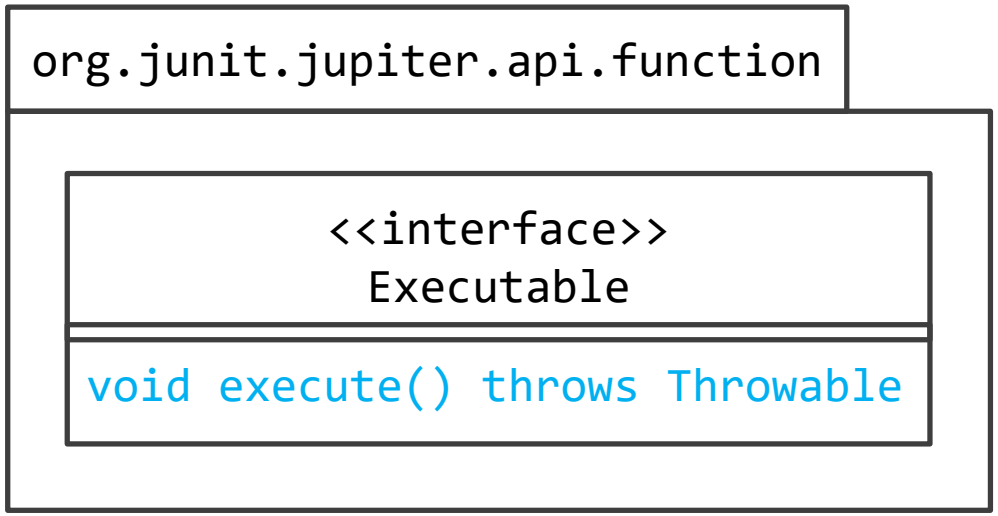
# Wir wollen folgende Methode verwenden:

org.junit.jupiter.api

**Assertions**

```
public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)
public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable,
                                                   String message)
public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable,
                                                   Supplier<String> messageSupplier)

public static void assertDoesNotThrow(Executable executable)
public static void assertDoesNotThrow(Executable executable, String message)
public static void assertDoesNotThrow(Executable executable, Supplier<String> messageSupplier)
public static <T> T assertDoesNotThrow(ThrowingSupplier<T> supplier)
public static <T> T assertDoesNotThrow(ThrowingSupplier<T> supplier, String message)
public static <T> T assertDoesNotThrow(ThrowingSupplier<T> supplier,
                                       Supplier<String> messageSupplier)
```

# Das Executable-Interface sieht wie folgt aus:




# Wie muss der Aufruf der assertThrows Methode

`public static <T extends Throwable> T assertThrows(Class<T> expectedType, Executable executable)`

# aussehen?

```
@Test
public void testFehlerfallMitAssertThrows()
{
    assertThrows(RuntimeException.class,                    );
}
```

```java
@Test
public void testFehlerfallMitAssertThrows()
{
    assertThrows(RuntimeException.class, () -> berechneOberflaeche(1, 1, -1));
}
```

oder

```java
@Test
public void testFehlerfallMitAssertThrowsMitAnonymerKlasse()
{
    assertThrows(RuntimeException.class, new Executable()
    {
      @Override
      public void execute() throws Throwable
      {
        berechneOberflaeche(1, 1, -1);
      }
    });
}
```

# Wenn wir die Fehlermeldung überprüfen wollen, können wir den Rückgabewert von assertThrows nutzen…

```
@Test
public void testFehlerfallMitAssertThrowsUndFehlermeldung()
{
  RuntimeException e = assertThrows(RuntimeException.class, () -> berechneOberflaeche(1, 1, -1));
  assertEquals("Ungültiges Argument", e.getMessage());
}
```

# Exception Handling in Lambda-Ausdrücken

# Wir versuchen ein „komplexes“ Mapping: von einer URL auf den Text einer Webseite.

```
List<String> urls = Arrays.asList(
  "https://en.wikipedia.org/wiki/Donald_Trump",
  "https://en.wikipedia.org/wiki/Angela_Merkel",
  "https://en.wikipedia.org/wiki/Vladimir_Putin");

urls.parallelStream()
    .map(url -> {


            return Jsoup.connect(url)
                .get()
                .text();




    })
    …
```

