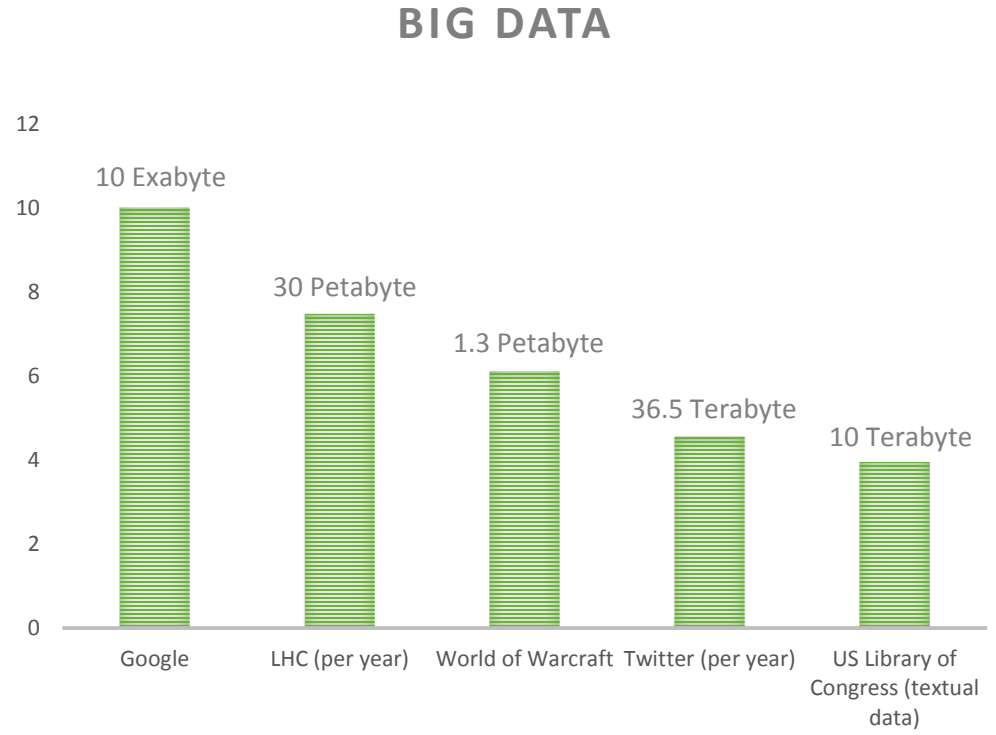
# BIG DATA

"*Big data is high-**volume**, high-**velocity** and high-**variety** information assets that demand **cost-effective**, **innovative** forms of information processing for **enhanced insight and decision making***." -- Gartner

# VOLUME

Woher kommt Big Data?

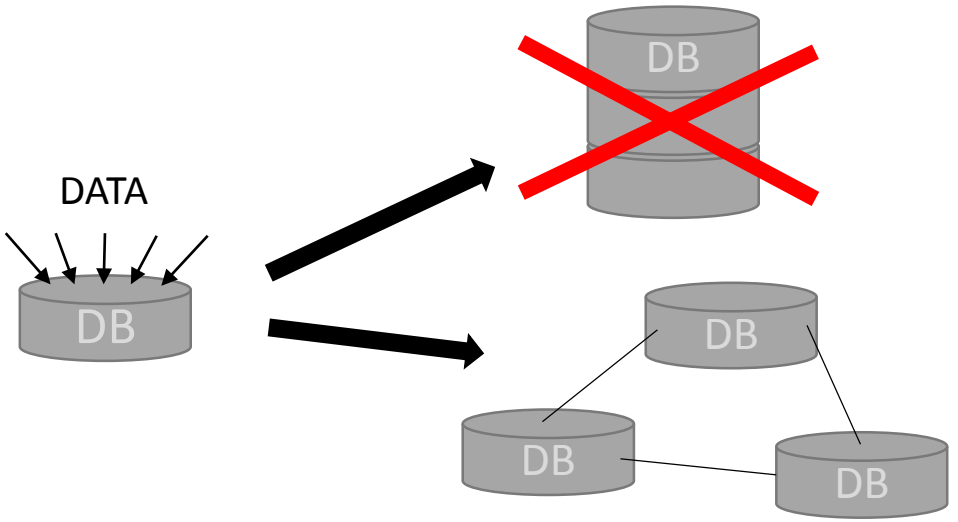
- Online: Klicks, Netzwerktraffic,…
- Social Media: Facebook, Twitter,…
- Gaming: MMOGs,…
- Health Care: Genforschung,…
- Wissenschaft: Large Hadron Collider,…
- http://www.internetlivestats.com/



# VOLUME

- Massive Datenmengen können nicht mehr auf einzelnen Maschinen gespeichert werden

- Lösung: Verteilen der Daten auf große Cluster
  - Data Warehouses für strukturierte Daten
  - Apache Hadoop Lösungen für unstrukturierte Daten



# VARIETY

- Daten sind oft unterschiedlich
  - Inkonsistente Typen (z.B. Preise in € oder $)
  - Inkonsistente Werte (z.B. Wal-Mart oder WalMart)
  - (Verschiedene) Datenquellen liefern verschiedene Daten (Twitter: Bilder oder Text)

