Big-Data-Technologien

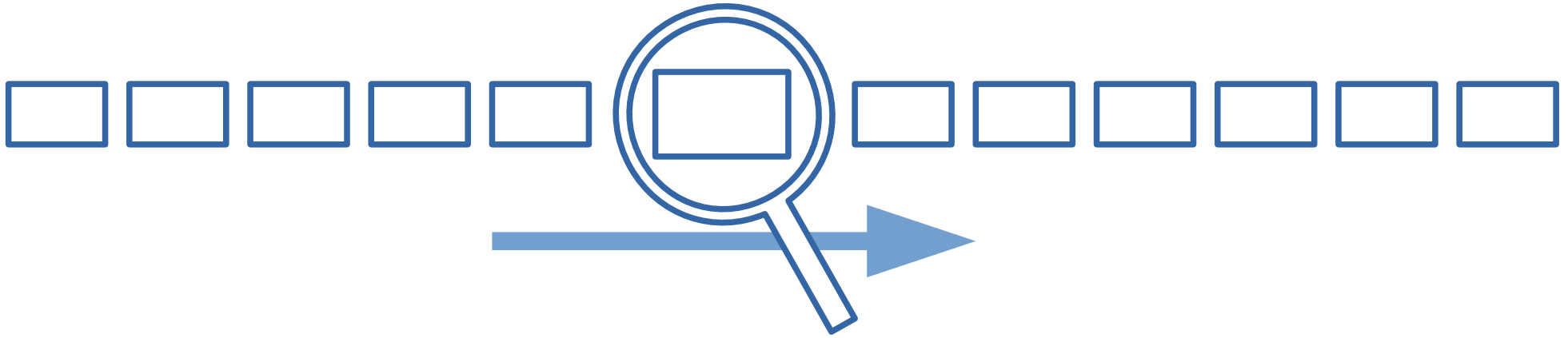
Kapitel 15: Stream-Verarbeitung

Hochschule Trier
Prof. Dr. Christoph Schmitz

Stream

- **kontinuierlicher** Fluss von Datensätzen
- **gleichartige** Datensätze
- **unendlich** (bzw. Ende nicht absehbar)

- Zugriff nur **sequentiell**
- Zugriff nur **ein Mal** pro Datensatz



"Messaging" vs. "Streaming" vs. …

- **Messaging**
  - Transport von Daten von A nach B
  - Architekturprinzip für verteilte Systeme
  - Garantien, Transaktionen, Skalierbarkeit, …
- **Streaming**
  - Verarbeitungsmodell für Datenströme
  - Grundlage für Algorithmen
- **Echtzeit**
  - hier: "weiche" Echtzeit

Operationen

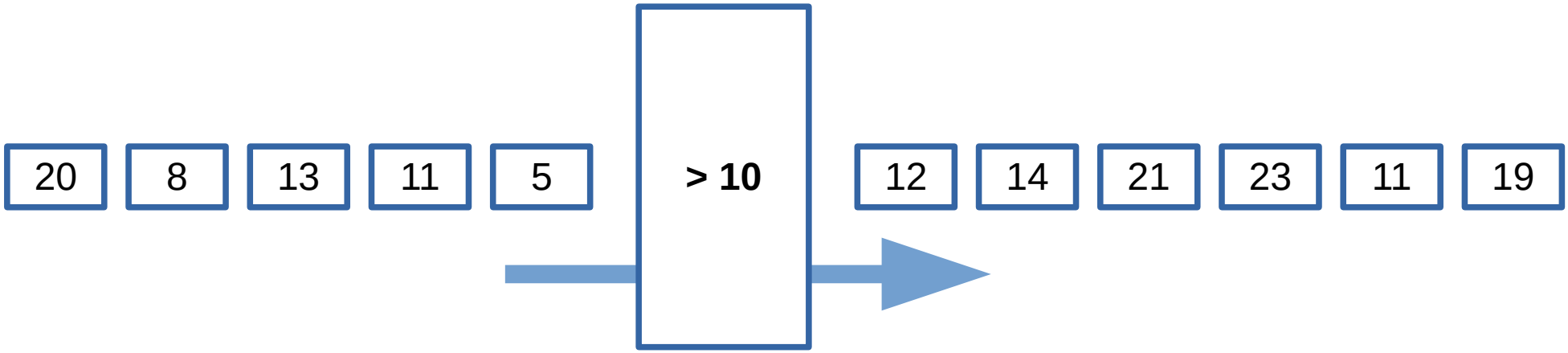
- **Filtern**
- **Transformieren**

- **Aggregation**
- **Join**

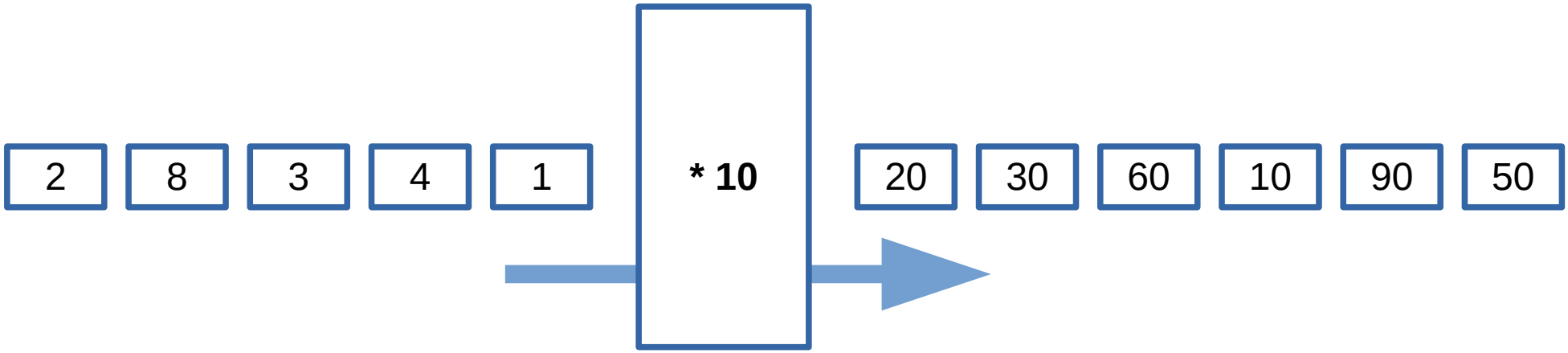
  → Worüber?

- **Complex Event Processing**

Operationen: Filtern



Operationen: Transformieren



Filtern und Transformieren

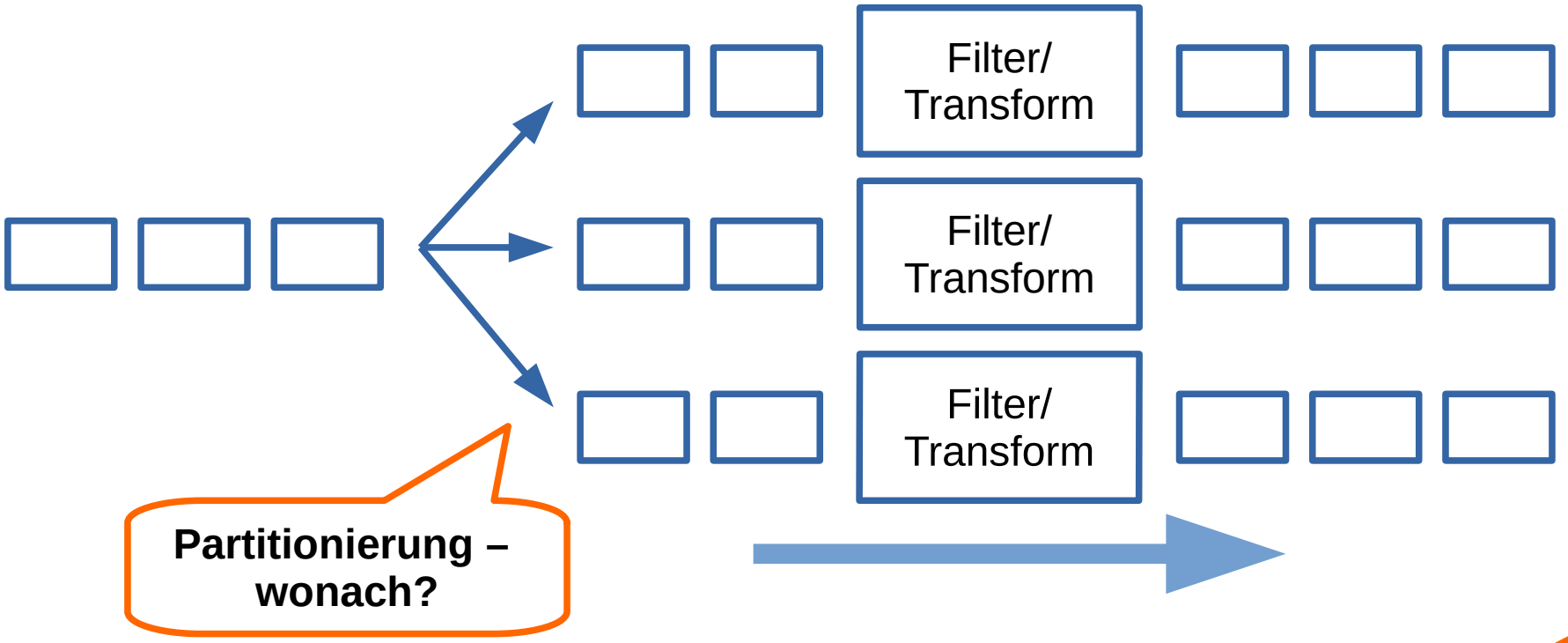
- Filtern und Transformieren sind einfach!

```
for data in inputstream:
    if filter(data):
        outputstream.write(transform(data))
```

