KAPITEL 5: DATA-STREAM-PROCESSING

LERNZIELE

- Erklären was ein Stream und in diesem Kontext eine Relation ist
- Erklären was es mit Stream-to-Relation-, Relation-to-Relation-, Relation-to-Stream-Operatoren auf sich hat, und an einem Beispiel die unterschiede darstellen
- Verschiedene Arten von Fenstern und deren Notwendigkeit erklären
- Stateful- und Stateless-Operatoren unterscheiden
- Kafka Streams anwenden können (z.B. in Pseudocode), um Data-Stream-Processing an einen Beispiel zu umzusetzen
- Kafka Streams und Apache Flink differenzieren, insbesondere hinsichtlich der Art der Ausführung von Topologien und der Skalierung von Tasks

5.1 MOTIVATION

Was ist ein Stream

> *A stream S is a (possibly infinite) bag (multiset) of elements $\langle s, t \rangle$, where s is a tuple belonging to the schema of S and $t \in T$ is the timestamp of the element.*

Ein Stream ist eine Menge von Tupeln in einem zeitlichen Kontext.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

Anzumerken ist, dass der Zeitstempel nicht Teil des Schemas eines Datenstroms ist und dass im Strom kein, ein oder mehrere Elemente mit demselben Zeitstempel geben kann. Es soll nur eine endliche (aber nicht begrenzte) Anzahl von Elementen mit einem bestimmten Zeitstempel geben.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

Dabei können zwei Klassen von Datenströmen unterschieden werden:

- **Basisdatenströme**: Quelldatenströme, die beim Datenstrom-Management-System ankommen. Diese Ströme könnten direkt oder indirekt von Quellen wie Sensoren stammen, z.B. über Message Brokern.
- **abgeleitete Datenströme**: Zwischenströme, die von Operatoren aus anderen Basisdatenströmen oder abgeleiteten Datenströmen erzeugt werden

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

Datenströme können nicht wie statische Daten verwaltet und verarbeitet werden, da sie besondere Merkmale wie Unvorhersehbarkeit aufweisen

- Es sind potenziell unendliche Menge von Daten
- Speichern auf der Festplatte ist nicht möglich, da durch die unendliche Menge von Daten eine unbegrenzten Speicherplatz benötigen
- Datenströme können nicht vollständig erfasst werden, bevor sie verarbeitet werden
- Streams weisen Eingangsschwankungen auf
- Systeme können das Eintreffen von Daten nicht antizipieren, da sie kontinuierlich und nicht in regelmäßigen Abständen eintreffen.
- Stream-Rate-Schwankungen können jederzeit auftreten und erfordern Anpassung des Ressourcenbedarfs während der Laufzeit

Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

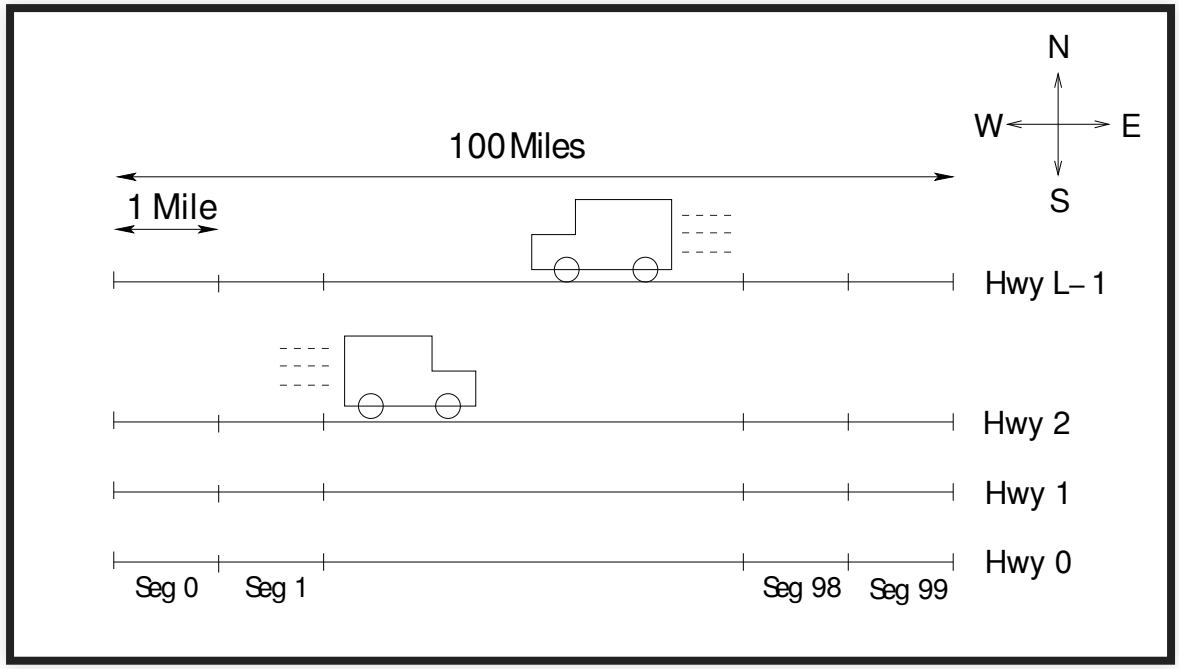
Beispiel

```
PosSpeedStr(vehicleId,speed,xPos,dir,hwy)
```

Ein Basis Datenstrom mit Fahrzeuginformationen.

Die `vehicleId` identifiziert das Fahrzeug, `speed` bezeichnet die Geschwindigkeit in kmh, `hwy` die Autobahnnummer, `dir` die Richtung (Ost oder West) und `xPos` die Position des Fahrzeugs auf der Autobahn. Als Zeitstempel könnte hier der Messzeitpunkt Verwendung finden.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.




```
PosSpeedStr(21,130,0,east,A9)
PosSpeedStr(42,130,99,west,A9)
PosSpeedStr(21,130,1,east,A9)
PosSpeedStr(42,130,98,west,A9)
PosSpeedStr(21,130,2,east,A9)
PosSpeedStr(42,130,97,west,A9)
...
```

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

A relation R is a mapping from T to a finite but unbounded bag of tuples belonging to the schema of R.

Eine Relation R definiert einen ungeordnete Menge von Tupeln zu jedem Zeitpunkt $t \in T$, bezeichnet als $R(t)$.

Unterschied zur Definition von Relation im relationalen Standardmodell: eine Relation ist eine Menge von Tupeln, ohne einen Begriff von Zeit, soweit es die Semantik relationaler Abfragesprachen betrifft.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

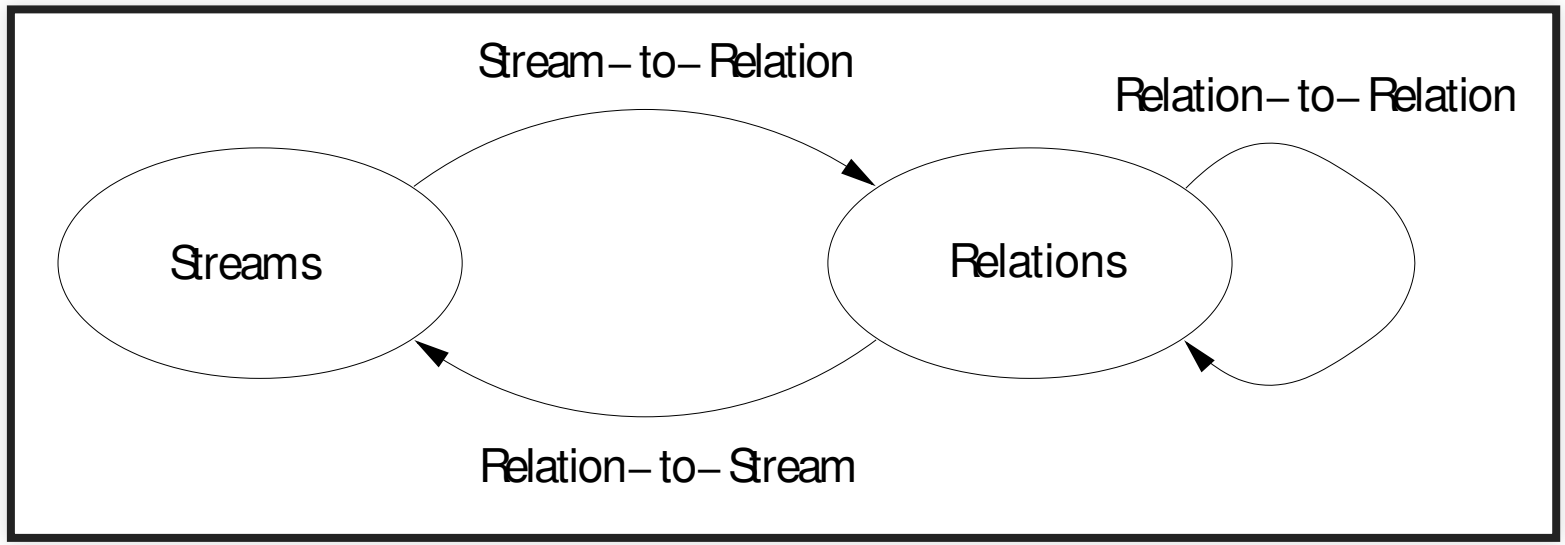
Beispiel einer abgeleiteten Relation

```
SegVolRel(segNo,dir,hwy,numVehicles)
```

Das Attribut `hwy` bezeichnet die Autobahnnummer, `dir` die Richtung, `segNo` das Segment innerhalb der Autobahn und `numVehicles` die Anzahl der Fahrzeuge im Segment. Zum Zeitpunkt `$t$` enthält `SegVolRel(t)` die Anzahl der Fahrzeuge in jedem Autobahnsegment zum Zeitpunkt `$t$`.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

Operatoren zwischen Streams und Relationen



Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

1. Ein Stream-to-Relation-Operator nimmt einen Stream S als Eingabe und erzeugt eine Relation R als Ausgabe mit demselben Schema wie S. Zu jedem Zeitpunkt t sollte $R(t)$ aus S bis t berechenbar sein.
2. Ein Relation-zu-Relation-Operator nimmt eine oder mehrere Relationen $R_1, ..., R_n$ als Eingabe und erzeugt eine Relation R als Ausgabe. Zu jedem Zeitpunkt t sollte $R(t)$ aus $R_1(t), ..., R_n(t)$ berechenbar sein.
3. Ein Relation-zu-Stream-Operator nimmt eine Relation R als Eingabe und erzeugt einen Stream S als Ausgabe mit demselben Schema wie R. Zu jedem Zeitpunkt t sollte S zu t aus R bis t berechenbar sein.