# Der Compiler zwingt uns innerhalb von Lambda-Ausdrücken checked Exceptions zu fangen.

```
List<String> urls = Arrays.asList(
  "https://en.wikipedia.org/wiki/Donald_Trump",
  "https://en.wikipedia.org/wiki/Angela_Merkel",
  "https://en.wikipedia.org/wiki/Vladimir_Putin");

urls.parallelStream()
    .map(url -> {
        try
        {
            return Jsoup.connect(url)
                .get()
                .text();
        }
        catch (IOException e)
        {

        }
    })
    …
```

# Wenn wir die Exception nicht beheben können, können wir sie als unchecked Exception weiterwerfen.

```
List<String> urls = Arrays.asList(
  "https://en.wikipedia.org/wiki/Donald_Trump",
  "https://en.wikipedia.org/wiki/Angela_Merkel",
  "https://en.wikipedia.org/wiki/Vladimir_Putin");

urls.parallelStream()
    .map(url -> {
        try
        {
            return Jsoup.connect(url)
                .get()
                .text();
        }
        catch (IOException e)
        {
            throw new RuntimeException(e);
        }
    })
    …
```

Checked Exceptions verhindern die Verwendung von kurzen, präzisen Lambda-Ausdrücken.

Als Workaround können wir neue Functional Interfaces zu den gängigen FunctionalInterfaces definieren.

```
@FunctionalInterface
public interface ThrowingFunction<T, R, E extends Exception>
{
  R apply(T t) throws E;

  static <T, R, E extends Exception> Function<T, R> unchecked(ThrowingFunction<T, R, E> function)
  {
    return t -> {
      try
      {
        return function.apply(t);
      }
      catch (Exception e)
      {
        throw new RuntimeException(e);
      }
    };
  }
}
```

Wir definieren eine Function mit drei statt zwei generischen Typen.

Die Signatur der apply-Methode von Function wird übernommen und um throws E erweitert.

Die statische Methode unchecked macht aus einem Lambda-Ausdruck, der das ThrowingFunction-Interface erfüllt, einen Lambda-Ausdruck, der das Function-Interface erfüllt.

# Jetzt ist folgender Aufruf möglich mit Hilfe des ThrowingFunction-Interfaces möglich:

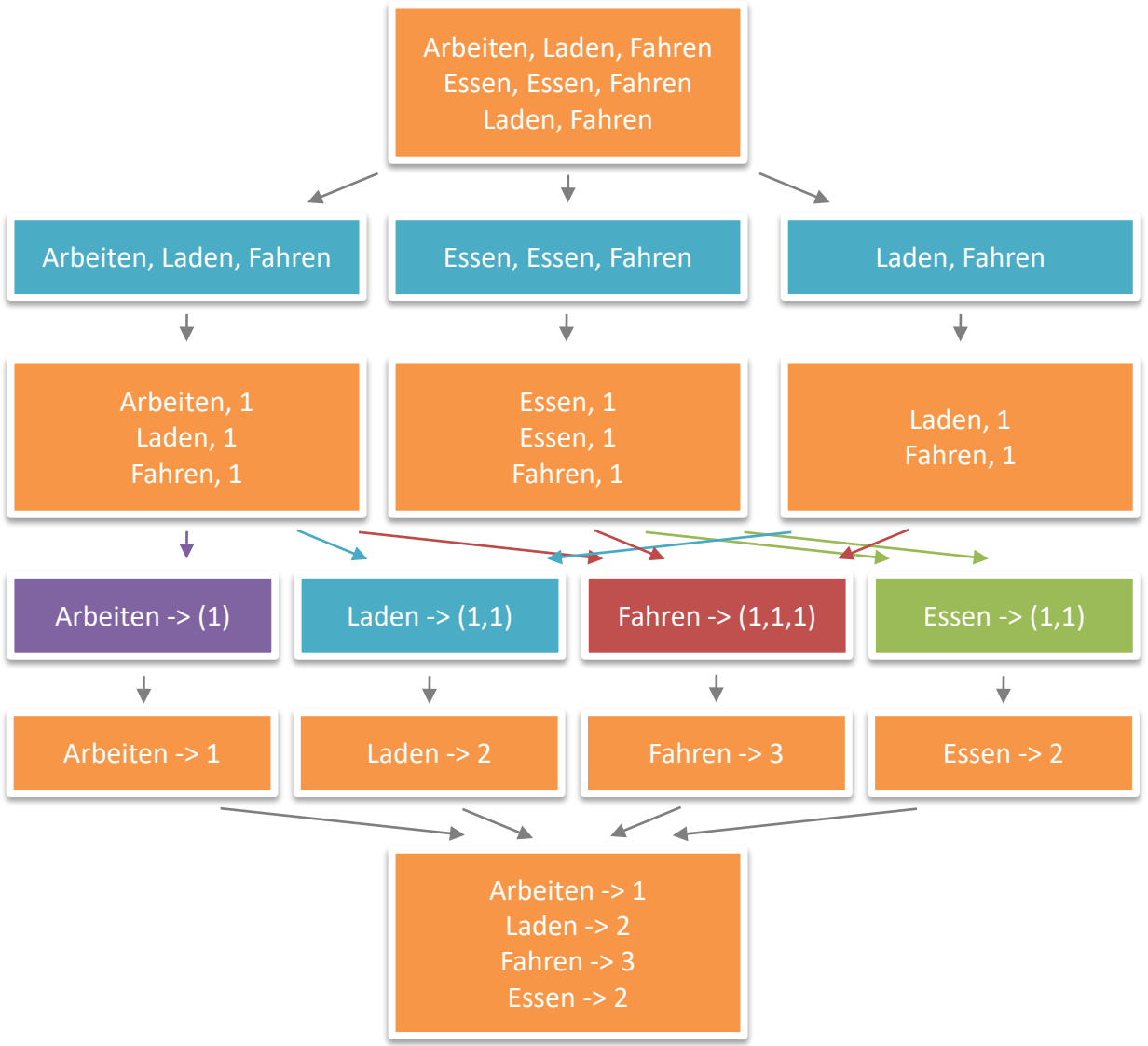
```
List<String> urls = Arrays.asList(
  "https://en.wikipedia.org/wiki/Donald_Trump",
  "https://en.wikipedia.org/wiki/Angela_Merkel",
  "https://en.wikipedia.org/wiki/Vladimir_Putin");

urls.parallelStream()
    .map(ThrowingFunction.unchecked(url ->
        Jsoup.connect(url)
            .get()
            .text()))
     …
```