- Datenstrukturspektrum:

| strukturiert | semi-strukturiert | unstrukturiert |
| --- | --- | --- |
| - Relationale Datenbank<br>- Formatierte Nachrichten | - Dokumente/Formulare<br>- XML Inhalte<br>- Gekennzeichnete Texte | - Plain Text<br>- Bilder/Videos |

# Variety

Beispiel: Apache Web Server Log Format

```
127.0.0.1 - - [10/Oct/2000:13:55:36 -0700] "GET /apache_pb.gif HTTP/1.0" 200 2326
```

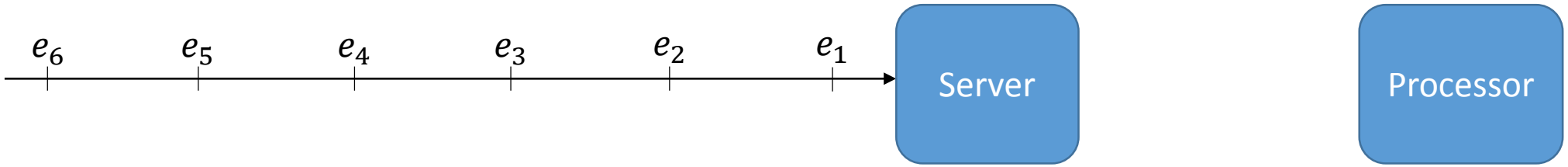
- Komponenten haben keine einheitlichen Typen
  → Die letzte Komponente (= size), kann eine Zahl oder auch – sein
- Nachdem die Komponenten erzeugt/vorverarbeitet wurden können sie in ein Schema gebracht werden

→ semi-strukturiert

# VELOCITY

- Durch steigende Datenmengen muss auch die Verarbeitungs-geschwindigkeit steigen

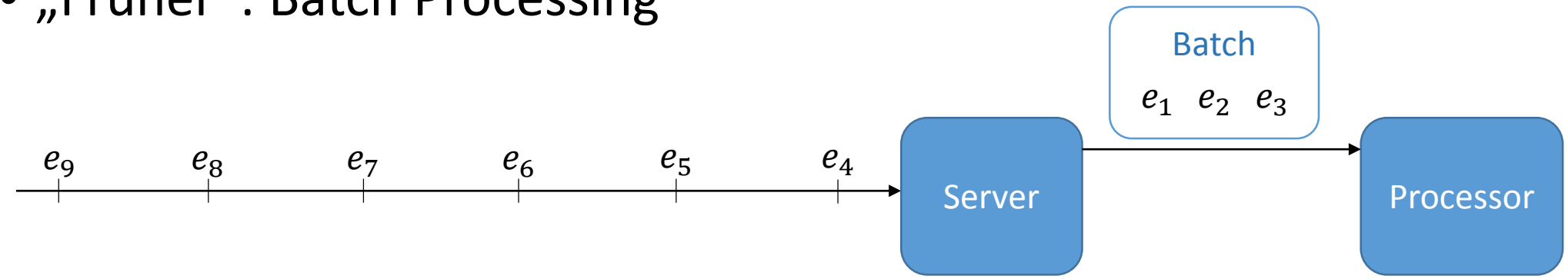
- „Früher“: Batch Processing



# VELOCITY

- Durch steigende Datenmengen muss auch die Verarbeitungs-geschwindigkeit steigen

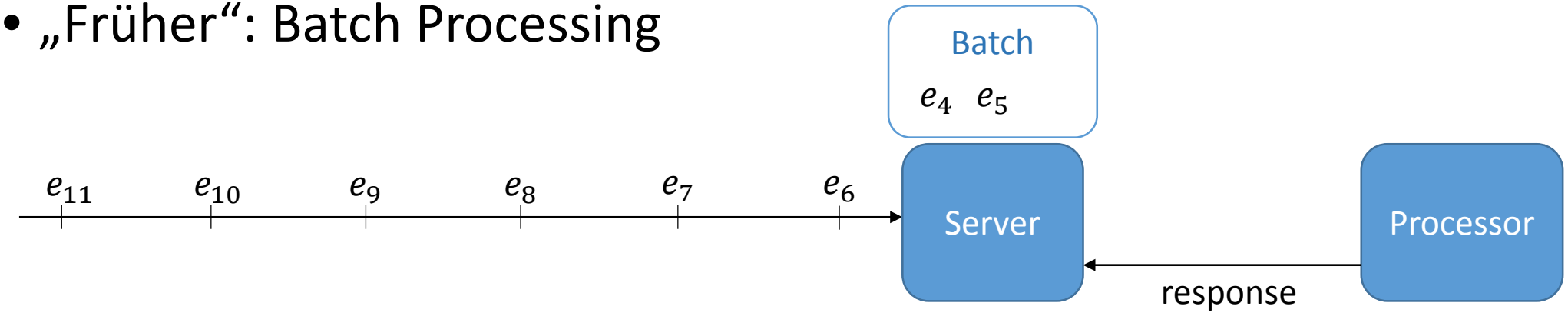
- „Früher“: Batch Processing



# VELOCITY

- Durch steigende Datenmengen muss auch die Verarbeitungs-geschwindigkeit steigen