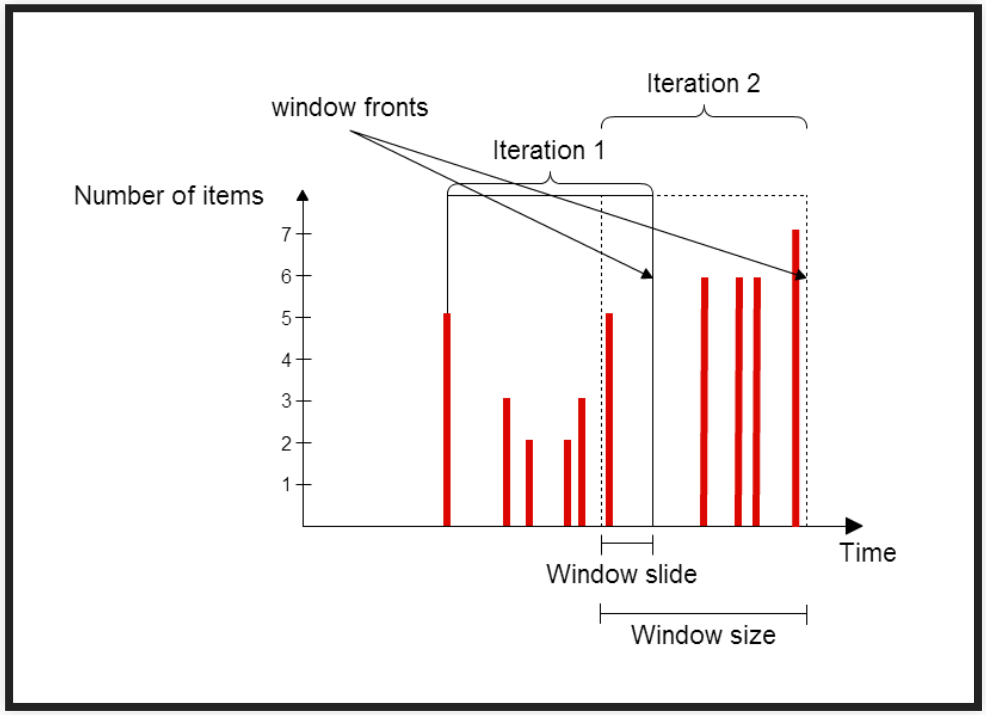
Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

WINDOW

Da Datenströme nicht im Speicher abgelegt werden können, besteht eine Alternative darin, nur die jüngsten Daten zu berücksichtigen, die als Berechnungsfenster bezeichnet werden, um ein Ergebnis auf der Grundlage einer Datenuntermenge zu erhalten.

> *A computation window is a logic stream discretization which is defined by a size and a slide. Considering the front of a window and the chronological order, the size defines the timestamp interval of elements to consider. The slide defines the step between two consecutive window fronts.*

Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.



Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

Zeitbasierte Fenster

A time-based window is defined on a time interval t_0 to t_n. A stream element e belongs to the window if $t_e \in [t_0, t_n)$ with t_e the timestamp of e.

Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

Tuple-basierte Fenster

> *A tuple-based sliding window on a stream S takes a positive integer N as a parameter and is specified by following the reference to S in the query with $[\text{Rows }N]$.*

Beispiel wäre ein Fenster für $\text{PosSpeedStr }[\text{Rows } 3]$, welches immer die letzten drei Elemente des Datenstroms umfasst.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

Partitionierende Fenster

> *A partitioned sliding window on a stream* `$S$` *takes a positive integer* `$N$` *and a subset* `$\{A_1, ..., A_k\}$` *of* `$S$`*'s attributes as parameters. It is specified by following the reference to* `$S$` *in the query with* `$[\text{Partition By } A_1, ..., A_k \text{ Rows } N]$`*.*

Beispiel wäre ein Fenster von `$\text{PosSpeedStr }[\text{Partition By } vehilceId \text{ Rows } 1]$`, welches das letzte Element eines Datenstroms mit einer entsprechenden `vehilceId` liefert.

Quelle: Arvind Arasu, Shivnath Babu, Jennifer Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. In The VLDB Journal, 2004. DOI: 10.1007/s00778-004-0147-z.

STATELESS VS. STATEFUL

Operatoren (Filtern, Verknüpfen, Sortieren ...) zur Erzeugung von Relationen, die auf Datenströme angewendet werden, müssen möglicherweise Element für Element oder einen logischen Block von Elementen behandeln.

Es werden zwei Hauptkategorien von Operatoren unterschieden: **zustandslose** und **zustandsabhängige** Operatoren.

Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

Stateless Operators

- Zustandslose Operatoren, z. B. Filtern auf der Grundlage eines Attributwerts, verarbeiteten Datenströme Element für Element
- Liefern ein neues Ergebnis mit einer unvorhersehbaren Häufigkeit, z.B. Filter liefert nichts, wenn seine Eingabe das Filterprädikat nicht erfüllt.
- Haben keine Informationen über die aktuelle Iteration des Berechnungsfensters oder frühere Ergebnisse, wenn sie ein Ergebnis aus einem Datenstromelement berechnen
- Dennoch kann ein zustandsloser Operator historische Daten verwenden, die im lokalen Speicher oder auf der Festplatte gecached sind.

Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

Stateful Operators

- Zustandsbehaftete Operatoren verwenden als Eingabe einen logischen Block von Stream-Elementen, um ein einziges Ergebnis für die gesamte Menge zu berechnen
- Der Stream-Element-Block kann durch einen Bereich von Zeitstempeln (time-based) oder eine Anzahl (count-based) von zu berücksichtigenden Elementen definiert werden
- Zustandsbehaftete Operatoren speichern Informationen wie die aktuelle Iteration des Fensters oder die Liste der verschiedenen Elementwerte, um ein Ergebnis aus allen berücksichtigten Elementen zu verarbeiten
- Die während der Laufzeit eines zustandsbehafteten Operators erzeugten Informationen werden als sein Zustand bezeichnet

Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

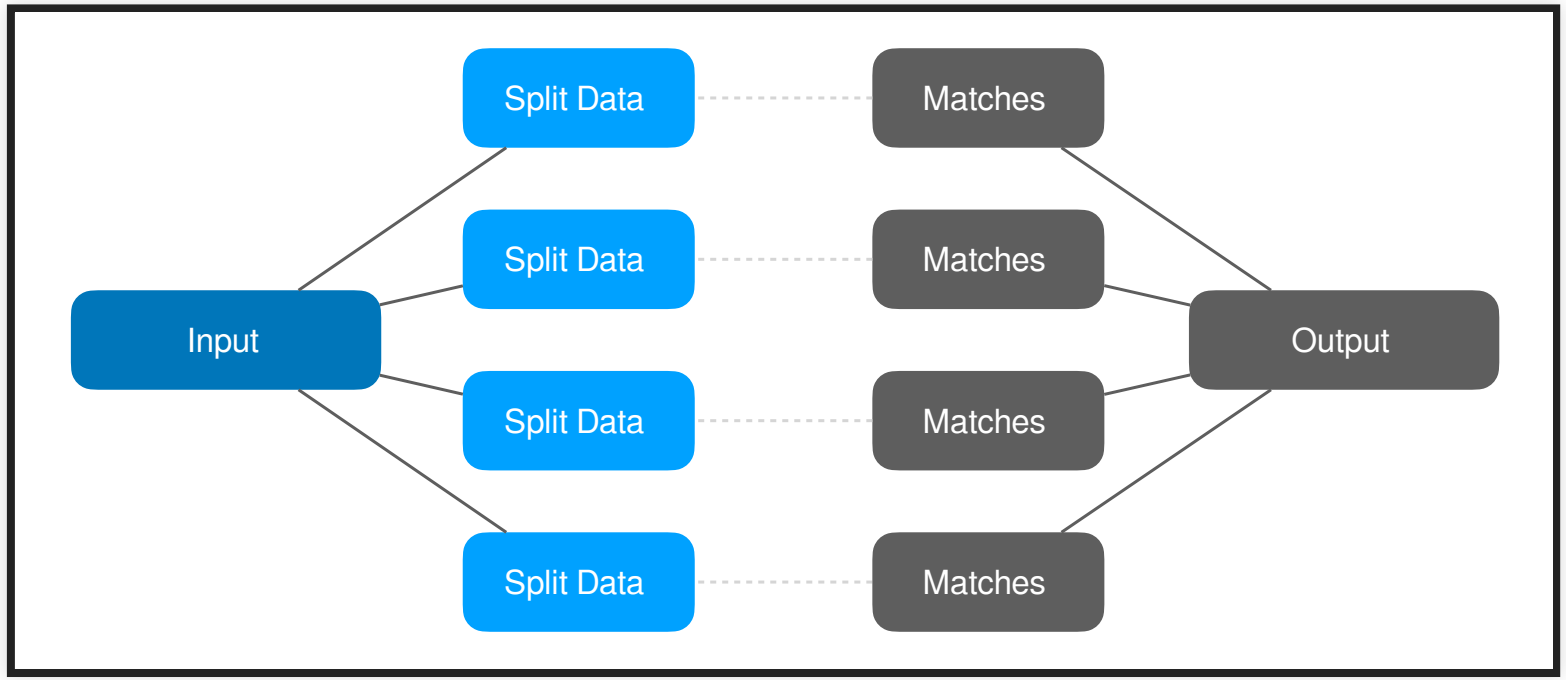
Beispiel

- Ein fensterbasierter zustandsbehafteter Operator berechnet die Summe eines Elementattributs für jedes Berechnungsfenster
- Sein Zustand enthält den Bezeichner der aktuellen Iteration, die zu berücksichtigenden Definitionselemente für jeden Eingabeblock und den aktuellen Summenwert

Quelle: Roland Kotto-Kombi, Nicolas Lumineau, Philippe Lamarre, and Yves Caniou. Parallel and Distributed Stream Processing: Systems Classification and Specific Issues. 2015.