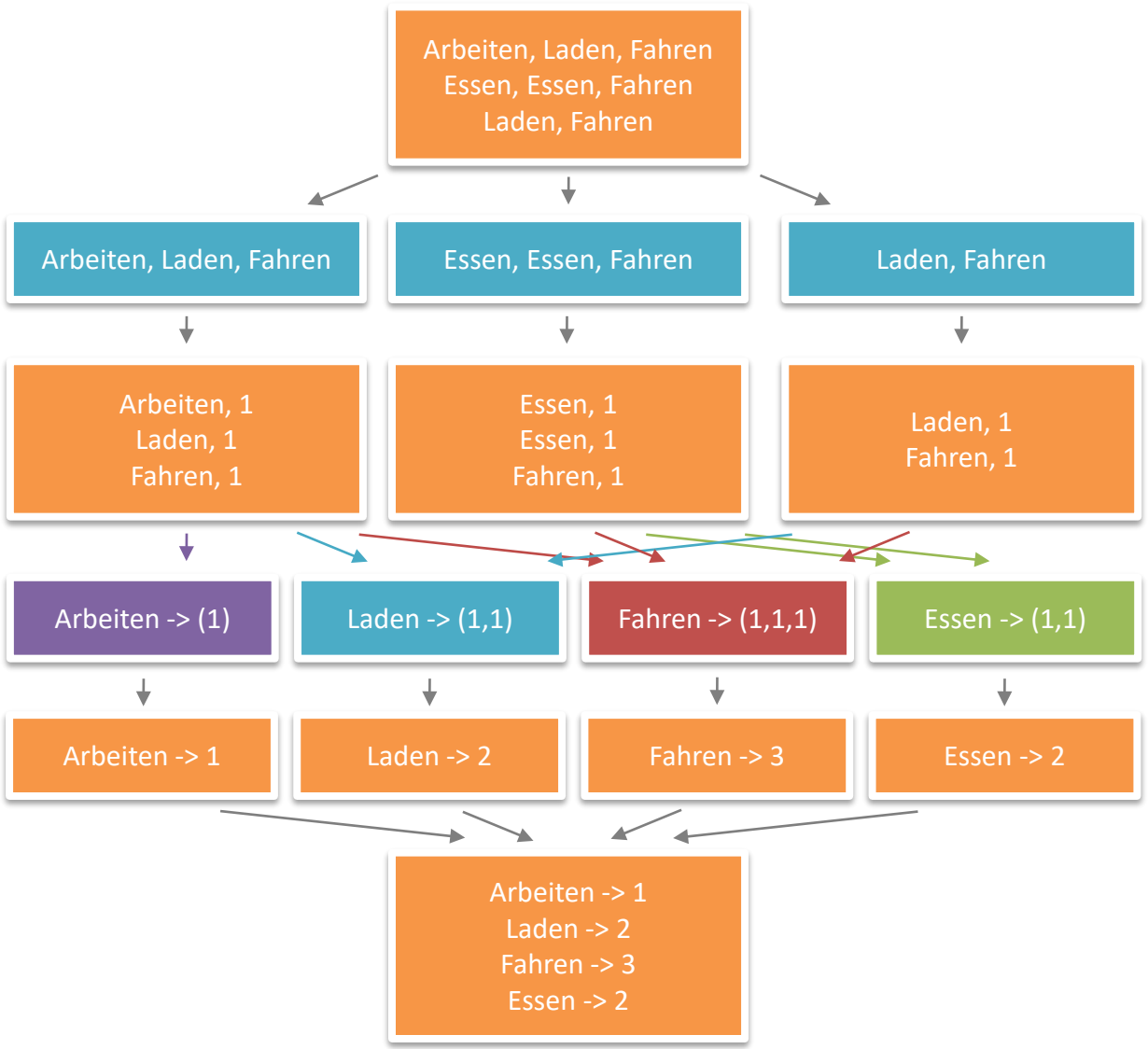
# Word Count mit Map Reduce