- „Früher“: Batch Processing

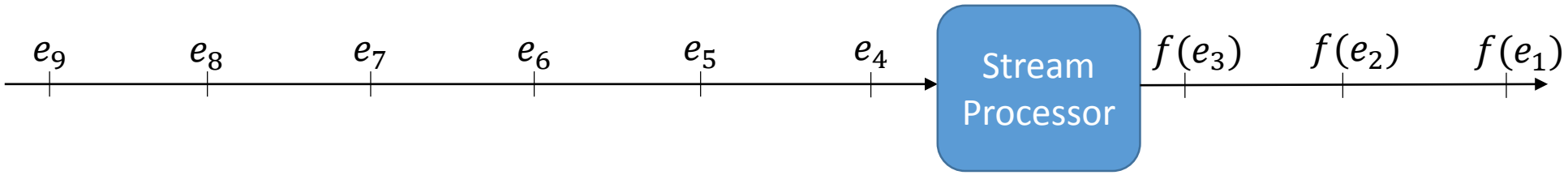



- Problem: Inputrate > Durchsatz

# VELOCITY

- Durch steigende Datenmengen muss auch die Verarbeitungs-geschwindigkeit steigen

- Heute: Stream Processing
  - Daten werden verarbeitet sobald sie registriert werden



# VELOCITY

Use Cases für Stream-Processing:

1. LHC: 600 Mio. Kollisionen à 1 MB Rohdaten / Sekunde
   - Problem: zu großer Input
   - Lösung: Echtzeit-Vorverarbeitung um nur interessante Events zu filtern

2. Recommendation Systems:
   - Problem: zu geringer Output
   - Lösung: Echtzeit-Analyse um möglichst schnell interessante Empfehlungen zu finden (meist inkrementelle Ansätze)

# BIG DATA SOLUTIONS

Apache Hadoop 


- Open Source Framework für skalierbares, verteiltes Rechnen
- Hauptkomponenten:
  - Hadoop File System (HDFS)
  - YARN Ressource Manager
  - Hadoop MapReduce
- Erweiterungen:
  - Hive: Data Warehouse-Funktionalitäten
  - ZooKeeper: Konfigurationstool für verteilte Systeme
  - …

# BIG DATA SOLUTIONS

MapReduce

1. Map-Schritt:

   Worker Knoten wenden `map( )`-Funktion auf lokal vorhandene Daten an und schreiben das Ergebnis (Key-Value-Paare) in Speicher

2. Shuffle-Schritt:

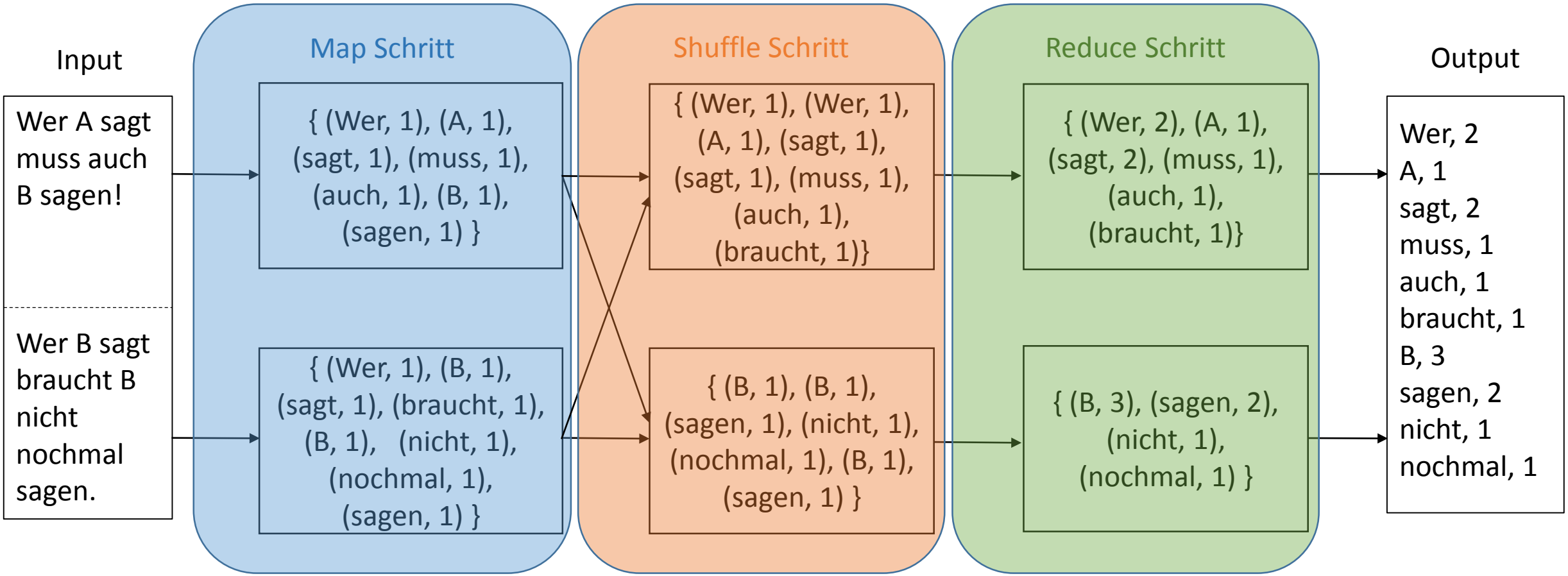
   Worker verteilen die Ergebnisse so, dass Paare mit gleichem Key bei gleichem Worker Knoten liegen