MAPREDUCE

Traditioneller Weg

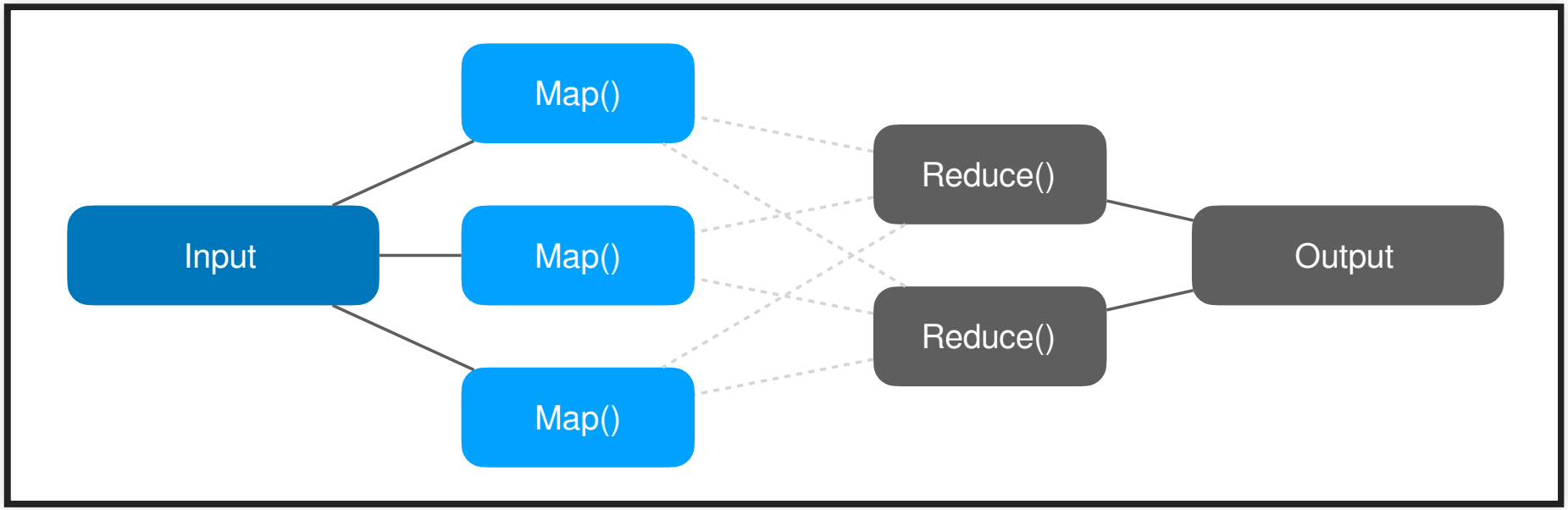


Quelle: https://medium.com/edureka/mapreduce-tutorial-3d9535ddbe7c

Für die traditionelle Verarbeitung großer Datenmengen im Cluster ist die Aufteilung der Datenmenge in kleinere Teile, deren Verteilung auf verschiedene Knoten, die Bearbeitung und anschließende zusammenstellung des Ergebnisses notwendig.

Quelle: https://medium.com/edureka/mapreduce-tutorial-3d9535ddbe7c

Alternative: MapReduce



Mittels MapReduce-Framework lassen sich Aufgaben wie die Verarbeitung großer Datenmengen einfacher für parallele Umgebung auf- und verteilen.

Quelle: https://medium.com/edureka/mapreduce-tutorial-3d9535ddbe7c

- MapReduce besteht aus zwei verschiedenen Aufgaben - Map und Reduce (vergleichbar mit den entsprechenden Operatoren in Rx)
- Die Reduktionsphase erfolgt nach Abschluss der Mapping-Phase
- Der Map-Job, bei dem ein Datenblock gelesen und verarbeitet wird, um dient dazu Schlüssel-Wert-Paare als Zwischenergebnisse zu erzeugen
- Die Ausgabe eines Mapper- oder Map-Auftrags (Schlüssel-Wert-Paare) ist die Eingabe für den Reducer
- Der Reducer empfängt die Schlüssel-Wert-Paare von mehreren Map-Jobs.
- Anschließend aggregiert der Reducer diese Zwischendatentupel (Zwischen-Schlüsselwertpaare) zu einer kleineren Menge von Tupeln oder Schlüsselwertpaaren, die die endgültige Ausgabe darstellen.

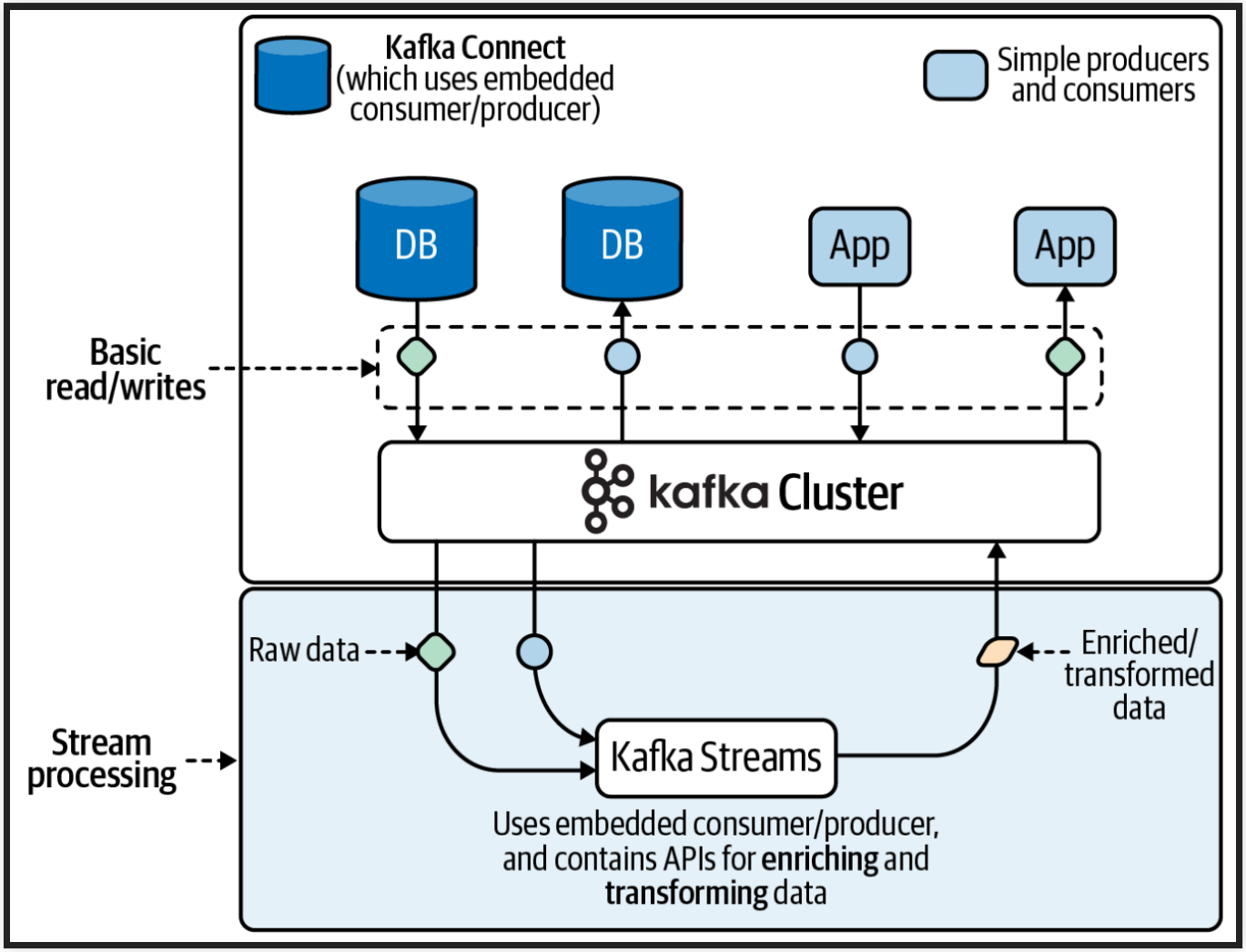
Quelle: https://medium.com/edureka/mapreduce-tutorial-3d9535ddbe7c

5.2 KAFKA STREAMS

> *Unlike the Producer, Consumer, and Connect APIs, Kafka Streams is dedicated to helping you process real-time data streams, not just move data to and from Kafka.*

Kafka Streams arbeiten als Schicht im Kafka-Ökosystem damit Daten aus verschiedenen Quellen zusammenlaufen. Ziel ist es Datenanreicherung, -umwandlung und -verarbeitung zu realisieren.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Kafka Streams ...