- Keine **Korrelation** zwischen Elementen
- Trivial zu **parallelisieren**

→ vgl. Mapper (MapReduce),
schmale Transformationen (Spark)

Parallelisierung: Filter, Transform



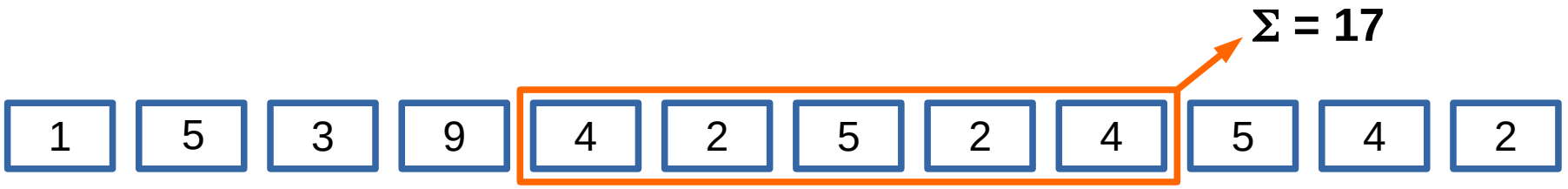
Trivial parallelisierbar

Operationen: Aggregation

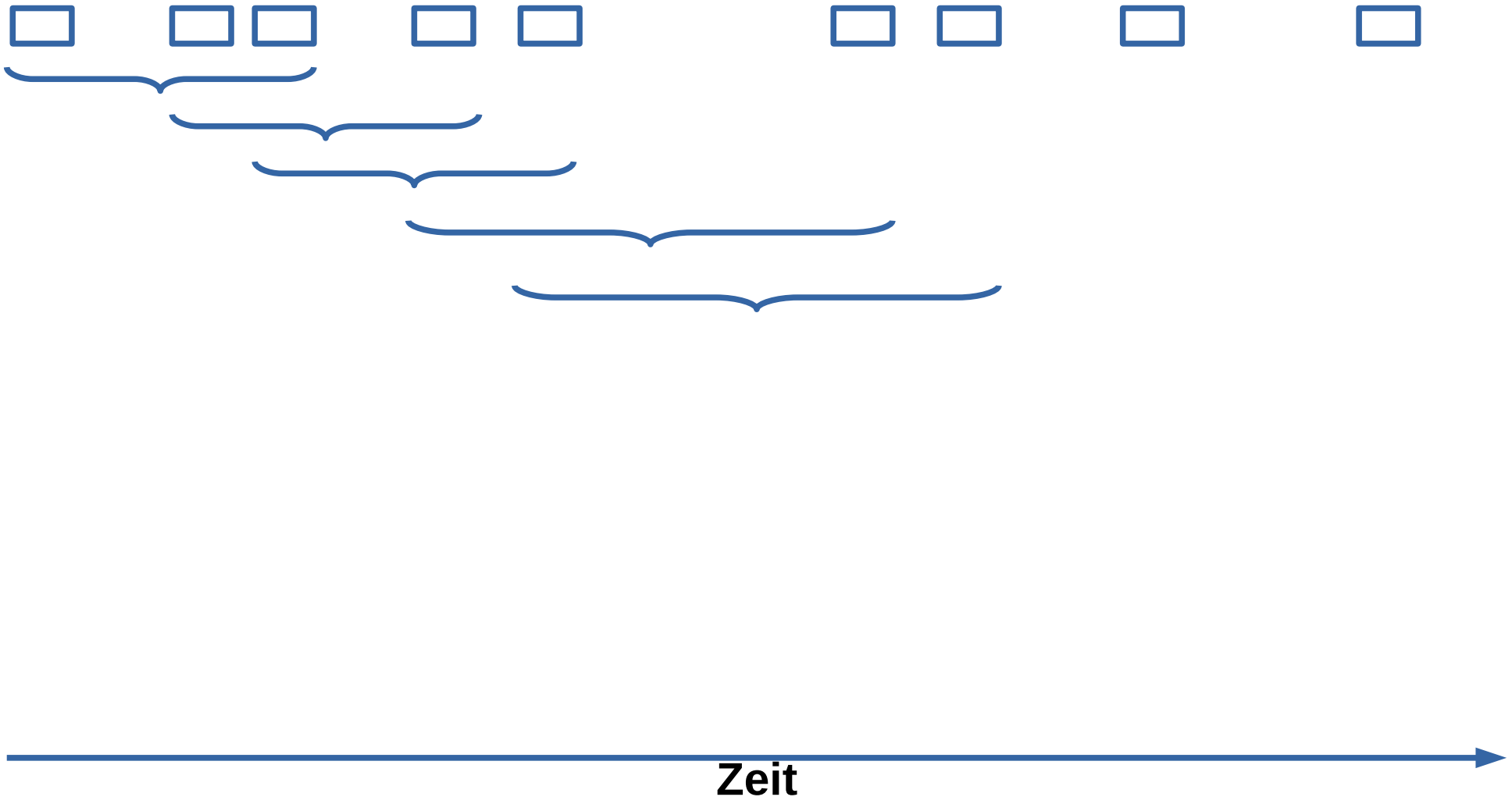
- **Worüber** wird aggregiert?
  - Summe eines unendlichen Stroms?
  - Maximum eines unendlichen Stroms?

Operationen: Aggregation

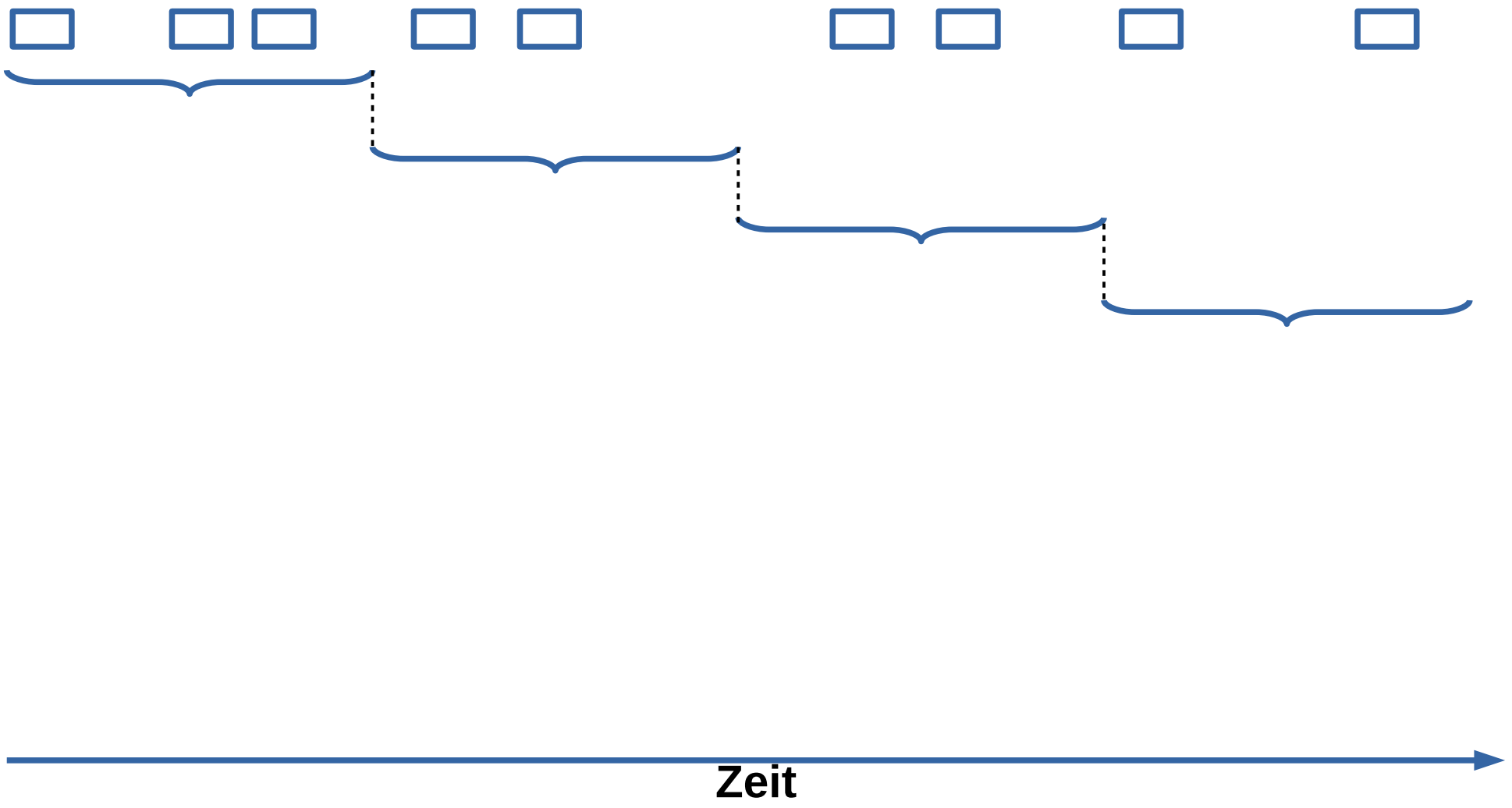
- **Worüber** wird aggregiert?
  - Summe eines unendlichen Stroms?
  - Maximum eines unendlichen Stroms?
- Einschränkung auf **Fenster**
  - Bestimmte **Anzahl** von Elementen
  - Elemente innerhalb eines **bestimmten Zeitraumes**
- Was ist das **Ergebnis**?
  - → Strom von Ergebnissen