3. Reduce-Schritt:

   Worker Knoten verarbeiten die Ergebnisse des map-Schrittes durch Anwendung der `reduce( )`-Funktion

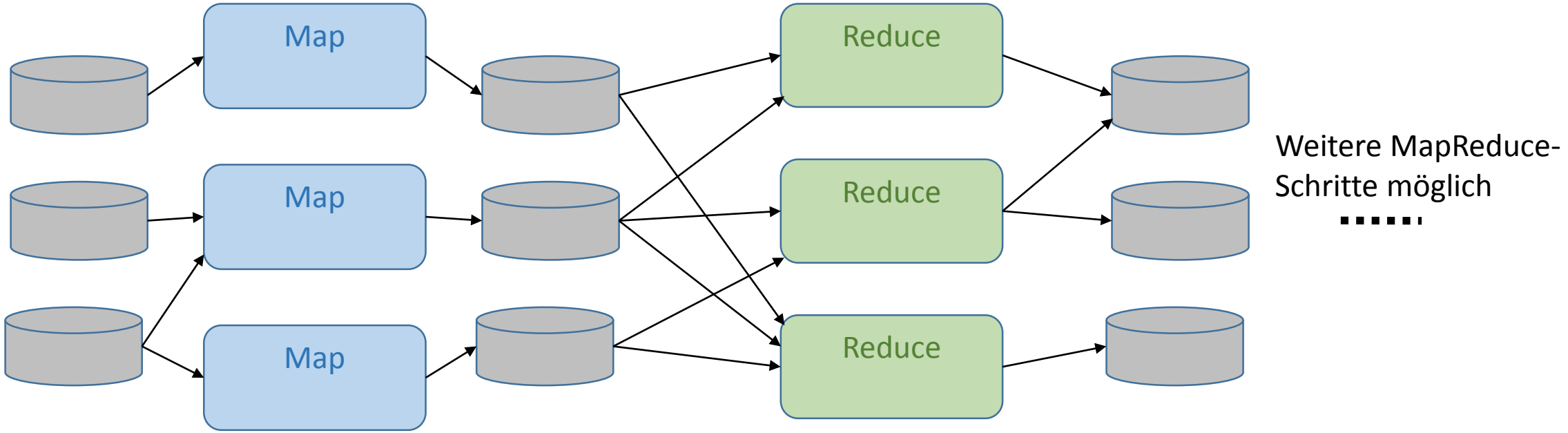
# BIG DATA SOLUTIONS

## MapReduce – Beispiel WordCount



# BIG DATA SOLUTIONS

Verteiltes MapReduce




Problem: Speicherzugriffe sind teuer!

# BIG DATA SOLUTIONS

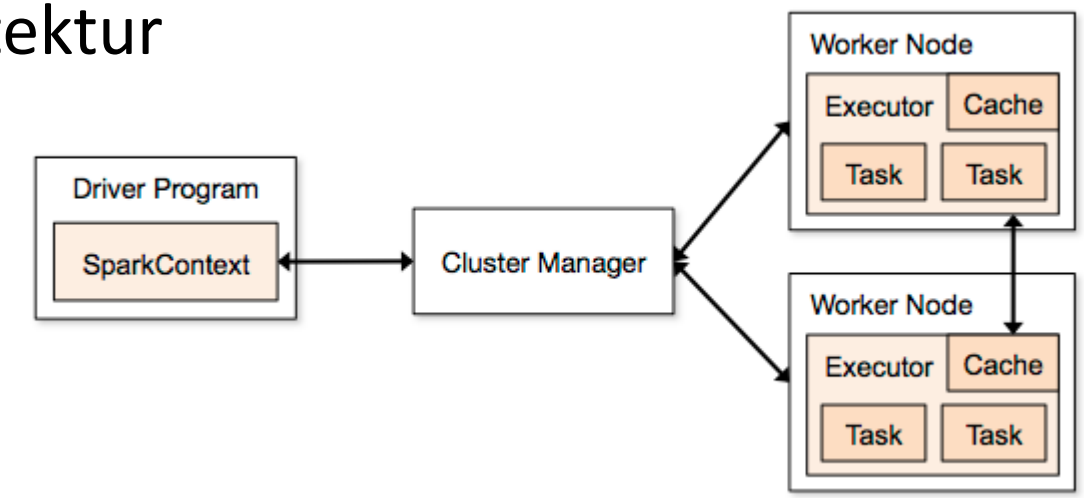
Lösung: Apache Spark


- Vorteile:
  - Daten können im Hauptspeicher gehalten werden
  - Weitere Operationen, zum Beispiel Joins
  - Neben Batch Processing ist auch Stream Processing möglich
  - Unterstützt neben Java auch Python, Scala, R

# BIG DATA SOLUTIONS

Apache Spark

- Komponenten
  - Hauptprogramm (Driver) definiert SparkContext
  - SparkContext stellt Verbindung zum Clustermanager her
  - Executors erhalten Code (jars) über SparkContext
- Architektur



# BIG DATA SOLUTIONS

## Apache Spark

- Schritt 1: Einen SparkContext erzeugen

```
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.SparkConf

SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
/*
 * master = local[n] für local mode mit Benutzung von n CPUs oder
 * master = cluster URL für Verbindung zu Spark Cluster
 */
JavaSparkContext sc = new JavaSparkContext(conf);
```