- stellt eine High-Level-DSL bereit, die vergleichbar ist mit der Streaming-API von Java bzw. Operatoren in Reactive Programming; die DSL ermöglicht einen funktionalen Ansatz für die Verarbeitung von Datenströmen
- stellt eine Low-Level-Prozessor-API bereit
- ermöglicht Abstraktionen für die Modellierung von Daten als Streams oder Tabellen
- ermöglicht Datenströme und Tabellen für die Datentransformation und -anreicherung zu verbinden
- bieten Operatoren für die Erstellung von zustandslosen und zustandsabhängigen Stream-Transformationen
- ermöglicht zeitbasierte Operatoren, inkl. Windowing und periodische Ausführung

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Scalability

- Skalierung wird über Consumer-Groups und Partitionierung erreicht
- Ein Stream-Knoten erlaubt die Verarbeitung von Daten innerhalb der zugewiesenen Partitionen
- Mehrere Consumer-Groups können parallel und unabhängig auf den selben Daten arbeiten

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Reliability

- Kafka bringt über Partitionen und automatisierte Neuverteilung bei Ausfall eines Consumers innerhalb einer Gruppe eine Failover-Strategie mit
- Fällt ein Knoten aus, werden die Partitionen auf die verbleibenden Knoten verteilt
- Commits auf Queues erlaubt es den bisherigen Verarbeitungsstand genau zu protokollieren, selbst wenn alle Knoten einer Gruppe weggefallen sind könnte der nächste Knoten einer Gruppe am bisherigen Verarbeitungsstand ansetzen

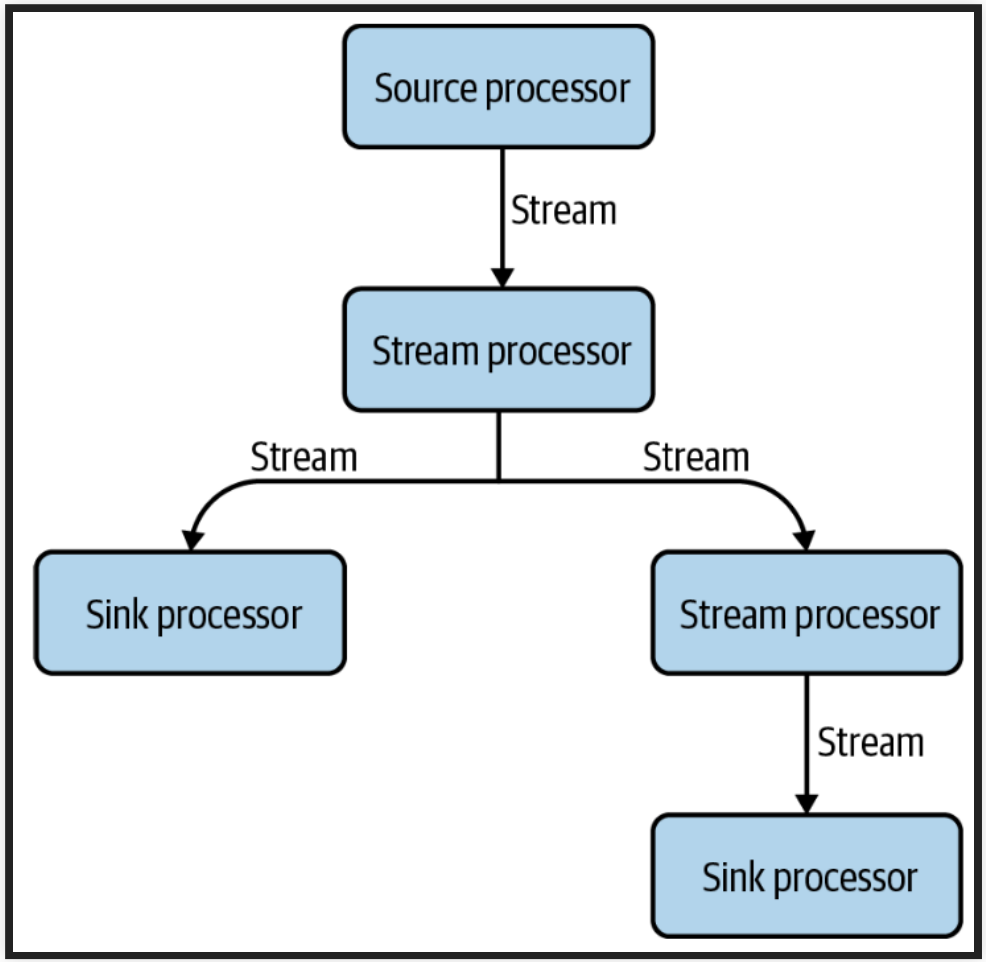
Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Maintainablitity

- Stream-Anwendungen werden als eigener Java-Prozess (Standalone) gestartet
- Ansätze zur Fehlerbehebung und Überwachung können hier wie üblich Anwendung finden
- Quellcode ist durch DSL ähnlich zu Streaming- und Reactive-Ansätzen

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

PROCESSING TOPOLOGIES

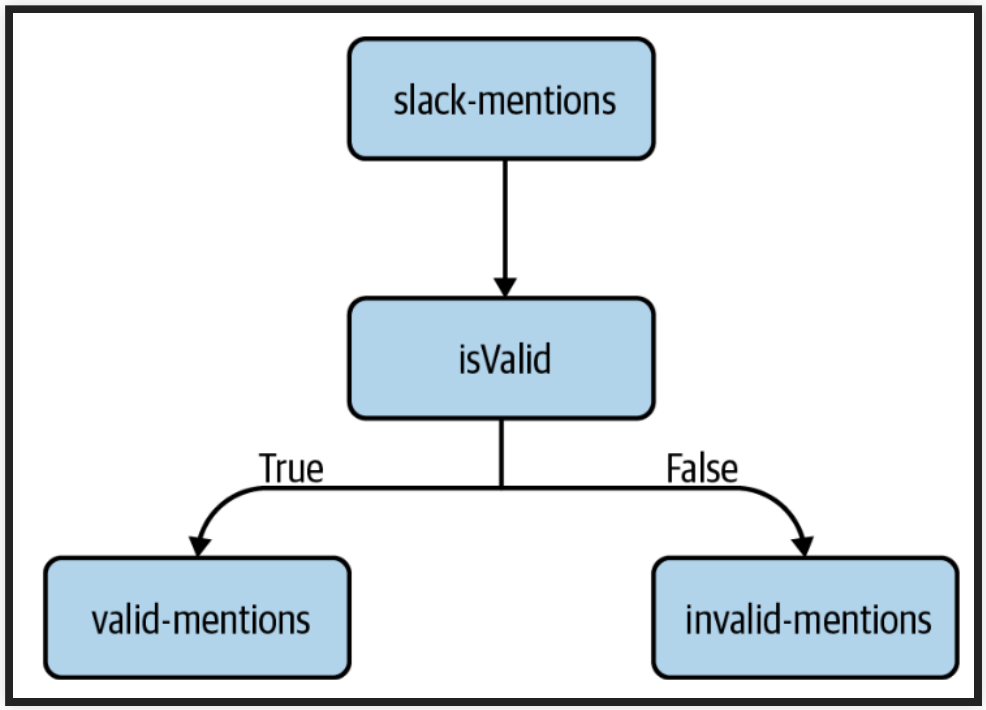


Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

- **Source-Prozessoren**: Über die Sources fließen die Informationen in die Kafka-Streams-Anwendung. Die Daten werden aus einem Kafka-Queue gelesen und an einen oder mehrere Stream-Prozessoren gesendet.
- **Stream-Prozessoren**: Diese Prozessoren sind für die Datenverarbeitungs-/Transformationslogik auf den Input-Stream verantwortlich. Mit der DSL werden diese Prozessoren über eine Reihe von Operatoren definiert. Einige Beispieloperatoren sind filter, map, flatMap und join.
- **Sink-Prozessoren**: In Senken werden angereicherte, umgewandelte, gefilterte oder anderweitig verarbeitete Datensätze zurück nach Kafka geschrieben, um entweder von einer anderen Stream-Verarbeitungsanwendung verarbeitet oder über eine Anwendung wie Kafka Connect an einen nachgeschalteten Datenspeicher gesendet zu werden. Wie Quellprozessoren sind auch Senkenprozessoren mit einem Kafka-Topic verbunden.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Beispiel für Verarbeitung