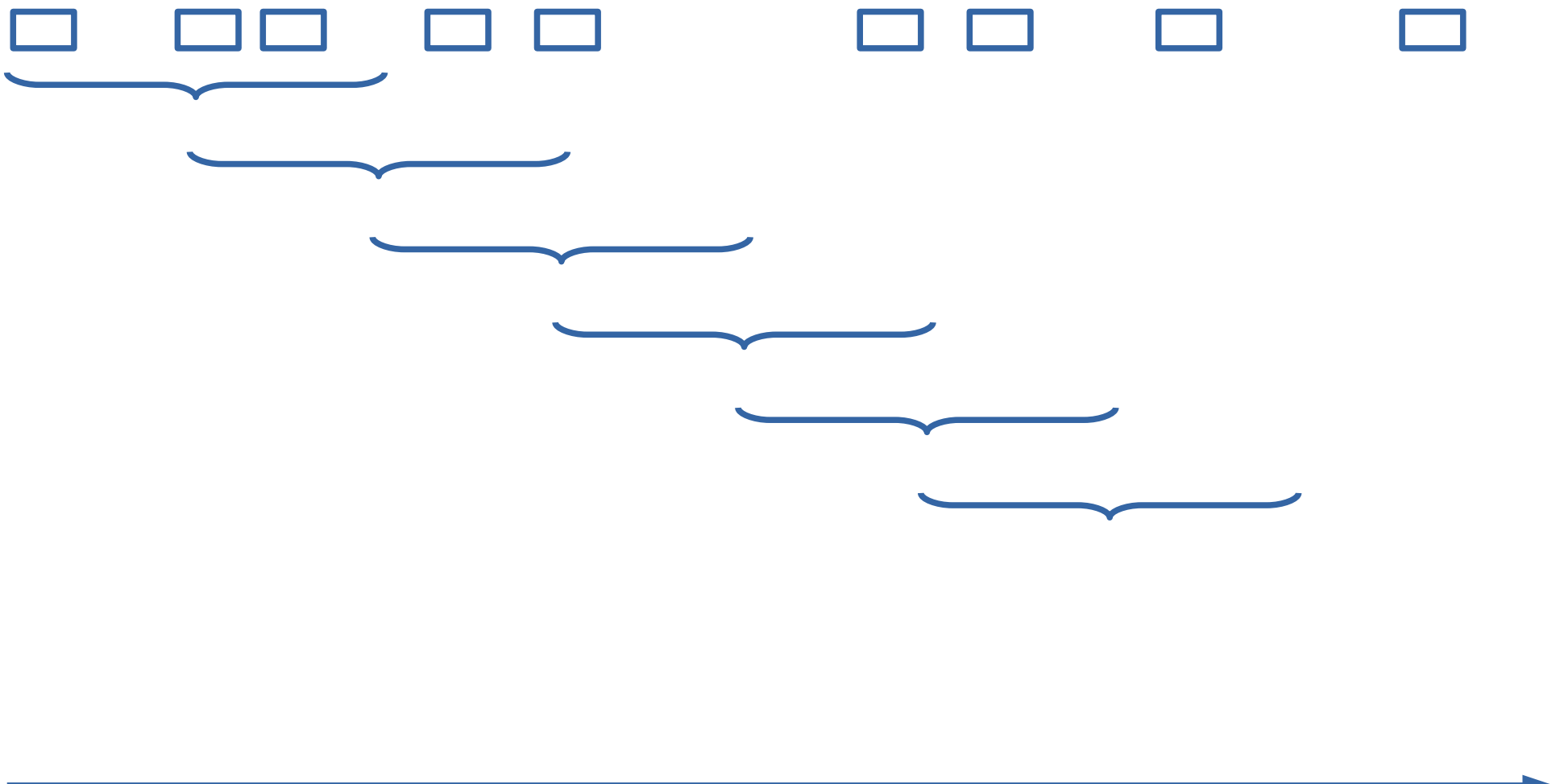
Fenster: letzte n Elemente



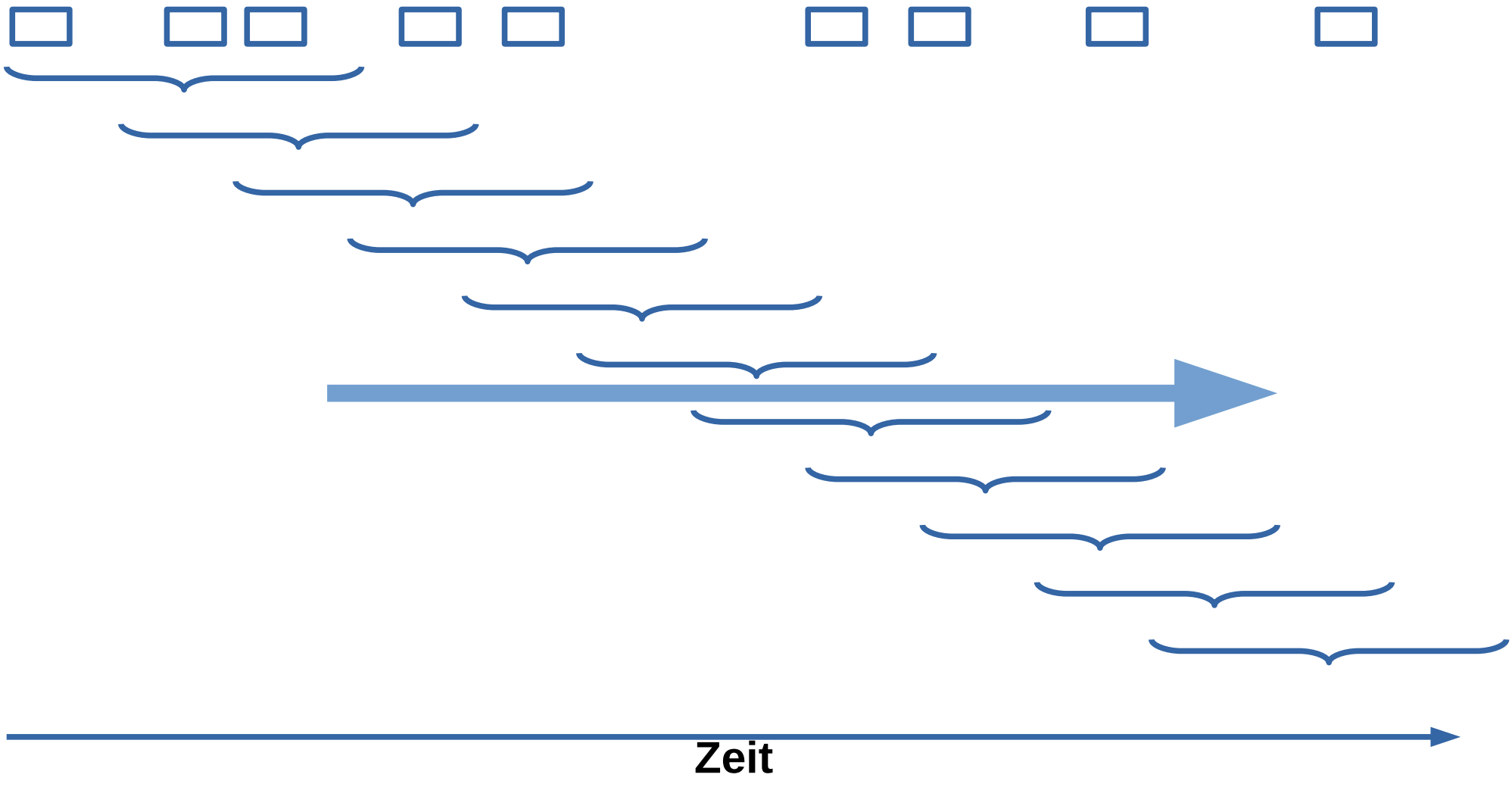
Zeitfenster: Hopping Windows



Zeitfenster: Tumbling Windows



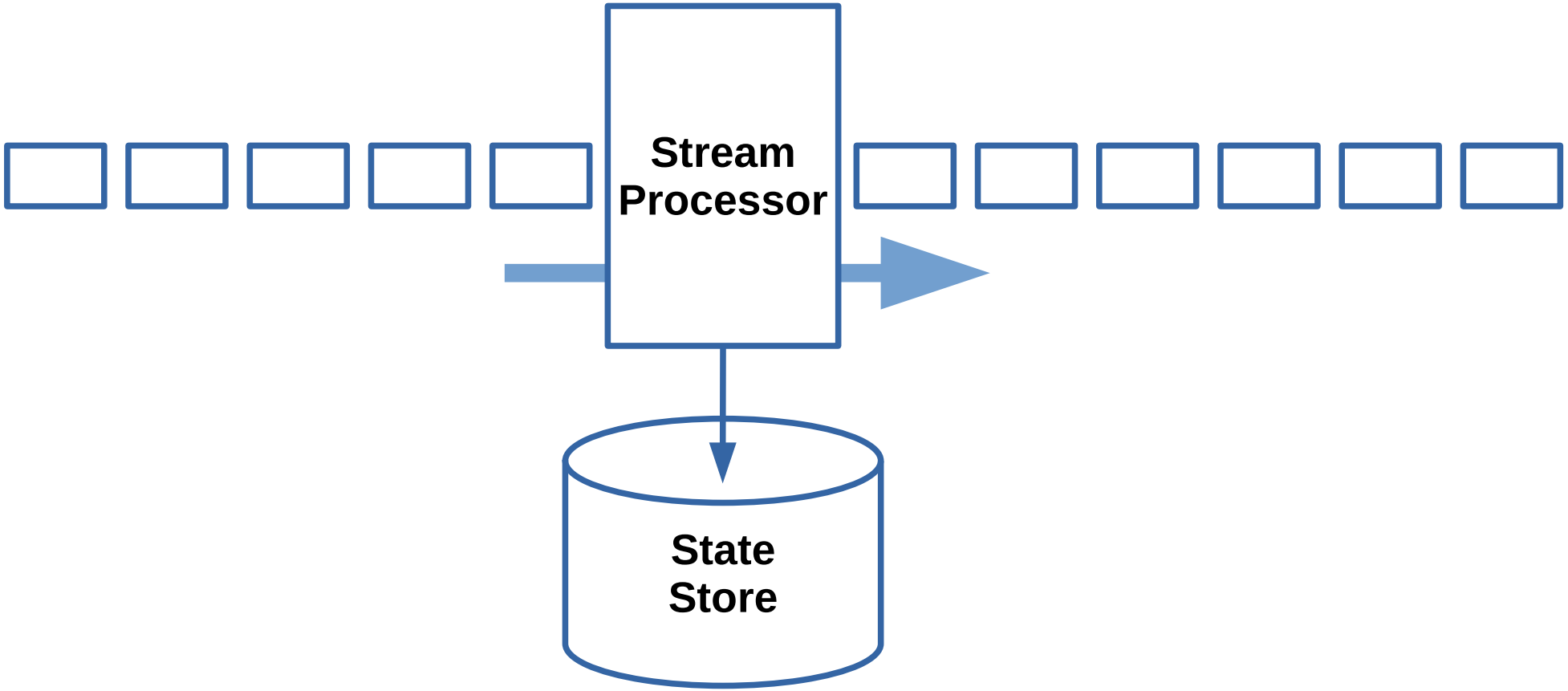
Zeitfenster: Sliding Windows