# BIG DATA SOLUTIONS

Apache Spark

- Resilient Distributed Datasets (RDDs) als Hauptabstraktion
  - über Cluster verteilt
  - Fehlertolerant

- RDDs können auf zwei Wege erzeugt werden:
  - Durch das Aufrufen der `parallelize()` Funktion

```
List<Integer> data = Arrays.asList(1, 2, 3);
JavaRDD<Integer> rddData = sc.parallelize(data);
```

  - Durch Auslesen aus externen Quellen

```
JavaRDD<String> rddFile = sc.textFile("data.txt");
```

# BIG DATA SOLUTIONS

Apache Spark

- Transformations: erzeugen neue RDDs aus Bestehenden (lazy)
  - `map(func)`
  - `filter(func)`
  - `union(otherDataset)`
  - …

```
JavaRDD<Integer> data = sc.parallelize(Arrays.asList(1, 2, 3));
JavaRDD<Integer> odd = data.map(new Function<Integer, Integer>() {
      public Integer call(Integer i) { if (i % 2 != 0) return i;
                                       return 0;}
});
```

# BIG DATA SOLUTIONS

Apache Spark

- Actions: berechnen einen Wert und geben ihn an das Hauptprogramm zurück
  - `reduce(func)`
  - `count()`
  - `take(n)`
  - …

```
int sum = odd.reduce(new Function2<Integer, Integer, Integer>() {
        public Integer call(Integer i, Integer j) { return i+j; }
});
```

# BIG DATA SOLUTIONS

Apache Spark

- Persistenz: Spark bietet verschiedene *Storage Levels* an
  - `MEMORY_ONLY`: Speichert deserialisierte Java Objekte im Hauptspeicher (default). Wenn Hauptspeicher voll, werden nicht gecachete Partitionen on-the-fly neu berechnet.
  - `MEMORY_AND_DISK`: Speichert Objekte zusätzlich auf Platte. Wenn Hauptspeicher voll, werden nicht gecachete Partitionen von der Platte gelesen.
  - `DISK_ONLY`: Speichert Objekte nur auf Platte
  - `MEMORY_ONLY_SER`: Speichert serialisierte Java Objekte im Hauptspeicher. Platzeffizienter, aber teurer zum Lesen (Deserialisierung)
  - …

# BIG DATA SOLUTIONS

Apache Spark

- Variablen werden für Spark Operationen auf Worker Knoten kopiert und Updates dieser Variablen nicht an den Driver zurückgesendet
- Daher: Zwei Arten von *Shared Variables*:
  - Broadcast Variables:
    - Read-only Variable
    - Wird in jedem Worker gecached anstatt Kopien zu versenden
  - Accumulators:
    - Add-only Variable
    - Kann nur vom Driver gelesen werden

# BIG DATA SOLUTIONS

Spark Streaming

= Streaming Framework, das auf Spark Core API aufbaut

- Daten können von mehreren Datenquellen aufgenommen werden
  - Twitter
  - HDFS
  - …


- Stream processing kann mit weiteren Spark Libraries kombiniert werden (z.B. Spark‘s Machine Learning Algorithmen)

# BIG DATA SOLUTIONS

Spark Streaming

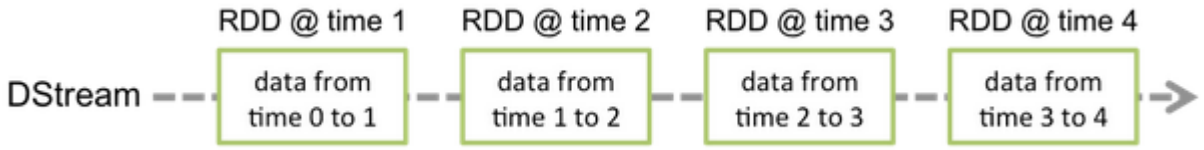
- Spark‘s Streaming Workflow:


  - Input Datenstrom wird aufgenommen
  - Stream wird in Microbatches unterteilt (Sequenz von RDDs)
  - Microbatches werden verarbeitet
  - Als Ergebnis kommt wieder ein Stream von Microbatches raus

# BIG DATA SOLUTIONS

Spark Streaming

- DStream als Basis-Datenstruktur




- StreamingContext als Startpunkt

```
SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);
JavaStreamingContext ssc = new JavaStreamingContext(conf, Duration(1000));
```

# BIG DATA SOLUTIONS

## Spark Streaming

```
JavaDStream<String> lines = ssc.fileStream("hdfs://...");
// liest alle Dateien aus dem angegebenen Ordner im HDFS
JavaDStream<String> string_numbers = lines.flatMap(
        new FlatMapFunction<String,String>() {
                public Iterable<String> call(String s) {
                        return Arrays.asList(s.split(";"));
                }
        });

JavaDStream<Integer> odd_numbers = string_numbers.filter(
        new Function<String,Boolean>() {
                public Boolean call(String s) {
                        if (Integer.parseInt(s) % 2 != 0) return true;
                        return false;
                }
        });
```