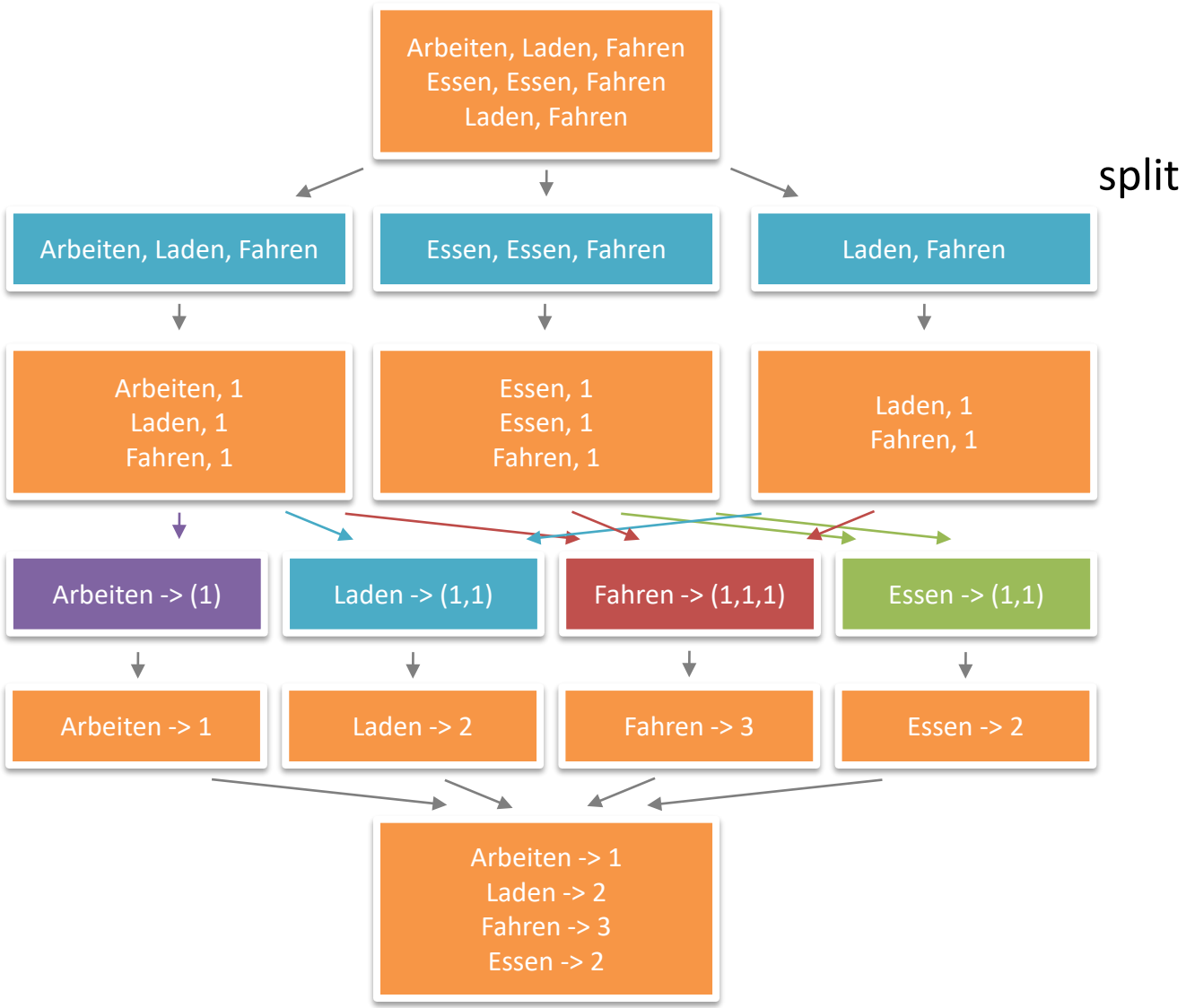
Wir wollen mit Java 8 Streams ein etwas komplexeres Problem lösen:

Das Zählen des Auftretens gleicher Wörter in einem oder verschiedenen Texten.

# Map-Reduce-Algorithmus am Word-Count-Beispiel

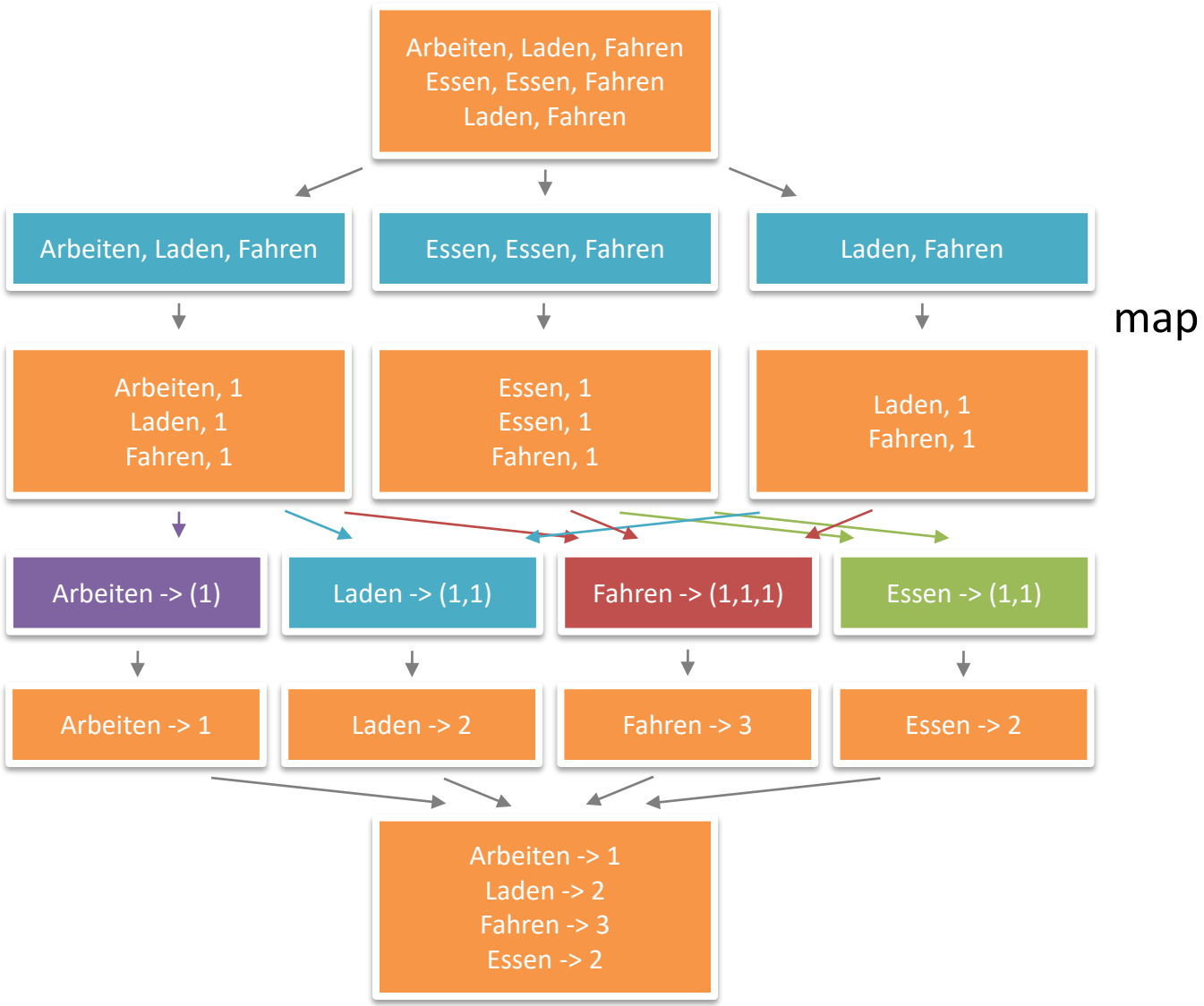