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

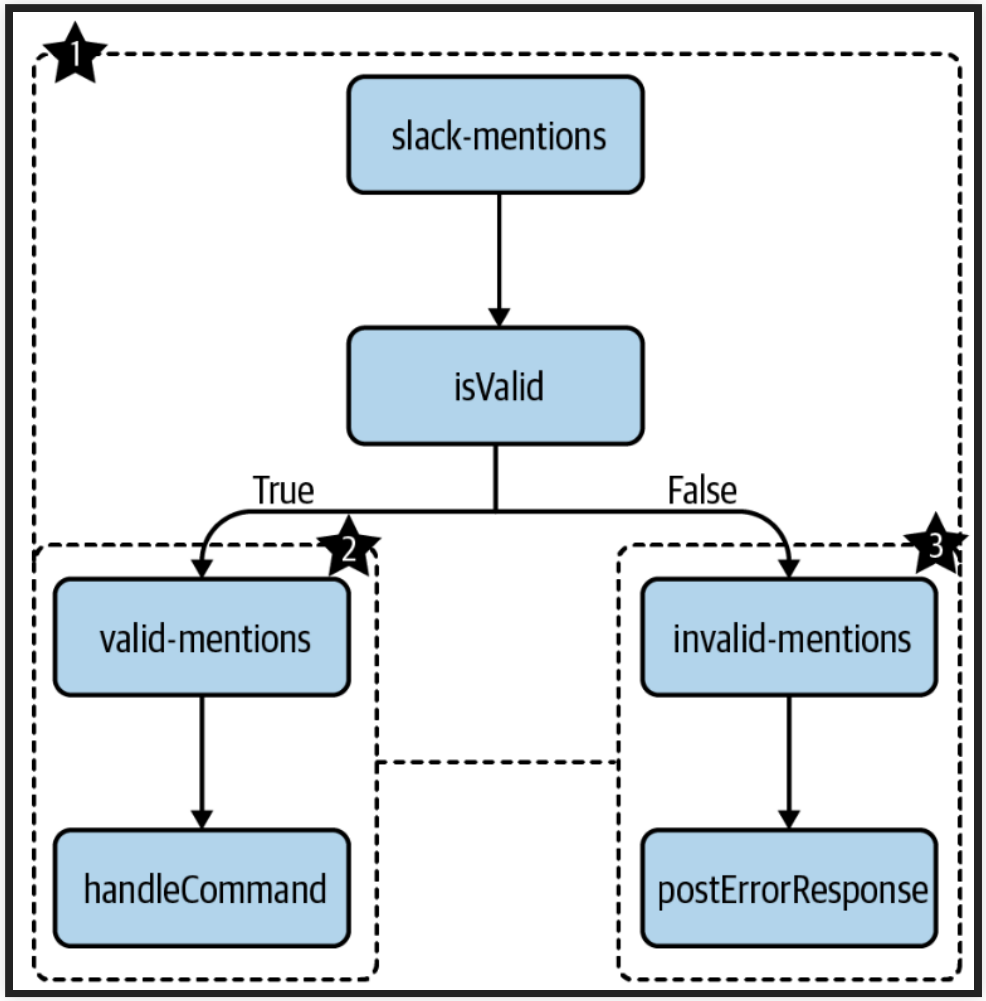
Sub-Topologies

Innerhalb einer Stream-Anwendung ist es möglich Topologien zu unterteilen

Unterteilung wird realisiert, indem Zwischenergebnisse in Queues geschrieben werden, die anschließend mit einer eigenen Verarbeitungslogik behandelt werden.

Mit der Aufteilung ist die parallele Verarbeitung möglich und insbesondere für rechenintensive Aufgaben oder bei Verwendung der Ergebnisse in verschiedenen folgenden Verarbeitungsschritten sinnvoll.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



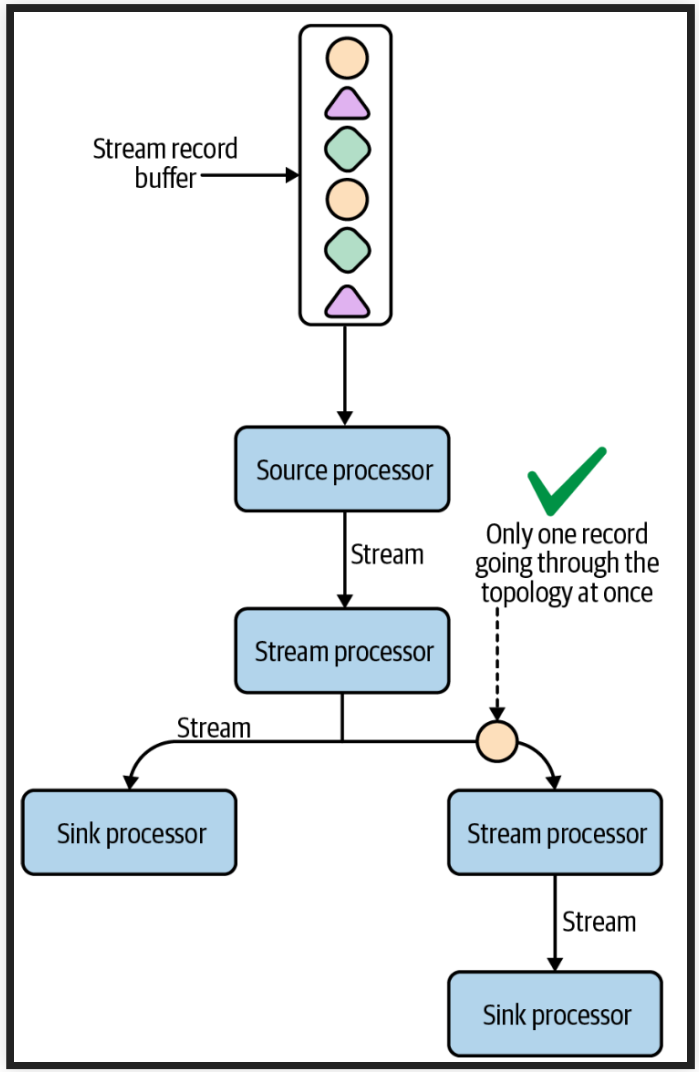
Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Depth-First Processing

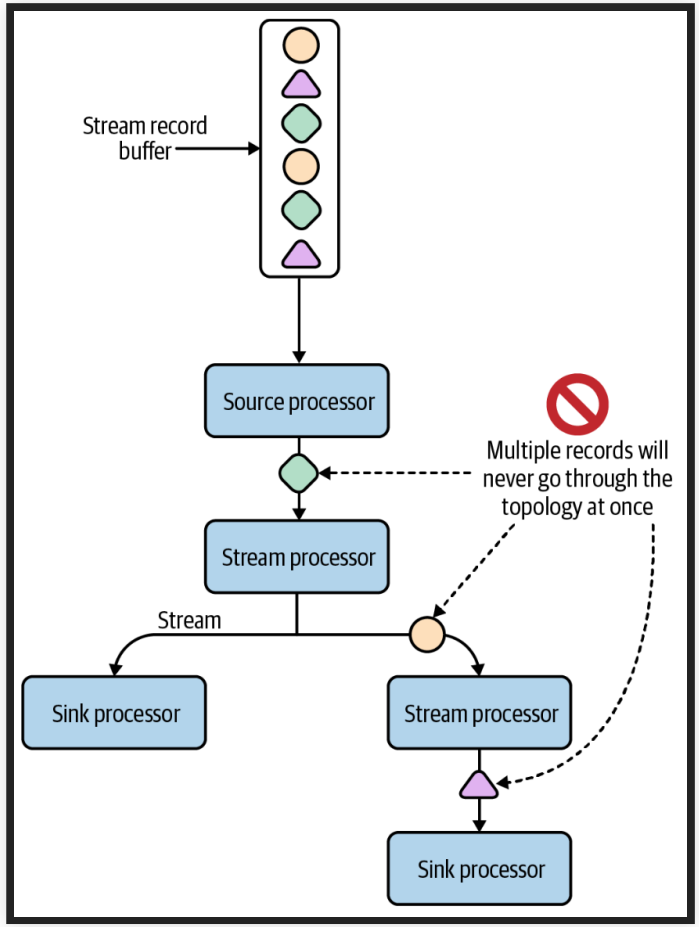
Kafka Streams verwendet bei der Verarbeitung von Daten eine Depth-First-Strategie. Wenn ein neuer Datensatz empfangen wird, wird er durch jeden Stream-Prozessor in der Topologie geleitet, bevor ein weiterer Datensatz verarbeitet wird.

Dies bezieht sich auf eine Abfolge von Prozessoren und kann durch Sub-Topologien durchbrochen werden.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

FIRST EXAMPLE

Mittels `StreamsBuilder`-Klasse werden Source-, Stream- und Sink-Prozessoren verknüpft.

```
StreamsBuilder builder = new StreamsBuilder();
builder.stream("hello-world")
    .foreach((key, value) -> System.out.println("Hello, " + value));
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

- Das HelloWorld-Beispiele benötigen die Instanz der entsprechenden `StreamsBuilder`-Klasse
- ... welche anschließend genutzt wird um die Konfiguration zu starten.
- Über Aufrufe am `builder` wird die Verarbeitungskette konfiguriert.

Sollen Ergebnisse Verarbeitet und über eine Senke zurück an Apache Kafka kommen weitere Operatoren zum Einsatz.

```
StreamsBuilder builder = new StreamsBuilder();
builder.stream("hello-world")
    .map((key, value) -> KeyValue.pair(key, "Hello, " + value))
    .to("hello-world-answer");
```

- Source ist hier die Queue `hello-world`
- Jeder Eintrag wird mittels `map`-Operator in eine Hello-Meldung umgewandelt
- Das Ergebnis wird in der Queue `hello-world-answer` veröffentlicht

Bleibt der Key unverändert, kann der `mapValues`-Operator verwendet werden.

```
StreamsBuilder builder = new StreamsBuilder();
builder.stream("hello-world")
    .mapValues((value) -> "Hello, " + value)
    .to("hello-world-answer");
```

Mit dem `mapValues`-Operator wird der eingehende Key als ausgehender verwendet.

Ausführbare Beispiele

Enthält eine `docker-compose.yml` für Apache Kafka und Zookeeper, welche mit `docker compose up -d` gestartet werden kann.

Nutzen Sie das bereitgestellte Projekt-Archiv apache-kafka-streams.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Zum Ausprobieren können in der Konsole folgende Befehle für Consumer und Producer verwendet werden.

```
docker compose exec kafka kafka-console-producer.sh --topic hello-world \
    --bootstrap-server localhost:9092

docker compose exec kafka kafka-console-consumer.sh --topic \
    hello-world-answer --group foo --from-beginning --bootstrap-server \
    localhost:9092
```

Anschließend können in der Consumer-Konsole Eingaben platziert werden, welche vom Stream verarbeitet werden sollten.

Topologie anzeigen

Das Ergebnis von `StreamBuilder.build()` ist die Topologie. Diese kann als ASCII-Repräsentation angezeigt werden:

```
StreamBuilder builder = new StreamBuilder();
// ...
Topology t = builder.build();
System.out.println(t.describe());
```

Die Ausgabe kann anschließend mit Visualisieren angezeigt werden.

SERDES

Serializer und Deserializer

Für die Verarbeitung von Daten von Topics und die Veröffentlichung auf Topics ist die Serialisierung und Deserialisierung von Daten notwendig. Kafka Streams verlangt daher die Konfiguration von Default-Serialisierung und -Deserialisierung bzw. die Angabe spezifischer Serialisierungs- und Deserialisierungsmethoden an verschiedenen Stellen, z.B. bei der Erzeugung eines Streams.