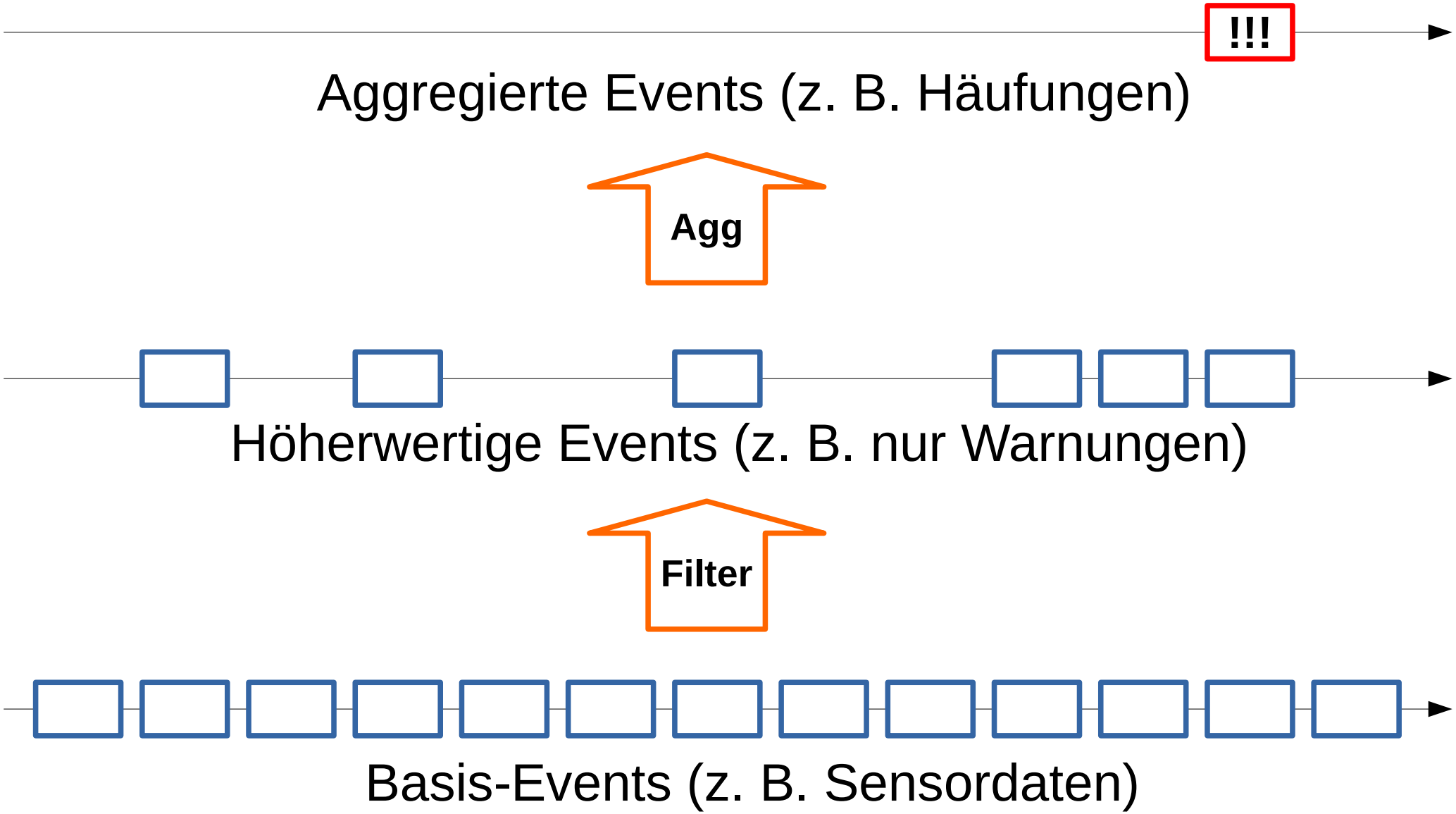
Aggregatfunktionen: Wo lebt der Zustand?

```
window = []
for data in inputstream:
   # move window
    window.add(data)
    window.removeOldData()

    if newWindow():
       # emit aggregate
       outputstream.write(aggregate(window))
```