# Split

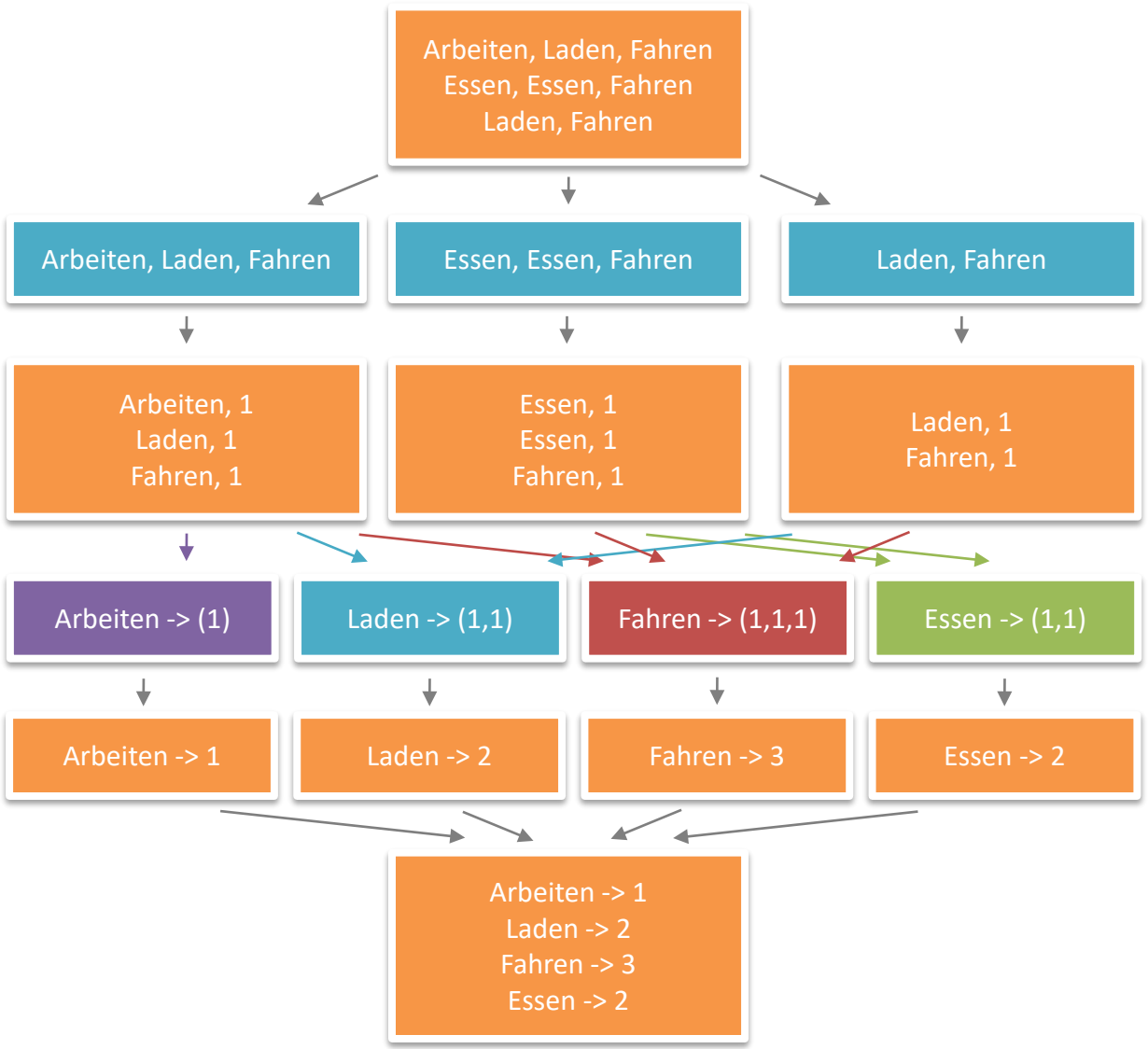
Ein Text wird aufgesplittet (bspw. zeilenweise, dokumentenweise, etc.) und auf verschiedene Nodes (Prozessoren, Threads, Rechner, …) verteilt.