```
KStream<String, String> someEvent = builder.stream("my-events",
    Consumed.with(Serdes.String(), Serdes.String()));
```

Hier wird der Key und Value des Datenstrom als String verwendet.

Die Klasse Serdes bietet verschiedene Methoden, um typische Basisdatentypen zu verarbeiten. Für spezielle Fälle kann mit...

```
Serdes.<T>serdeFrom(Serializer<T>, Deserializer<T>)
```

... ein neuer Serdes erzeugt werden. Hierfür ist eine Implementierung der Serializer und Deserializer Schnittstelle notwendig.

Beispiel für JSON-Serdes der Klasse MyData

```
public class JsonSerdes {
    private static final ObjectMapper mapper = new ObjectMapper();
    public static Serde<MyData> myData() {
        return Serdes.serdeFrom((topic, data) -> writeValue(data),
                (topic, data) -> readValue(data, MyData.class));
    }
    private static <T> T readValue(byte[] data, Class<T> clazz) {
        try {
            return mapper.readValue(data, clazz);
        } catch (IOException e) { return null; }
    }
    private static <T> byte[] writeValue(T data) {
        try {
            return mapper.writeValueAsBytes(data);
        } catch (IOException e) { return new byte[0]; }
    }
}
```

Das Beispiel zeigt eine einfache Verwendung des ObjectMappers der Jackson-Bibliothek, welche meist in Spring zum Einsatz kommt.

Eine Verwendung in Kafka Streams sieht dann wie folgt aus:

```
KStream<String, MyData> someEvent = builder.stream("my-events",
    Consumed.with(Serdes.String(), JsonSerdes.myData()));
```

STATELESS-PROCESSING

Die DSL in Kafka Streams liefert Operatoren, die ohne eigenen Zustand die Verarbeitung eines Datenstroms ermöglicht:

- Filtern von Records: `filter`, `filterNot`
- Verzweigen von Streams: `split`
- Transformieren von Records: `map` und `mapValues`
- Transformieren von Records in einen oder mehrere Records: `flatMap` und `flatMapValues`
- Zusammenführen von Streams: `merge`

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Filter

```
builder.<Void, String>stream("hello-world")
    .filter((key, value) -> !value.isBlank())
    .to("hello-world-answer");
```

Prüft ob der eingegebene Wert nicht leer bzw. aus Whitespaces besteht.

Verzweigen

```
builder.<Void, String>stream("hello-world")
    .split()
        .branch((key, value) -> !value.isBlank(), Branched.withConsumer(subS
            .to("hello-world-answer")))
        .defaultBranch(Branched.withConsumer(ks -> ks
            .to("hello-world-failed")));
```

Je Branch wird ein Predicate angegeben, welches wie bei einem Filtern die Bedingung setzt. Anschließend wird eine KStream-Verarbeitung für den Zweig erzeugt.

Dies kann wie bei einer if-Cascade bis zu einer `defaultBranch`-Konfiguration fortgesetzt werden.

Alternativ mit benannten Streams

```
Map<String, KStream<Void, String>> branches = builder.<Void, String>stream("
    .split(Named.as("Branch-"))
        .branch((key, value) -> !value.isBlank(), Branched.as("A"))
        .defaultBranch(Branched.as("B"));
branches.get("Branch-A").to("hello-world-answer");
branches.get("Branch-B").to("hello-world-failed");
```

- Der Prefix wird beim Split angegeben
- Anschließend werden alle resultierende Branches mittels Branched.as benannt
- Ergebnis der defaultBranch-Methode ist eine Map, welche verwendet werden kann um auf die Streams zuzugreifen
- ... im Beispiel werden die Ergebnisse in unterschiedlichen Queues veröffentlicht

Transformieren

Operatoren wie `map` oder `mapValues` erlauben die Transformation von Records.

```
builder.<Void, String>stream("hello-world")
    .split()
        .branch((key, value) -> !value.isBlank(), Branched.withConsumer(subS
            .mapValues(value -> "Hello, " + value)
            .to("hello-world-answer")))
        .defaultBranch(Branched.withConsumer(ks -> ks
            .mapValues(value -> "Hello, nobody!")
            .to("hello-world-answer")));
```

- Hier wird der Wert in eine erweiterte Meldung umgewandelt.
- Bzw. durch eine Default-Meldung ersetzt.

Transformieren in ein oder mehrere Records

Operatoren wie `flatMap` oder `flatMapValues` erlauben eingehende Records in kein, ein oder mehrere neue Records zu überführen.

```
builder.<Void, String>stream("hello-world")
    .split()
        .branch((key, value) -> !value.isBlank(), Branched.withConsumer(subS
            .flatMapValues(value -> Arrays.asList(value.split(",")))
            .mapValues(value -> value.trim())
            .mapValues(value -> "Hello, " + value)
            .to("hello-world-answer")))
        .defaultBranch(Branched.withConsumer(ks -> ks
            .mapValues(value -> "Hello, nobody!")
            .to("hello-world-answer")));
```

Die eingehende Nachricht wird anhand eines Kommas aufgeteilt und eine Liste erzeugt die weiter verwendet wird.

Zusammenführen

```
Map<String, KStream<Void, String>> branches = builder.<Void, String>stream("
    .split(Named.as("Branch-"))
        .branch((key, value) -> !value.isBlank(), Branched.as("A"))
        .defaultBranch(Branched.as("B"));
branches.get("Branch-B")
    .mapValues(value -> "nobody")
    .merge(branches.get("Branch-A"))
    .mapValues(value -> "Hello, " + value)
    .to("hello-world-answer");
```

- Der Stream mit dem Namen `Branch-B` wird über eine Transformation verändert
- ... und anschließend mit dem Stream `Branch-A` zusammengeführt
- Das unsortierte und vermischte Ergebnis wird dann weiterverarbeitet

SINK-PROCESSORS

To

Aktuellen Stream in einem Topic platzieren

```
Map<String, KStream<Void, String>> branches = builder.<Void, String>stream("
    .split(Named.as("Branch-"))
        .branch((key, value) -> !value.isBlank(), Branched.as("A"))
        .defaultBranch(Branched.as("B"));
branches.get("Branch-B")
    .mapValues(value -> "nobody")
    .merge(branches.get("Branch-A"))
    .flatMapValues(value -> Arrays.asList(value.split(",")))
    .mapValues(value -> value.trim())
    .mapValues(value -> "Hello, " + value)
    .to("hello-world-answer");
```

Wie bereits mehrmals verwendet, kann to als Sink-Processor am Ende einer Verarbeitungskette verwendet werden um die Ergebnisse entsprechend bereitzustellen.

Repartition

Aktuellen Stream in einer auto-generated Topic platzieren und direkt in einem neuen Stream verwenden