# BIG DATA SOLUTIONS

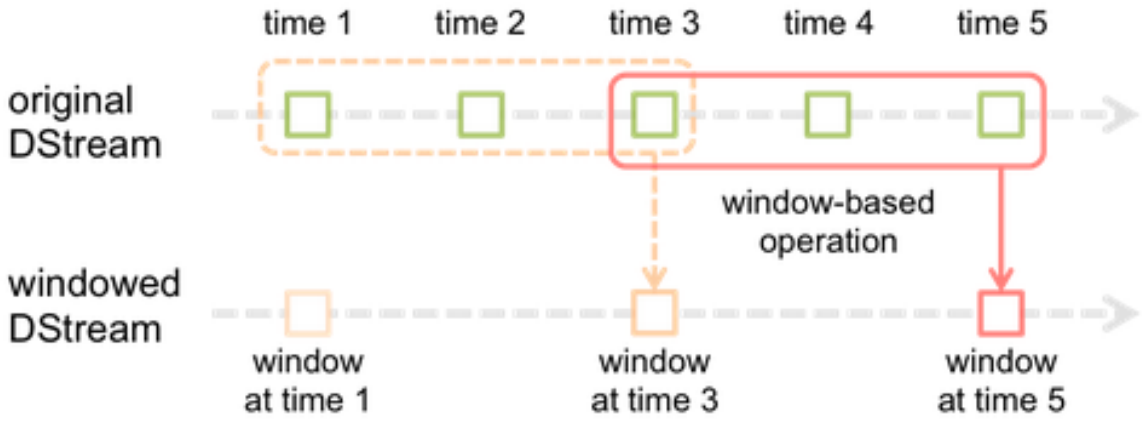
## Spark Streaming

```
JavaDStream<Integer> sums = odd_numbers.reduce(
        new Function2<Integer,Integer,Integer>() {
                public Integer call(Integer i, Integer j) { return i+j; }
        });

sums.print();

ssc.start()
ssc.awaitTermination();
```

# BIG DATA SOLUTIONS

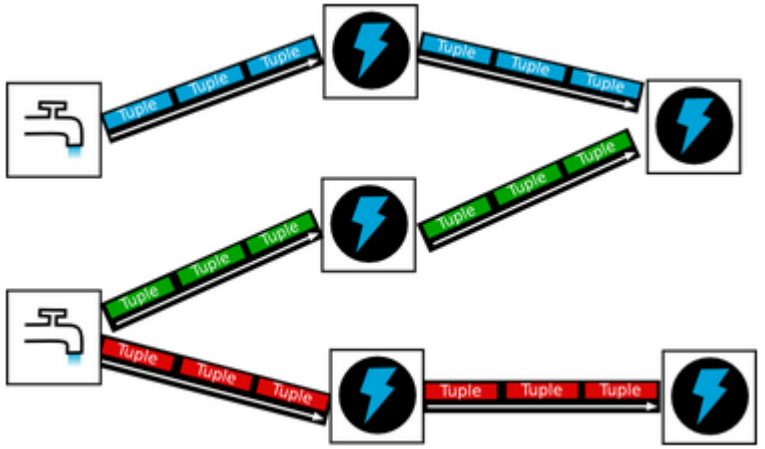
Spark Streaming

- Window Operationen
- 2 Basisparameter:
  - windowLength (`w`)
  - slideInterval (`s`)
- Transformationen:
  - `window(w,s)`
  - `reduceByWindow(func,w,s)`
  - `...`



# BIG DATA SOLUTIONS

## Apache Storm

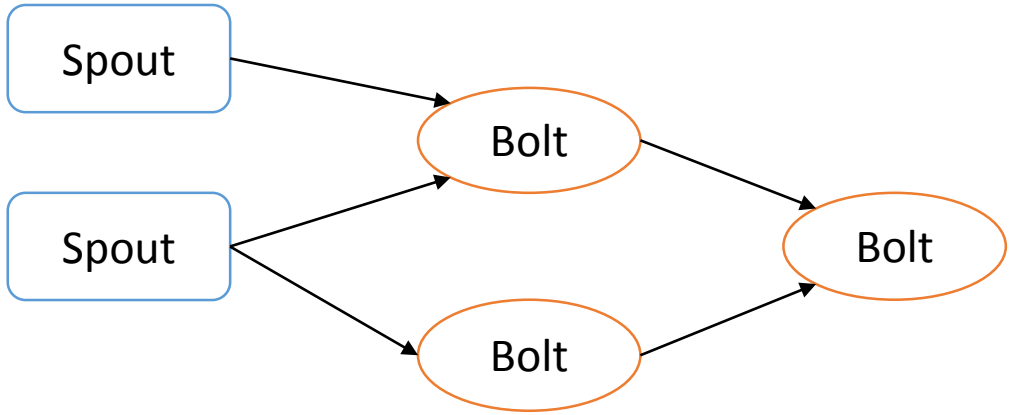



- Alternative zu Spark
- Real-time processing
- Topologie-basiert (Topologie = Berechnungsgraph)

# BIG DATA SOLUTIONS

## Apache Storm

- Komponenten
  - Master: verteilt Logik und weist Tasks zu
  - Worker: starten und stoppen Prozesse