# Map

Jede **Node** zählt, wie oft ein Wort auftritt, und speichert dies in einer Map mit Mappings von der Art:

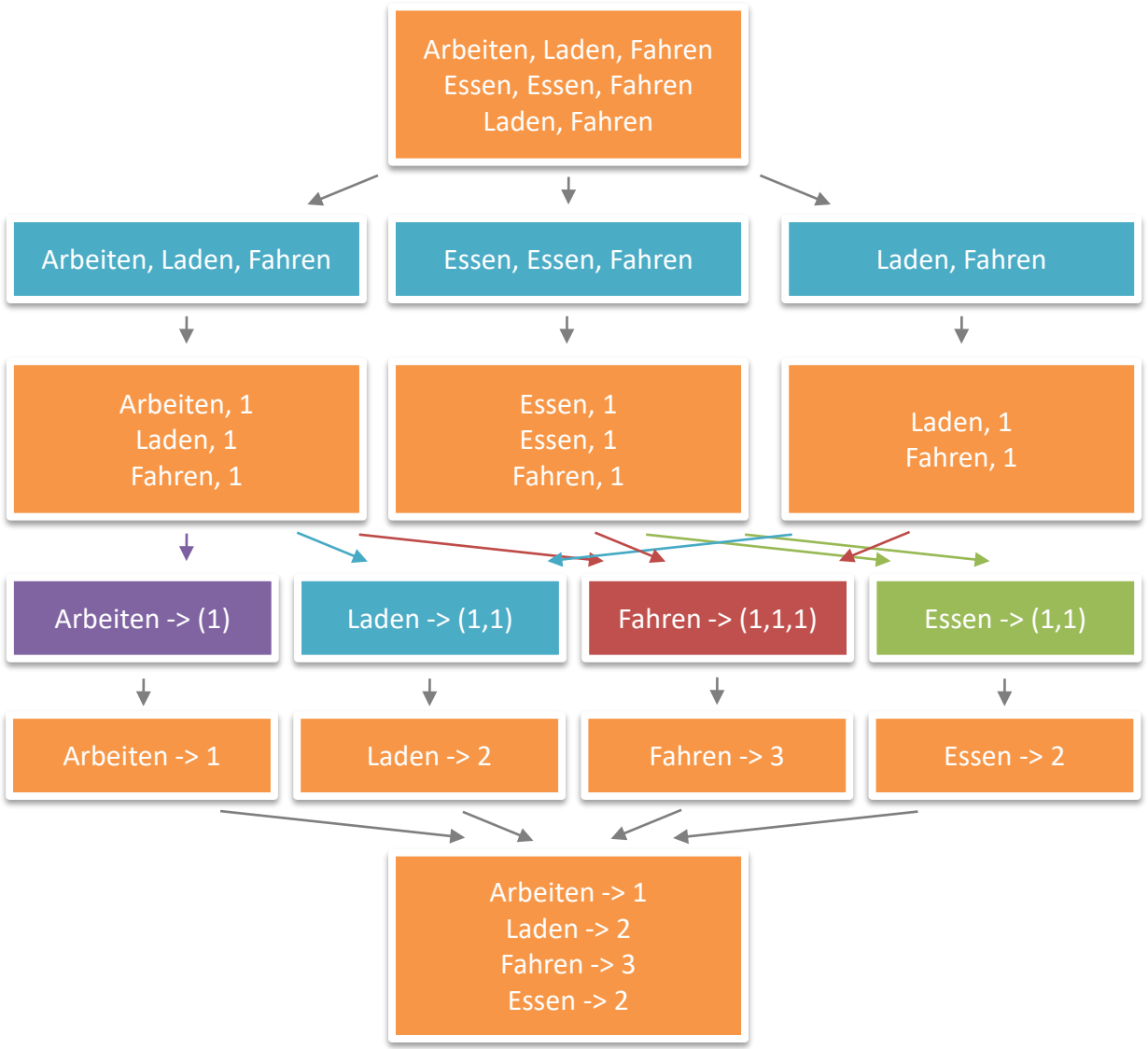
**Wort -> Auftrittshäufigkeit**




# Shuffle

Teilergebnisse werden auf verschiedenen **Nodes** zusammengeführt.