```
Map<String, KStream<Void, String>> branches = builder.<Void, String>stream("
    .split(Named.as("Branch-"))
        .branch((key, value) -> !value.isBlank(), Branched.as("A"))
        .defaultBranch(Branched.as("B"));
branches.get("Branch-B")
    .mapValues(value -> "nobody")
    .merge(branches.get("Branch-A"))
    .repartition()
    .flatMapValues(value -> Arrays.asList(value.split(",")))
    .mapValues(value -> value.trim())
    .mapValues(value -> "Hello, " + value)
    .to("hello-world-answer");
```

- Das Split- und Merge-Beispiel verwendend, bietet es sich an, nach der Zusammenführung eine Repartitionierung zu verwenden.
- Diese veröffentlicht alle Ereignisse in einer generischen Queue und erzeugt aus dieser einen neuen Stream (vgl. `builder.stream`)
- Die nachfolgenden Prozessoren können sind damit eine Sub-Topology

STATEFUL-PROCESSING

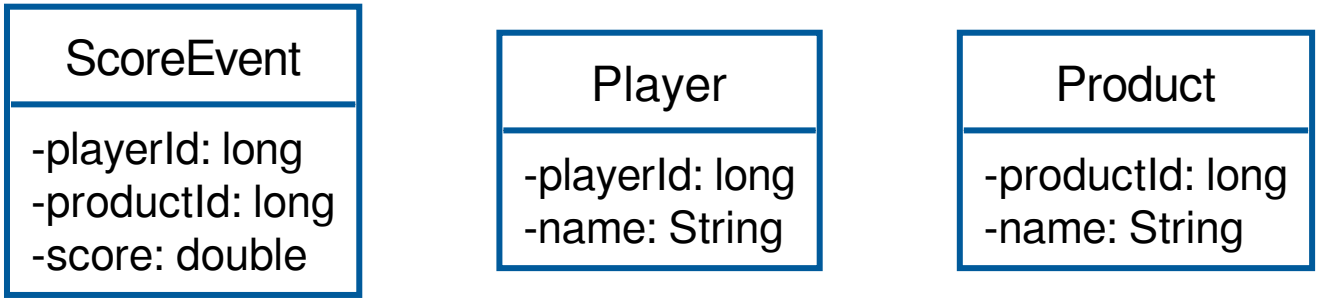
Die zustandsorientierte Verarbeitung hilft, die Beziehungen zwischen Ereignissen zu verstehen und diese Beziehungen für fortgeschrittene Anwendungsfälle der Stream-Verarbeitung zu nutzen. Hier stehen insbesondere im Vordergrund:

- Erkennen von Mustern und Verhaltensweisen in Ereignisströmen
- Aggregationen
- Daten durch Joins anreichern

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Beispiel

Im folgenden soll ein Beispiel für ein Leaderboard umgesetzt werden, welches sich an der Vorlage aus der Quelle orientiert. Hierfür kommen drei Queues zum Einsatz: ScoreEvent, Player und Product.



Das Beispiel enthält eine `docker-compose.yml` für Apache Kafka und Zookeeper, welche mit `docker compose up -d` gestartet werden kann.

Nutzen Sie das bereitgestellte Projekt-Archiv apache-kafka-streams-highscore.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

State Stores

- Für Stateful-Processing sind Zustandsspeicher notwendig, um Zustände über mehrere Ereignisse hinweg zu speichern
- Soll z.B. die Anzahl von Fehlern in einer Logging-Queue gezählt werden, muss die Zahl gespeichert werden
- Kafka Streams bietet hierfür eine Abstraktion über State Stores, welche es erlaubt Zustandsinformationen zu speichern
- Es stehen mehrere State-Store-Implementierungen zur Verfügung, welche unterschiedliche Vor- und Nachteile mitbringen

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

In-Memory State-Store-Implementierungen

Die Default-State-Store-Implementierung ist In-Memory oder Embedded in der Stream-Anwendung.

- Speicherung erfolgt direkt in der Ausführungsumgebung
- Kein Zugriff auf externe Speicher-Systeme
- Keine Netzwerk-Latenz bzw. Zugriffsverzögerungen
- Zustand bei Skalierung nur im jeweiligen Consumer verwaltet

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

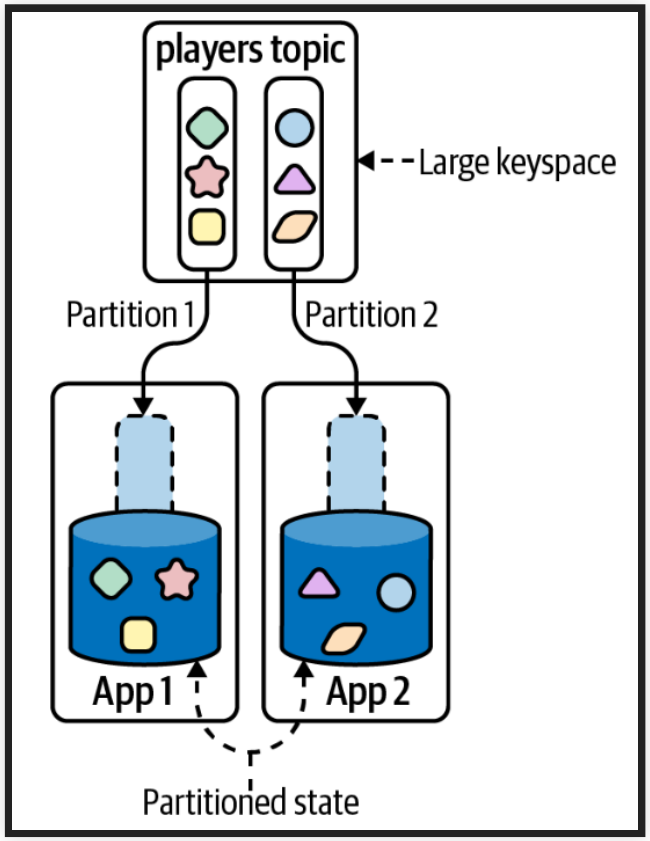
KStream, KTable, GlobalKTable

Kafka Stream bietet verschiedene Abstraktionen für Datenströme.

- `KStream`: Wird bei (zumeist Schlüssellosen Ereignissen) verwendet, um Verarbeitungsketten auf den Records zu realisieren
- `KTable` und `GlobalKTable`: Bietet einen Zugriff auf Datenströme in Form einer Tabelle. Die Abstraktionen unterscheiden sich in den Verteilungsstrategien bei mehreren Knoten.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

KTable



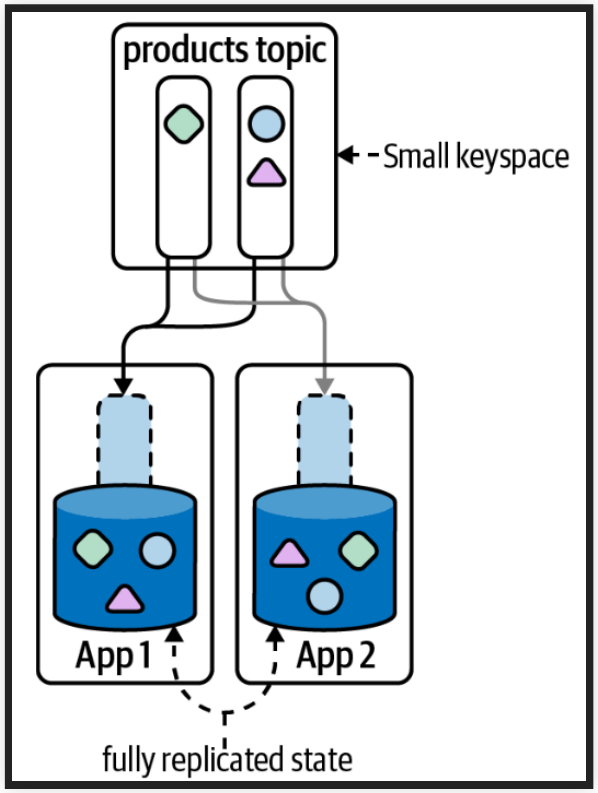
Ermöglicht Zugriff auf Daten in Kafka als Tabelle. Wird ein neues Ereignis empfangen, aktualisiert bzw. überschreibt dieses den letzten Zustand. Hierfür ist die Verwendung von Keys notwendig.

```
KTable<String, Player> players =
    builder.table("players");
```

Bei Skalierung gelten Partitionen und somit werden je Consumer nur Teile der Gesamten Menge an Einträgen in einer Tabelle verwendet.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

GlobalKTable



Wie die KTable ermöglicht auch die GlobalKTable einen Tabellen-basierten Zugriff. Dabei werden die Einträge jedoch nicht über Consumer verteilt, sondern alle Consumer haben den kompletten Datenbestand.

```
GlobalKTable<String, Product> products =
    builder.table("products");
```

Dies bietet sich nur für überschaubare Mengen an Daten an.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Für das Beispiel kommt ...

- ... für die ScoreEvents ein KStream zum Einsatz, da dieser ohne Keys ausgeliefert wird und kontinuierlich Ereignisse über neue Score-Änderungen enthält
- ... für die Player eine KTable zum Einsatz, da dieser Keys verwendet und aus vielen Spielern bestehen könnte, eine Verteilung über Consumer ist daher sinnvoll
- ... für die Products eine GlobalKTable zum Einsatz, da diese Keys verwenden und relativ wenig vorhanden sind

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

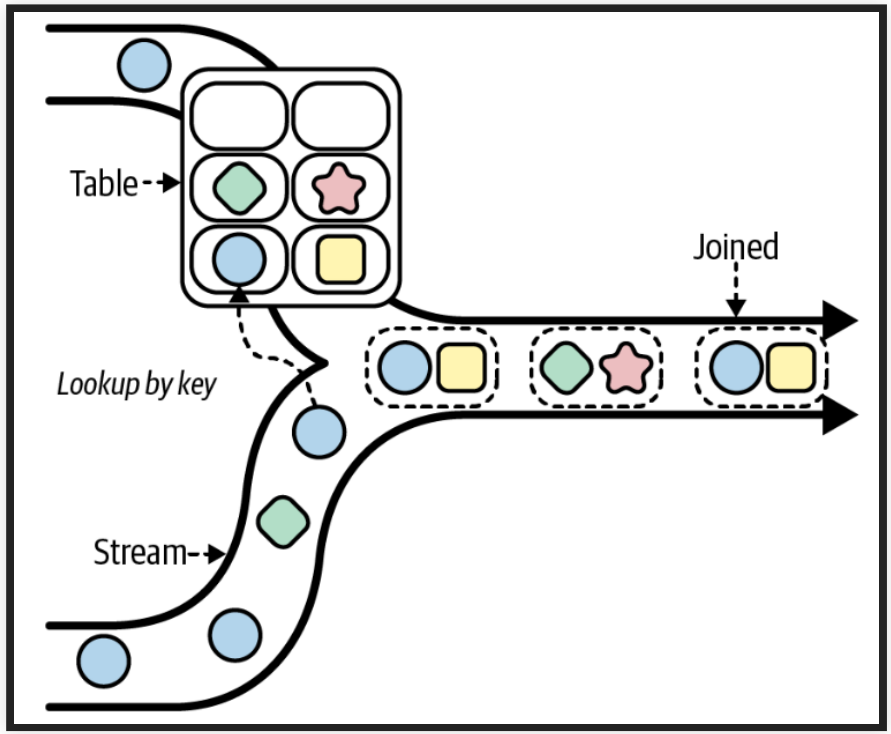
Beispiel: KStream, KTable und GlobalKTable