Behandeln von Zustand



Complex Event Processing



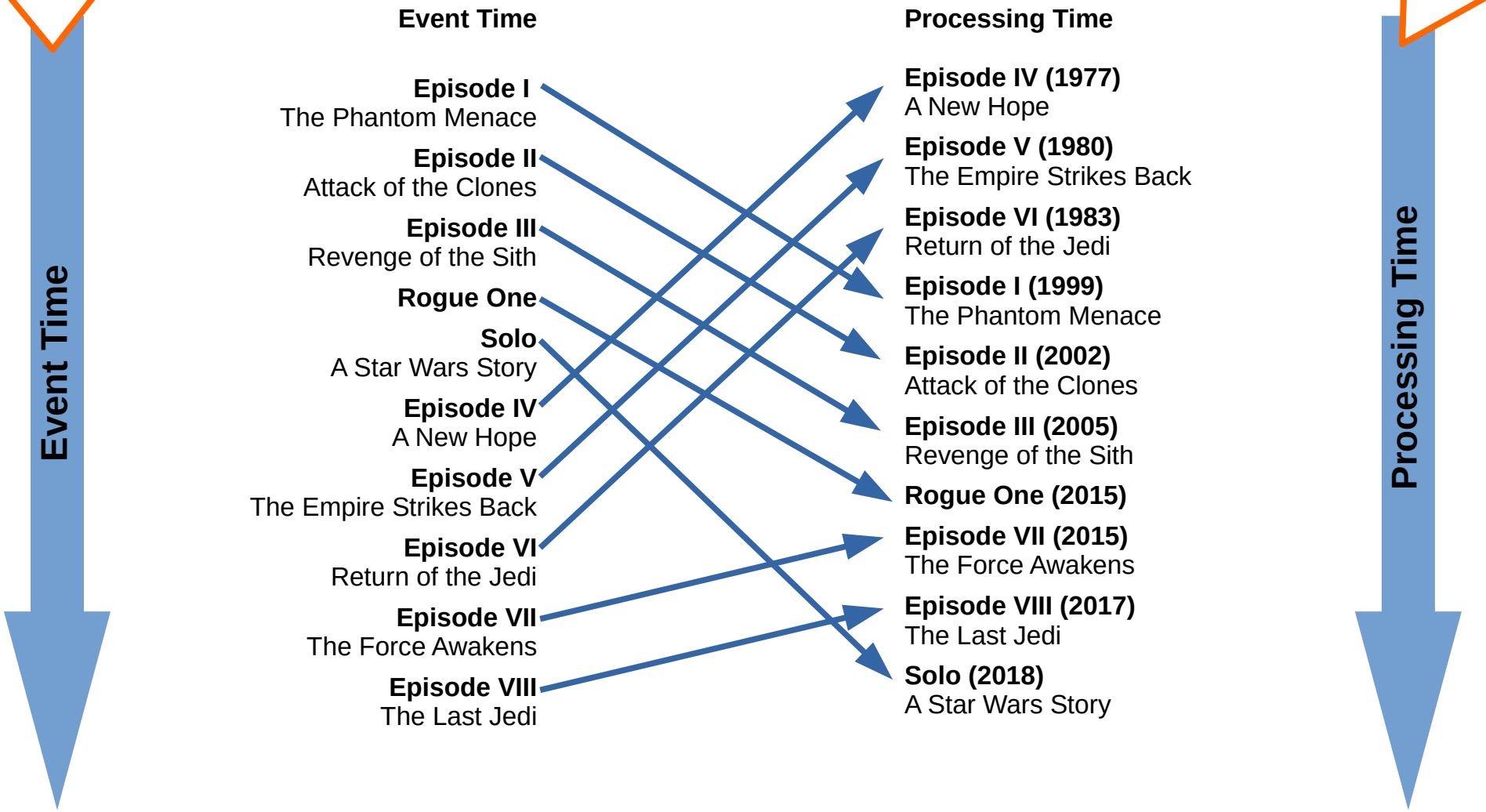
Zeitbegriffe

- Was ist der **Zeitstempel** eines Ereignisses?
- **Event Time:**
  Wann hat das Ereignis stattgefunden?
- **Ingestion Time:**
  Wann wurde das Ereignis in unser System eingefügt?
- **Processing Time:**
  Wann wurde das Ereignis verarbeitet?

Beispiel: Star Wars

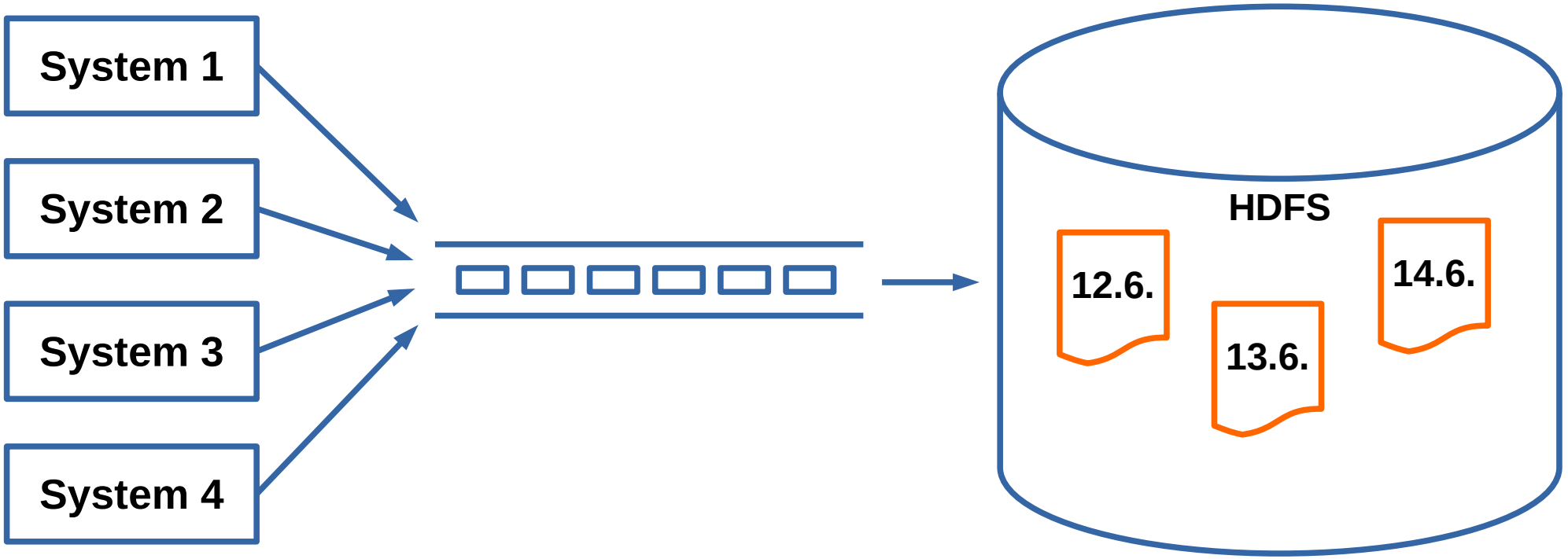
"A long time ago in a galaxy far, far away…"

Kinostart in unserer Zeitrechnung



Beispiel: Log-Daten

- "Aggregiere die Log-Daten pro Tag!"



Zeitbegriffe: Strategien

- Wie mit **verspäteten ("out-of-order") Daten** umgehen?

- Strategie 1: **Warten, dann verwerfen**
  - Warte bis 3:00 Uhr am Folgetag.
    Was danach kommt, wird verworfen.

- Strategie 2: **Ergebnisse aktualisieren**
  - Wenn am 14.6. noch Ereignisse vom 12.6. bekannt werden, publiziere neue Aggregate für den 12.6.

Zwischenfazit

- Kontinuierliche **Datenströme**
- **Operationen** wie Batch-Verarbeitung, aber…

- … andere **Semantik von Aggregationen**:
  → **Fenster**

- **Complex Event Processing**
  als Königsdisziplin

Zwischenfazit

- Unterschiedliche Zeitbegriffe
  - **Event Time** ist das Ideal
  - **Processing Time** ist umsetzbar

Herausforderungen

- Die Klassiker…
  - **Durchsatz**
  - **Latenz**
  - Skalierbarkeit
- Zustandsbehaftete Operationen
  - Wo lebt der **Zustand**?
  - **Skalierbarkeit**
  - Beziehungen zwischen Ereignissen → **Zeitbegriffe**
- **Complex** Event Processing