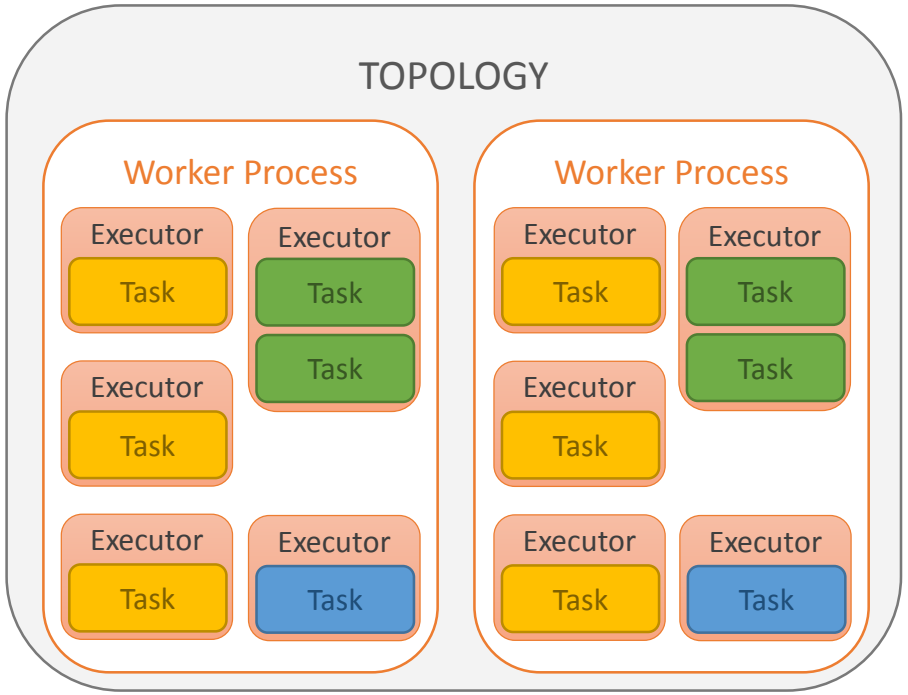



- Topologie
  - Spout: Datenquelle, versendet Daten an verlinkte Bolts
  - Bolt: empfängt Daten von potentiell mehreren Quellen, verarbeitet sie und schickt Ergebnisse weiter (evtl. wieder als Stream)

# BIG DATA SOLUTIONS

## Apache Storm


```
Config conf = new Config();
conf.setNumWorkers(2); // use two worker processes

topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2); // set parallelism hint to 2

topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
               .setNumTasks(4)
               .shuffleGrouping("blue-spout");

topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
               .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology(
        "mytopology",
        conf,
        topologyBuilder.createTopology()
    );
```



# BIG DATA SOLUTIONS

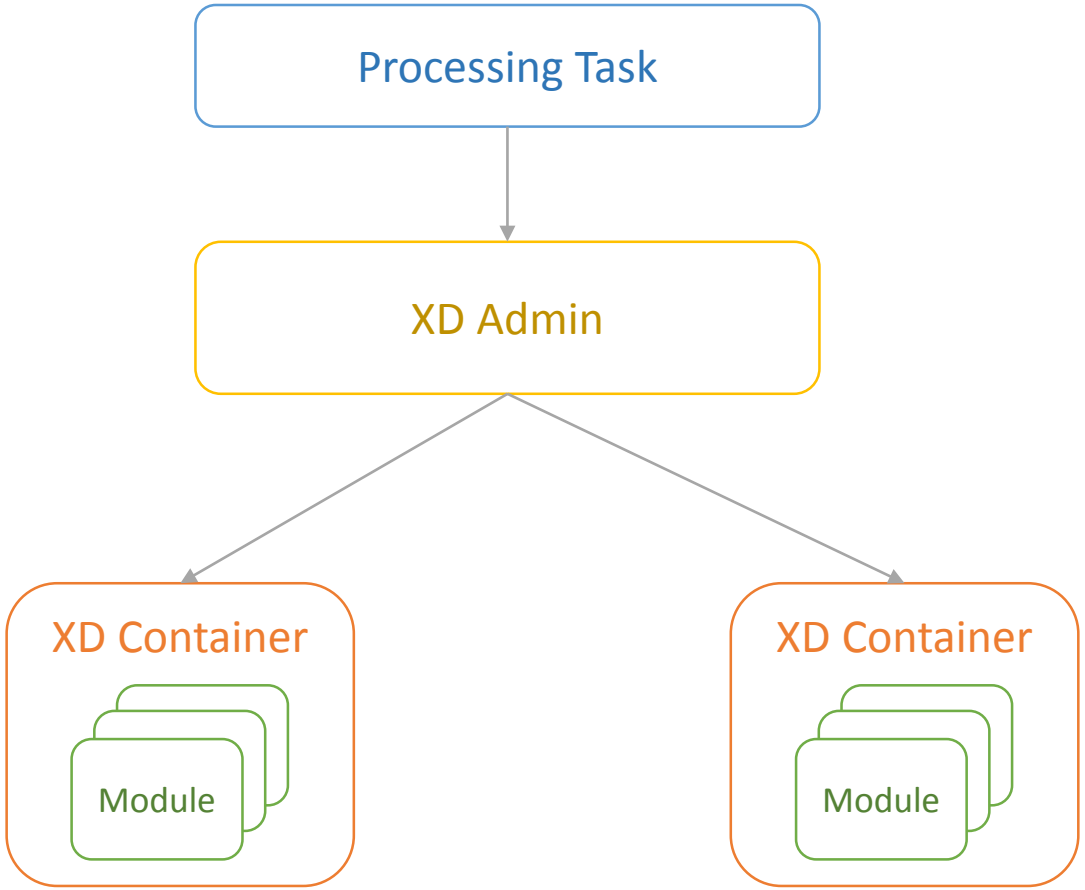
Spring XD 

- Framework zur Modellierung von Processing Workflows
- Modularer Aufbau
- Batch & Stream Processing