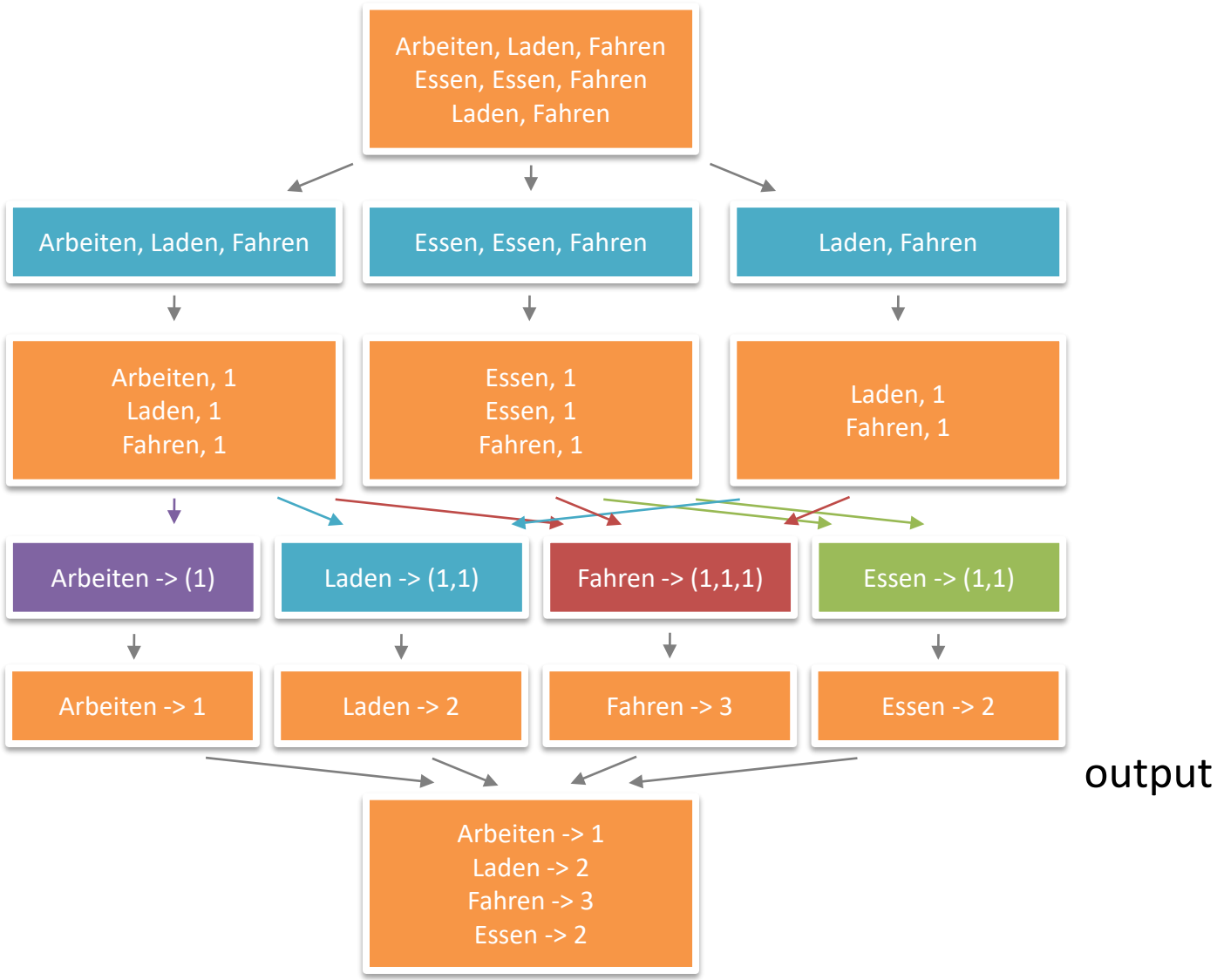
Jeder **Node** ist für eine bestimmte Menge von Wörtern verantwortlich.




reduce

# Reduce

Daraufhin findet eine Reduktion statt, indem die Teillisten der Auftrittshäufigkeiten zusammengelegt werden.




output

# Output

Alle Teilergebnisse werden wieder auf einer Node zusammengeführt.

Implementieren Sie den Word Count als Übung mit Java Streams (siehe Übungsaufgabe).

Überlegen Sie, welche Phasen Sie implementieren müssen und welche Phasen Ihnen automatisch abgenommen werden.