Stream-Verarbeitung: Systeme

Spezialisten

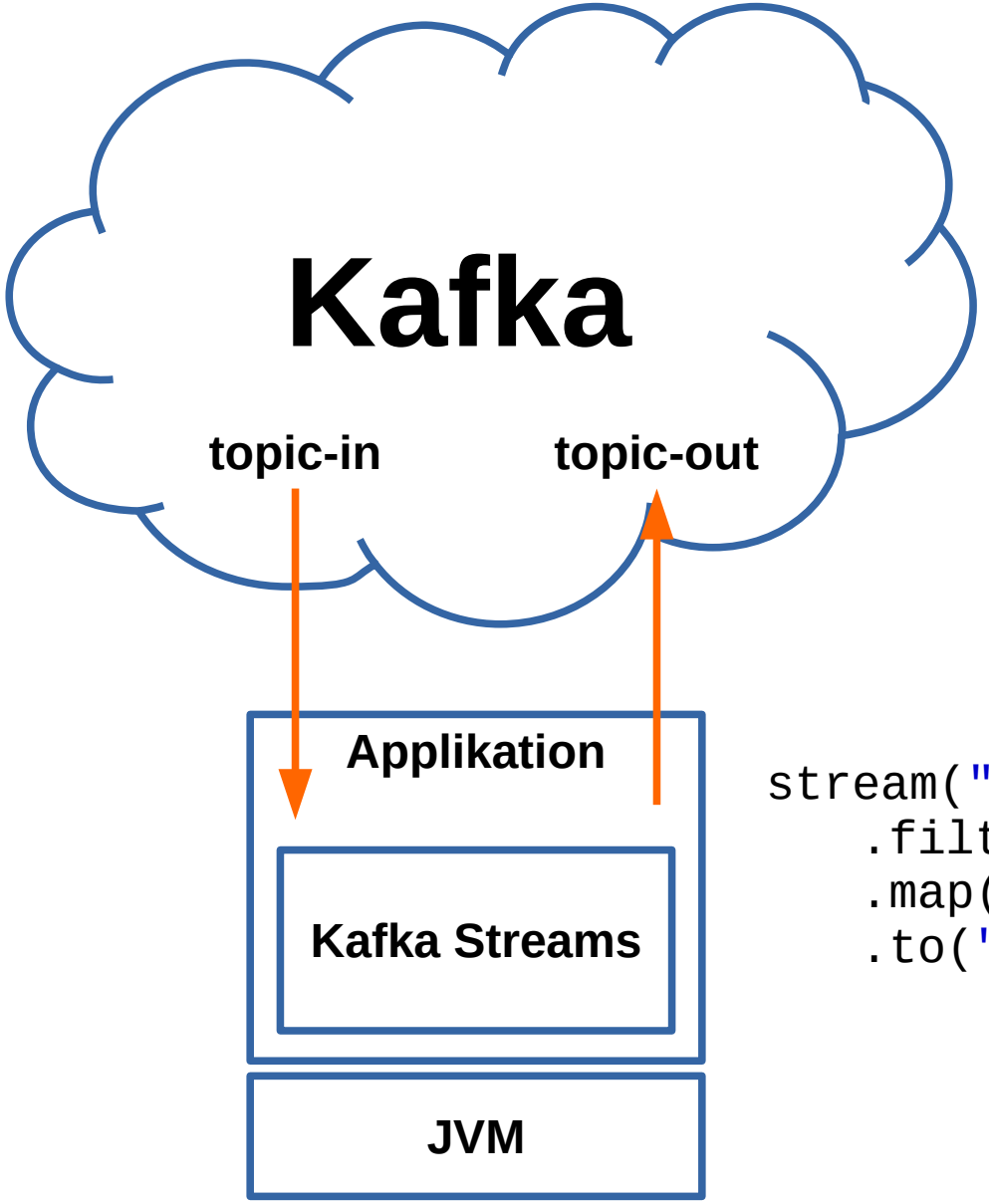
Micro Batches

Message Broker

Kafka Streams

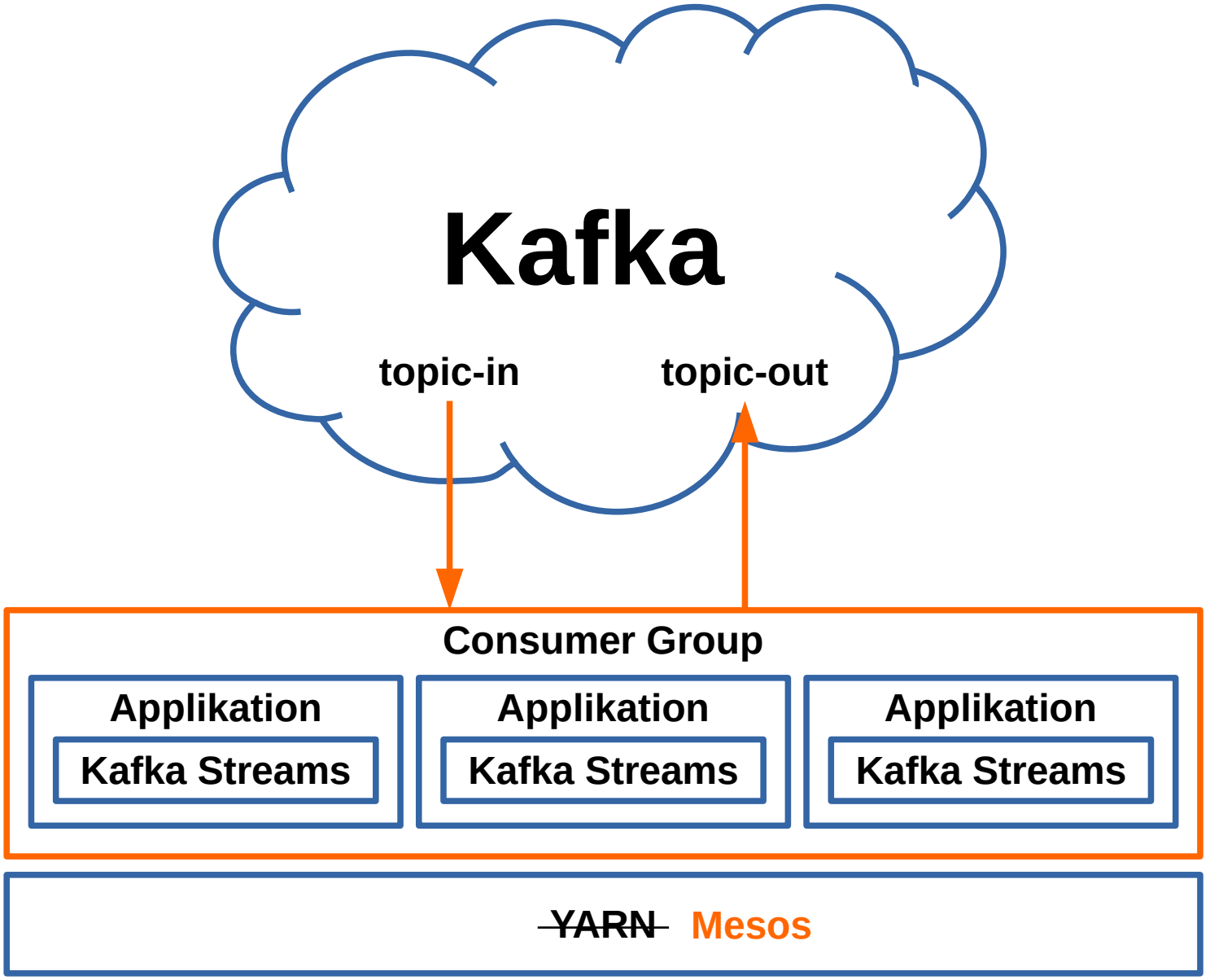
- **Kafka** transportiert nur Daten
  - keine Verarbeitung/Filterung/…
- **Kafka Streams** fügt Verarbeitung hinzu
- **Bibliothek** → keine Infrastruktur außer Kafka

Kafka Streams




```
stream("topic-in")
    .filter(…)
    .map(…)
    .to("topic-out")
```

Kafka Streams



Codebeispiel

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> lines = builder.stream(Serdes.String(),
                                    Serdes.String(), "inputTopic");

KStream<String, String> warnings =
       lines.filter((key, value) -> value.contains("WARNING"));

warnings.to("warningsTopic");

Properties config = new Properties();
// config befüllen...
KafkaStreams streams = new KafkaStreams(builder, config);
streams.start();

Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

Codebeispiel

```
KStreamBuilder builder = new KStreamBuilder();

builder.stream(Serdes.String(), Serdes.String(), "inputTopic")
       .filter((key, value) -> value.contains("WARNING"));
       .to("warningsTopic");

Properties config = new Properties();
// config befüllen...
KafkaStreams streams = new KafkaStreams(builder, config);
streams.start();

Runtime.getRuntime().addShutdownHook(new Thread(streams::close));
```

Streams und Tabellen

- Tabellen
  - beinhalten **Ist-Zustand**:

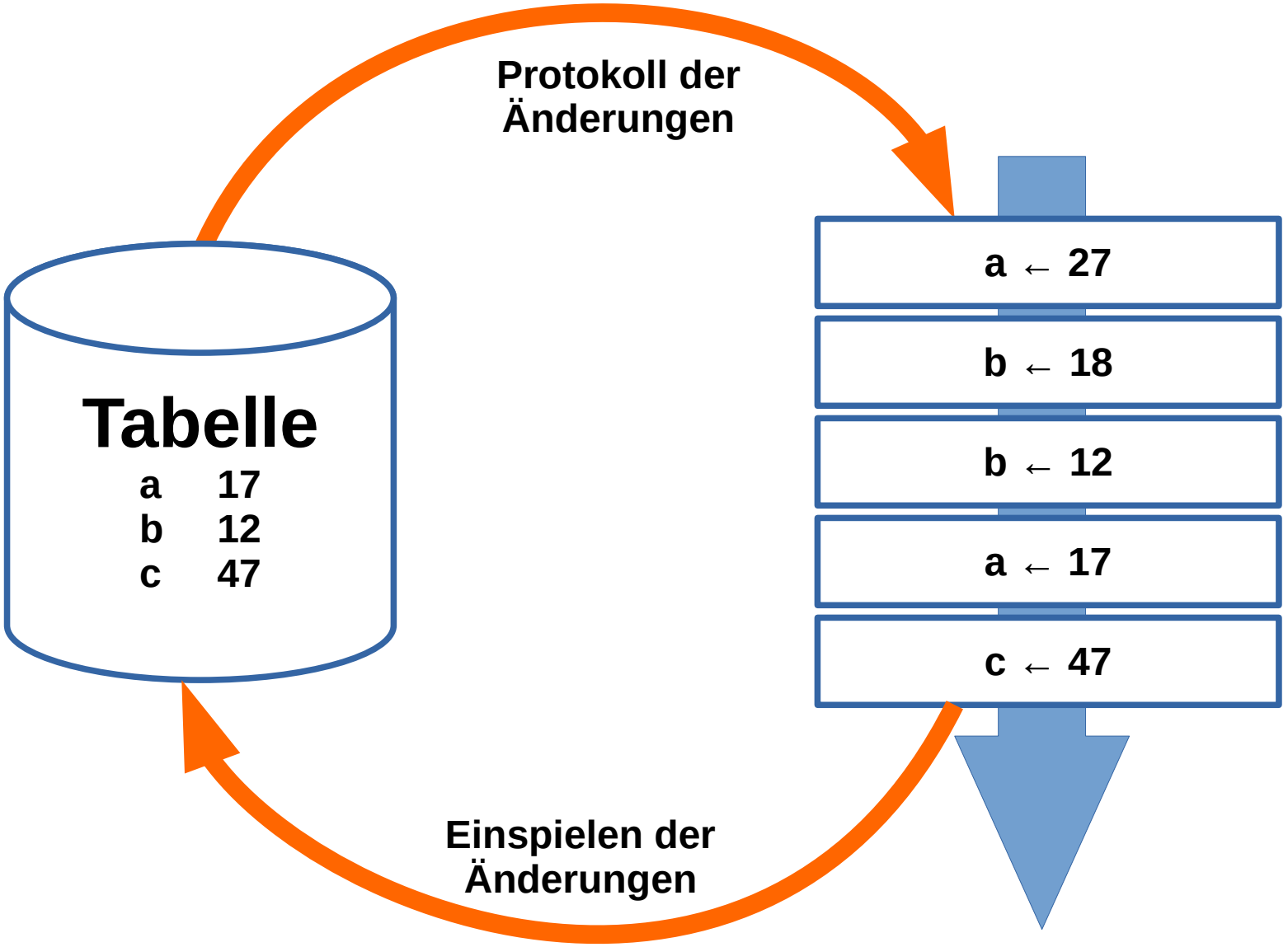
kontostand = 100

- Streams (Logs)
  - dokumentieren **Veränderungen**:

kontostand ← kontostand + 50

→ Neha Narkhede, Jay Kreps (confluent.io/LinkedIn)