# BIG DATA SOLUTIONS

## Spring XD

- Architektur
  - Processing Tasks werden an XD Admin geschickt
  - XD Admin zerlegt Tasks in Module
  - Clustermanager (ZooKeeper) verteilt Module an XD Container
  - XD Container arbeitet Module ab



# BIG DATA SOLUTIONS

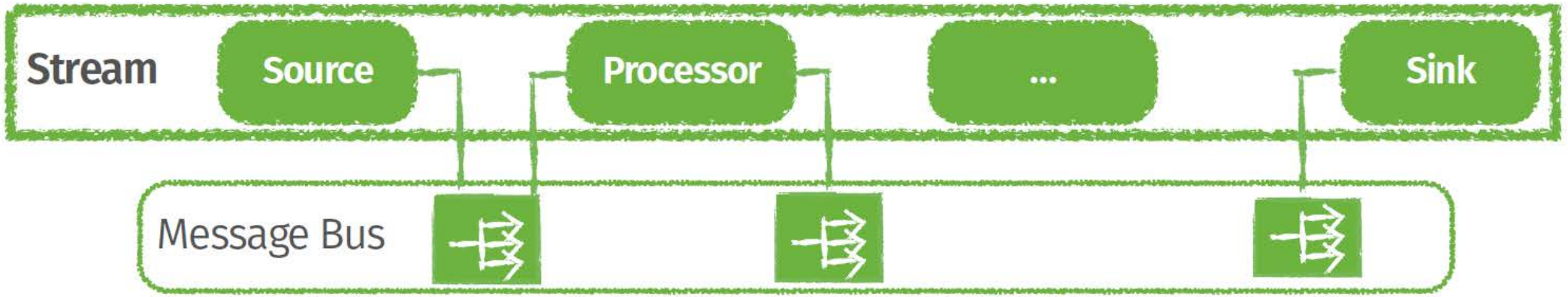
## Spring XD

- Stream Processing:
  - Streams werden mit der *Domain Specific Language* (DSL) deklariert
  - Stream wird als Verknüpfung von Modulen definiert (| als Verknüpfungs-operator)
  - Output eines Moduls ist wieder Datenstrom und wird in Message Bus eingespeist
  - Form: Quelle | <optional> Verarbeitungsmodul | Senke
  - Beispiel: http | filter | transform | file

# BIG DATA SOLUTIONS

## Spring XD

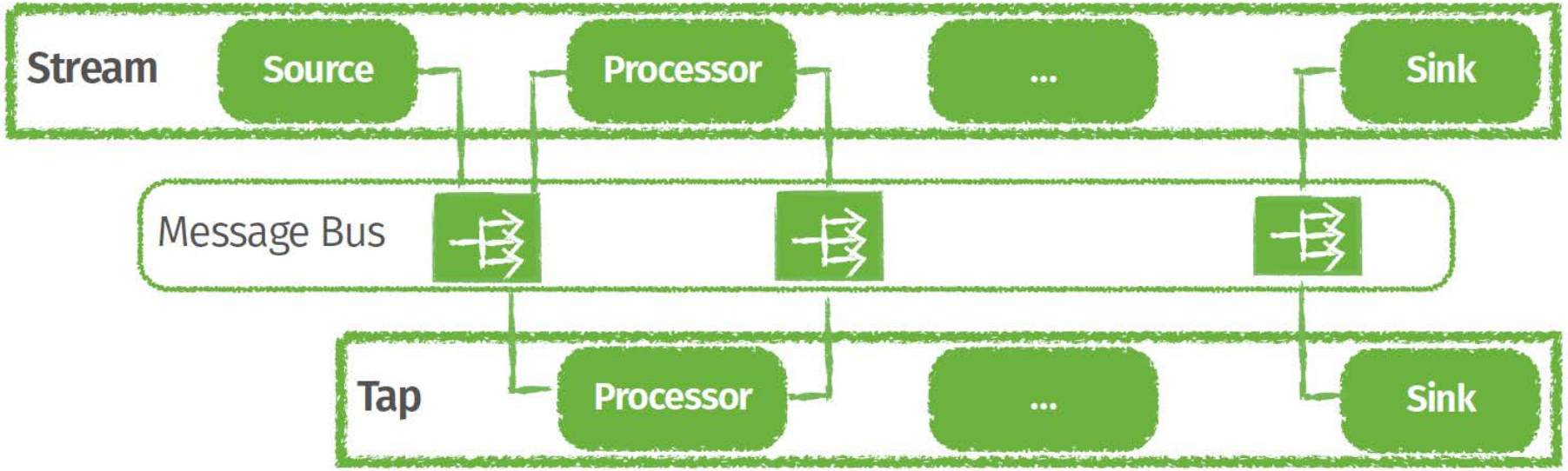
- Stream Processing:



# BIG DATA SOLUTIONS

## Spring XD

- Taps:
  - Streams können an beliebiger Stelle der Verarbeitungspipeline „abgehört“ werden
  - Originalstream bleibt unverändert



# BIG DATA SOLUTIONS

- Weitere Informationen:
  - https://spark.apache.org/
  - https://storm.apache.org/
  - http://projects.spring.io/spring-xd/

- Ab Wintersemester 2015/16:

  Seminar „Big Data Tools“