Streams und Tabellen



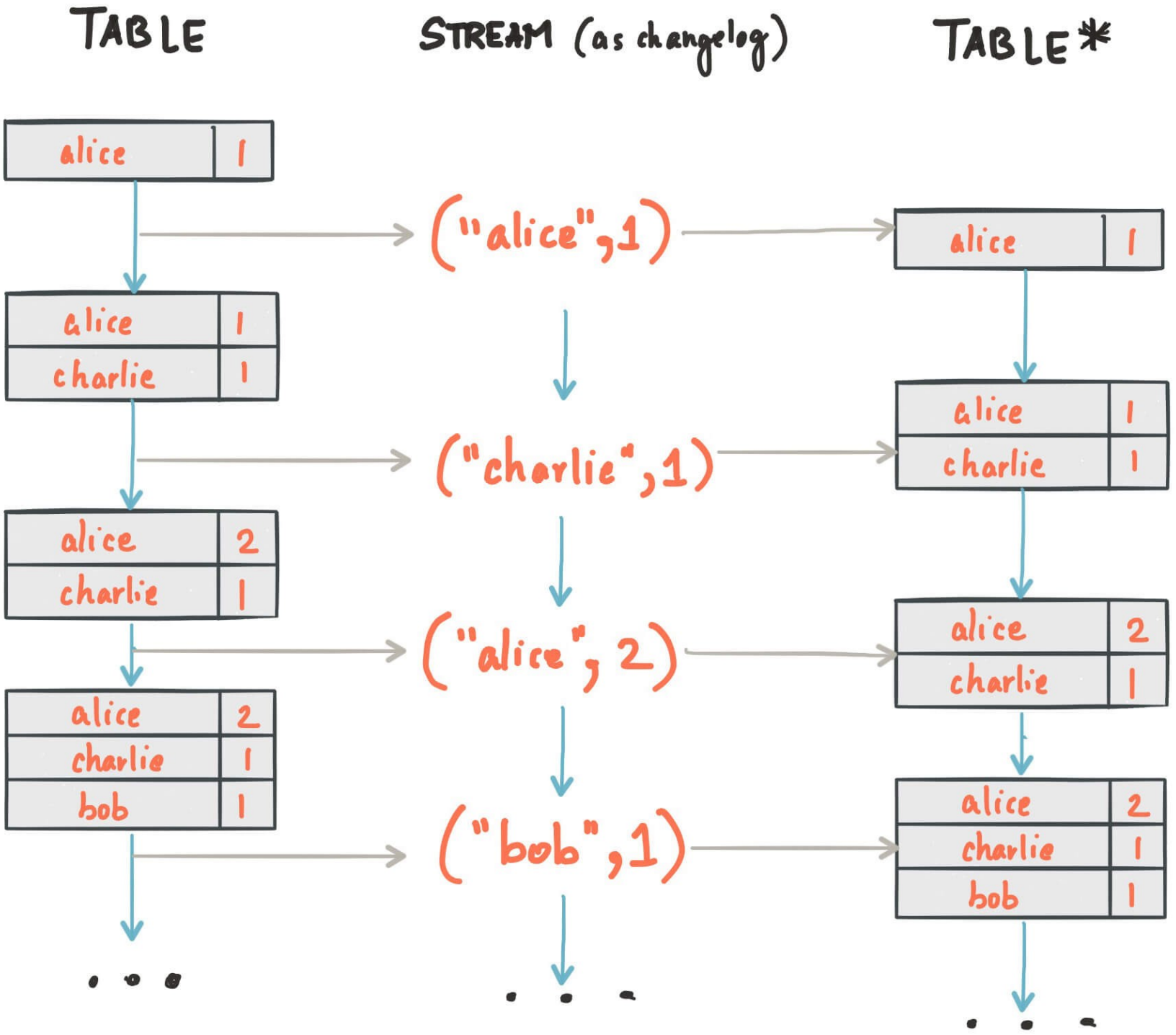
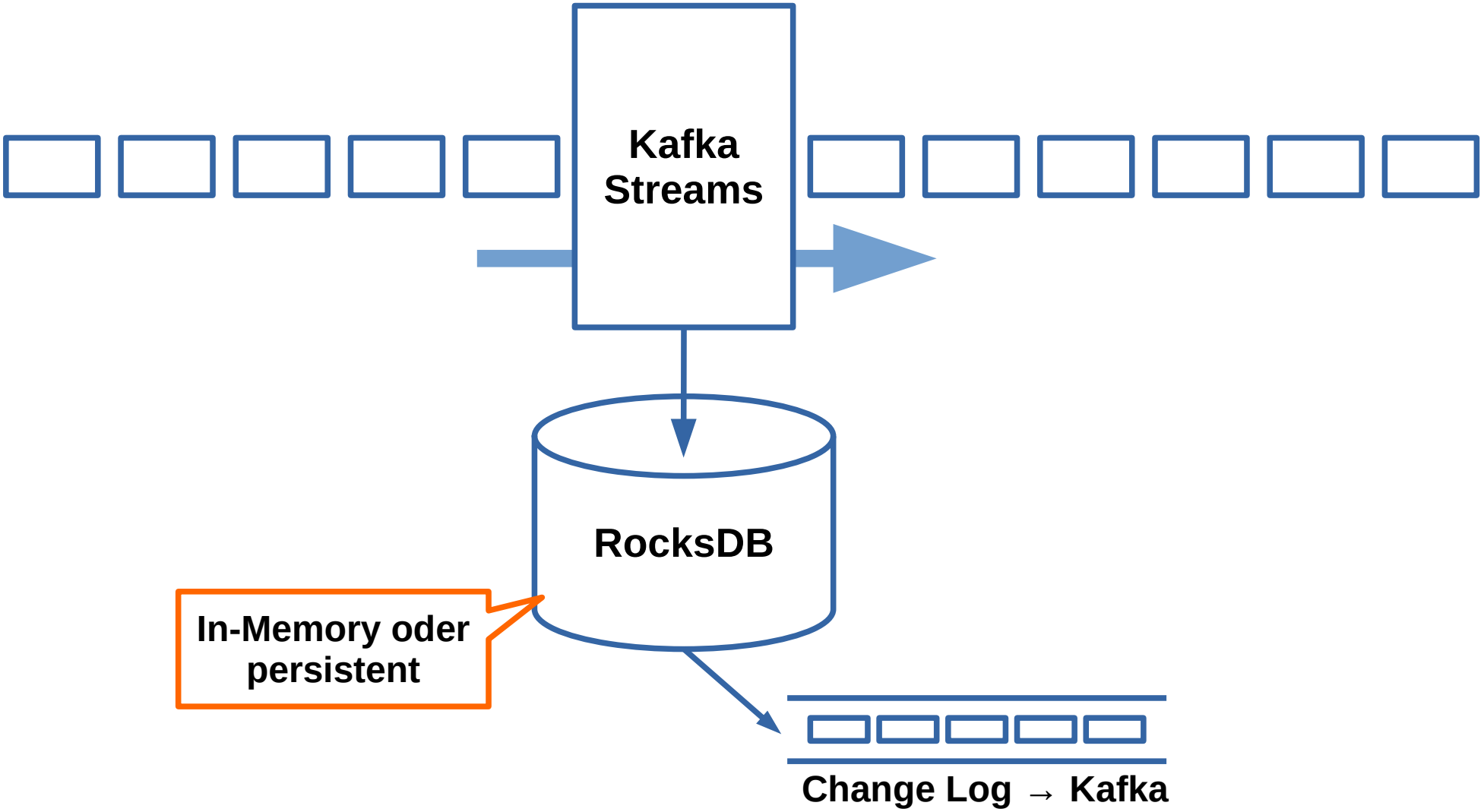


Bild: confluent.io

KStreams und KTables

- **KStream<K, V>**
  - Nachrichten in einem Topic als Update-Stream
- **KTable<K, V>**
  - Ist-Zustand, verändert durch Updates

Wo leben KTables?

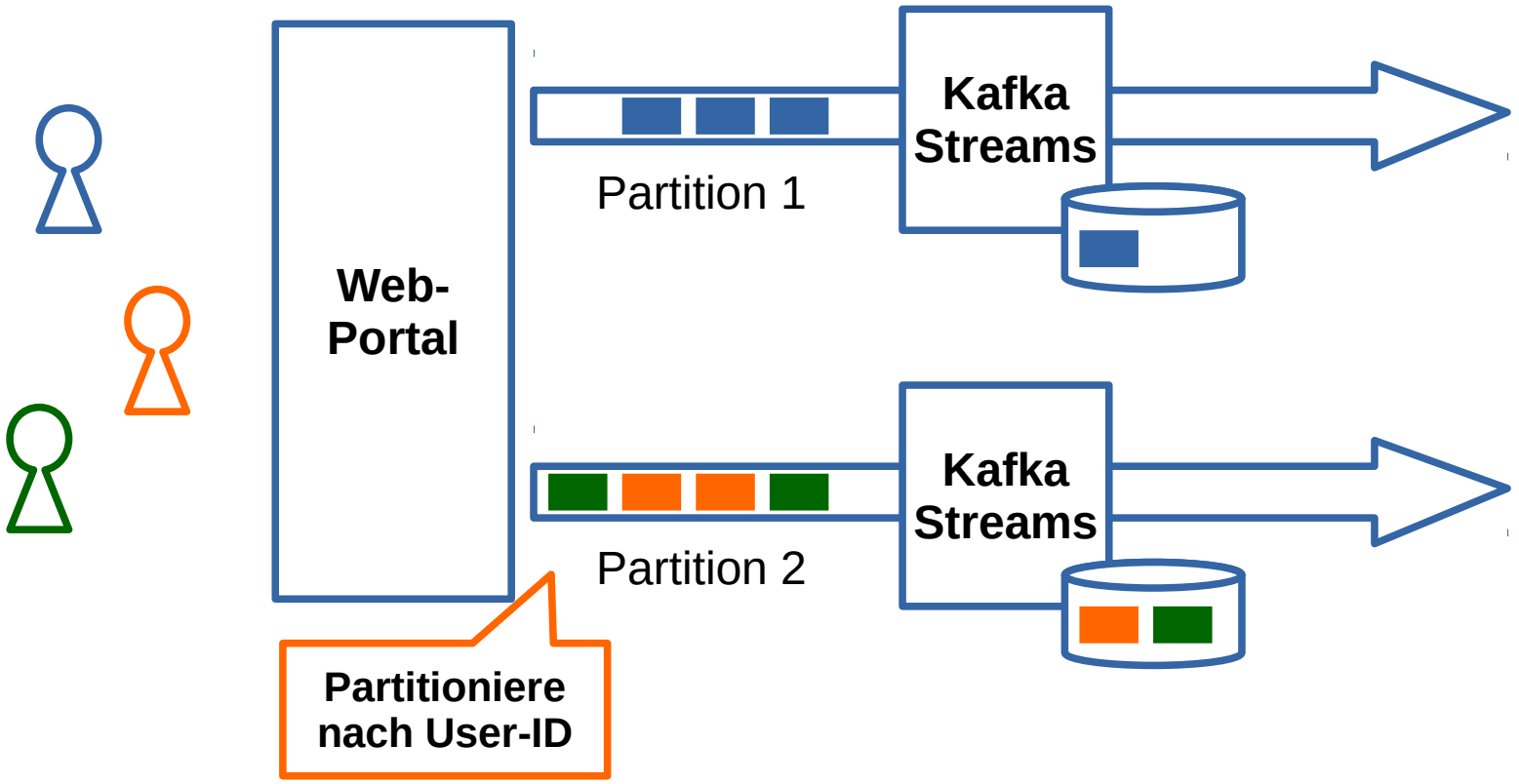


KTables: Beispiel

```
KStreamBuilder builder = new KStreamBuilder();

KStream<String, String> logs = builder.stream(
        Serdes.String(), Serdes.String(),
        "log");

KStream<String, String> status = logs.map(
         (k, v) -> new KeyValue<String, String>(v.split("\t")[0], null));

KTable<String, Long> countByKey = status.countByKey("counts");

countByKey.to("status-counts");
```

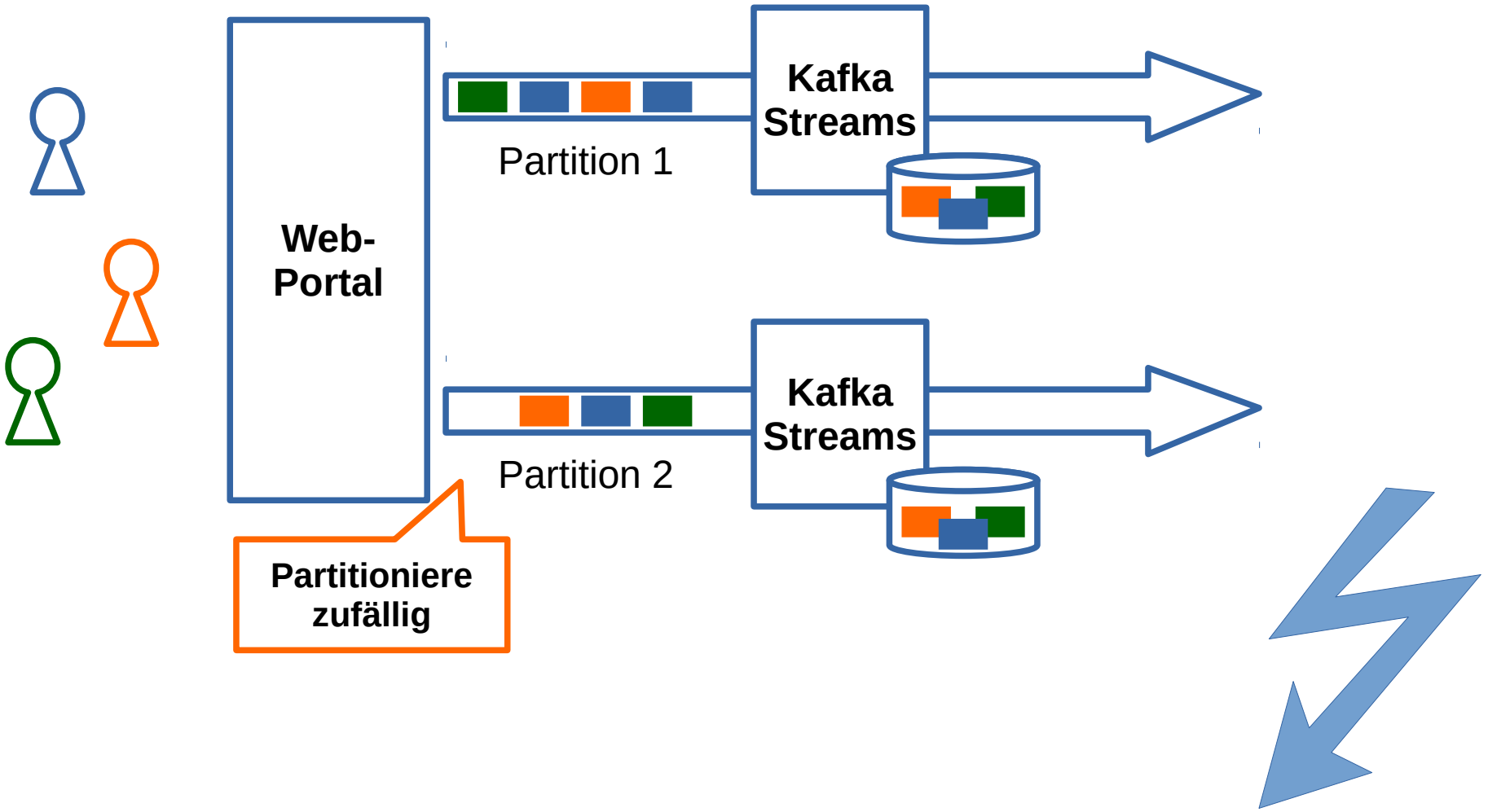
Partitionierung von Streams

Beispiel: Anzahl Zugriffe pro Benutzer



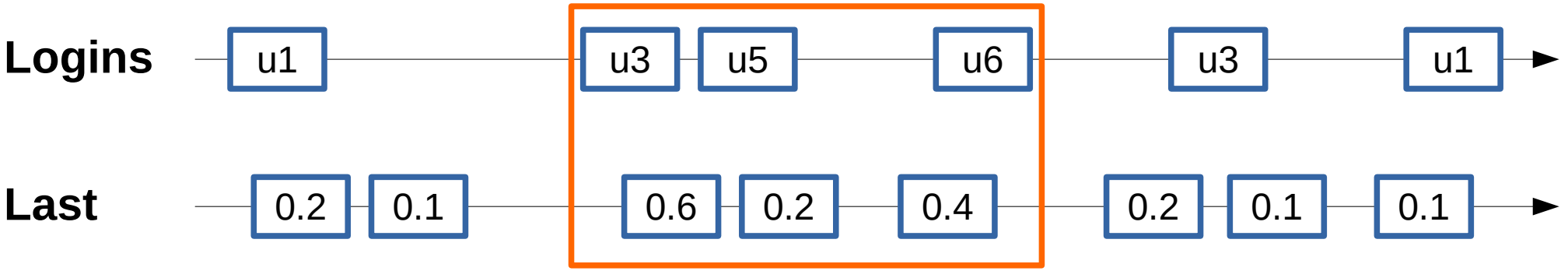
Partitionierung von Streams

Beispiel: Anzahl Zugriffe pro Benutzer



Joins

- **Join:** zusammengehörige Daten zusammenführen – aber welche?




```
KStream<Host, UserLogin> logins = …;
KStream<Host, LoadEntry> loadMeasurements = …;
ValueJoiner<UserLogin, LoadEntry, UserAndLoad> joiner
        = new UserLoadJoiner(…);

KStream<Host, UserAndLoad> join = logins.join(loads, joiner,
    JoinWindows.of("one minute").within(60_000));
```

Kafka Streams: Zusammenfassung

- **Streamverarbeitung** basierend auf Kafka
- Implementierung als **Kafka-Client**
- Wenig eigene **Infrastruktur**

- **Dualität** von Streams und Tabellen

- **Operationen** analog Spark etc.
- **Zustandsbehaftete Operationen** bleiben schwierig
- **Lokaler Zustand** pro Knoten