```
KStream<String, ScoreEvent> scoreEvents = builder
    .stream("score-events", Consumed.with(
        Serdes.String(), JsonSerdes.scoreEvent()));
KTable<String, Player> players = builder
    .table("players", Consumed.with(
        Serdes.String(), JsonSerdes.player()));
GlobalKTable<String, Product> products = builder
    .globalTable("products", Consumed.with(
        Serdes.String(), JsonSerdes.product()));
```

Für die Verarbeitung von JSON-Daten sind Serdes (Serializer und Deserializer) notwendig für Key und Value. Im Folgenden werden die Serdes nicht immer wieder angegeben.

Join

Ein Join stellt eine Möglichkeit dar, wie im relationalen Fall, verteile Daten zusammenzuführen.

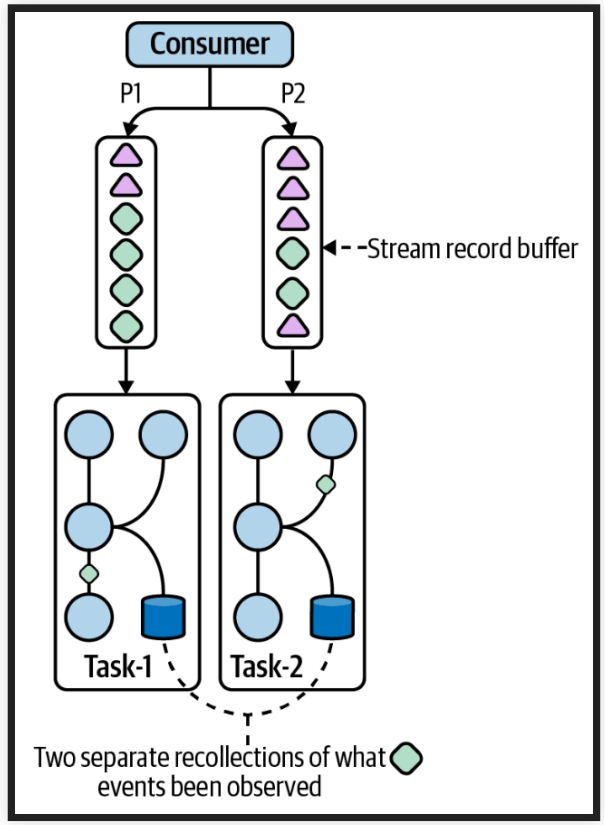


Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Co-Partitioning

Jede Partition wird genau einem Consumer, und damit einem Kafka-Stream-Task zugewiesen

Insbesondere beim Join und bei der Verwendung von KStream und KTable kann es aber dazu kommen, dass über die Partitionierung eine ungünstige Verteilung entsteht.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Um sicherzustellen, dass zusammenhängende Ereignisse an dieselbe Partition geleitet werden, müssen die folgenden Anforderungen an die gemeinsame Partitionierung erfüllt sein:

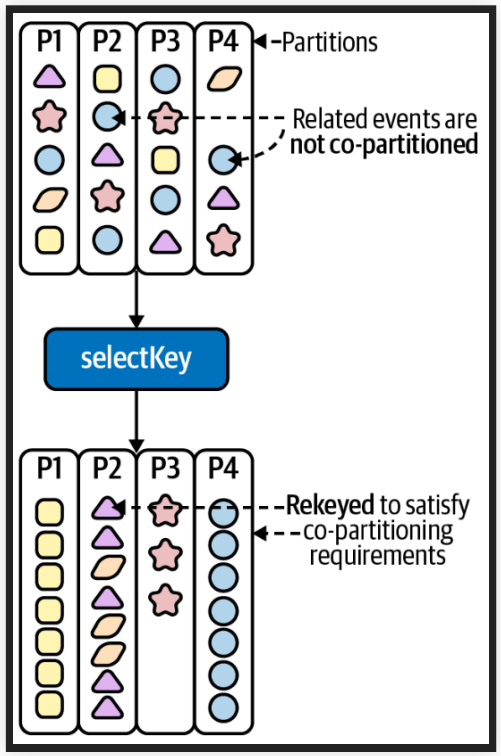
- Die Datensätze auf beiden Seiten müssen durch dasselbe Key-Feld und anhand dieses Schlüssels unter Verwendung derselben Partitionierungsstrategie partitioniert werden.
- Die Eingabethemen auf beiden Seiten der Verknüpfung müssen die gleiche Anzahl von Partitionen enthalten. Dies wird beim Starten überprüft. Wenn diese Bedingung nicht erfüllt ist, wird eine TopologyBuilderException ausgelöst.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Wann ist Co-Partitioning notwendig?

- `KStream` - `KStream`: Hier wird in Windows gearbeitet, Co-Partitioning ist **notwendig**
- `KTable` - `KTable`: In jeder Instanz ist die korrekte und gleiche Menge von Keys notwendig, Co-Partitioning ist **notwendig**
- `KStream` - `KTable`: In jeder Instanz ist die korrekte und gleiche Menge von Keys notwendig, Co-Partitioning ist **notwendig**
- `KStream` - `GlobalKTable`: Bei einem Join des Streams mit einer Tabelle die vollständig in jeder Instanz läuft, ist es **nicht notwendig** über Co-Partitioning nachzudenken

Rekeying



Mithilfe des `selectKey` kann der KStream von `ScoreEvent` in die richtige Partition organisiert werden.

```
KStream<String, ScoreEvent> scoreEvents =
    builder
        .stream("score-events", ...)
        .selectKey((key, value) ->
            value.getPlayerId().toString());
```

Mit dem Rekeying findet eine Repartitionierung statt, entsprechend ist eine `to` und `stream` Operation damit verbunden, es entsteht ergänzend eine auto-generated Queue.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Join-Beispiel: ScoreEvent join Player

```
KStream<String, ScoreEvent> scoreEvents = builder
    .stream("score-events", ...)
    .selectKey((key, value) -> value.getPlayerId().toString());
KTable<String, Player> players = builder
    .table("players", ...);
scoreEvents.join(players, (scoreEvent, player) -> ..., ...);
```

Der KStream wird über den Key mit den entsprechenden Player verbunden. Mit dem zweiten Argument wird das kombinierte Ergebnis erzeugt, dass sich aus diesem Join ergeben soll.

Für das Beispiel soll eine Instanz der Klasse ScoreWithPlayer erzeugt werden, welche ein Player- und ein ScoreEvent-Objekt referenziert. Die Klasse bietet einen Konstruktor `ScoreEvent(s: ScoreEvent, p: Player)`, daher kann in Kursform folgendes verwendet werden:

```
scoreEvents.join(players, ScoreWithPlayer::new, ...);
```

Hinweis: Mit dem Join muss der Serdes aktualisiert werden, da ScoreEvents.

Join-Beispiel: ScoreWithPlayer join Product

```
KStream<String, ScoreEvent> scoreEvents = builder
    .stream("score-events", ...)
    .selectKey((key, value) -> value.getPlayerId().toString());
KTable<String, Player> players = builder
    .table("players", ...);
GlobalKTable<String, Product> products = builder
    .globalTable("products", ...);
scoreEvents.join(players, ScoreWithPlayer::new, ...);
KStream<String, Enriched> withProducts = scoreWithPlayers.join(
    products,
    (leftKey, scoreWithPlayer) ->
        String.valueOf(scoreWithPlayer.getScoreEvent().getProductId()),
    Enriched::new);
```

Der Join findet hier über die Product-ID statt, welcher mittels Key-Mapper angegeben wird.

GroupBy und **GroupByKey**

Für die folgende Aggregation sollen Punkte je Spieler zusammengefasst werden, entsprechend müssen zuvor die Datenströme nach Spielern gruppiert werden.

```
withProducts.groupBy((key, value) -> ..., ...)
```

GroupBy ist wie das Rekeying mit `selectKey` eine repartitionierende Operation. Damit verbunden ist eine auto-generated Queue.

Beispiel

```
KStream<String, ScoreWithPlayer> scoreWithPlayers = scoreEvents.join(
    players, ScoreWithPlayer::new, ...);
KStream<String, Enriched> withProducts = scoreWithPlayers.join(
    products,
    (leftKey, scoreWithPlayer) ->
        String.valueOf(scoreWithPlayer.getScoreEvent().getProductId()),
    Enriched::new);
withProducts.groupBy((key, value) -> value.getProductId().toString(), ...);
```

GroupBy realisiert hier den Wechsel auf die Produkt-ID. Falls mit dem selben Key weitergearbeitet werden kann, wäre `groupByKey` die bessere Wahl, da hier kein Repartitioning zum Einsatz kommt.

Aggregation

Wie lassen sich mehrere Ereignisse zusammen betrachten?

Wie in anderen Ansätzen (Streaming-API oder Reactive Programming) bezieht sich Aggregation auf die Zusammenführung von Ereignissen wie über `count` oder `reduce`.

Notwendig sind hierfür Zustandsverwaltung und Initialisierung, welche im folgenden am Beispiel betrachtet werden.

Beispiel: Initialisierung und Aggregation

```
withProducts
    .groupBy((key, value) -> value.getProductId().toString(), ...)
    .aggregate(
        () -> new TreeSet<Enriched>(),
        (key, value, state) -> {
            state.add(value);
            return state;
        });
```

- Der `aggregate`-Operator besteht min. aus zwei Argumenten, einem Initializer und einen Aggregator
- Der Initializer soll einen initialen Zustand erzeugen, wenn noch kein Ereignis eingegangen ist
- Der Aggregator aktualisiert mit jedem neuen Ereignis den Zustand, hier wird dem Tree ein Element hinzugefügt

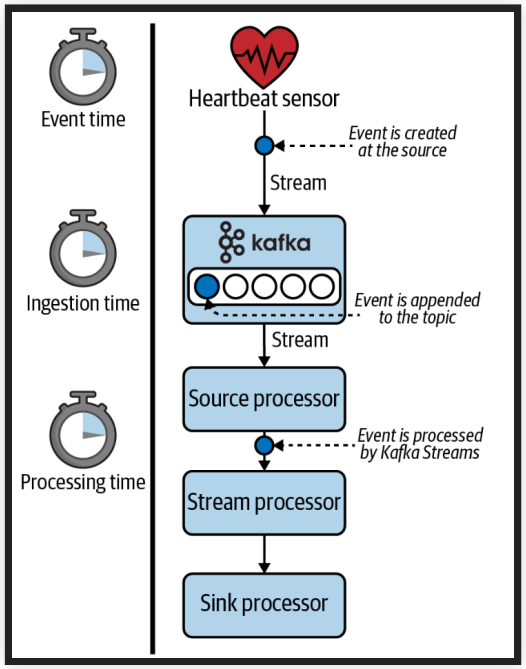
Auslagern des Leaderboards

```
public class Leaderboard {
    private TreeSet<Enriched> elements = new TreeSet<>();
    public Leaderboard update(Enriched value) {
        elements.add(value);
        if (elements.size() > 3) {
            elements.remove(elements.last());
        }
        return this;
    }
}
```

```
withProducts
    .groupBy((key, value) -> value.getProductId().toString(), ...)
    .aggregate(Leaderboard::new,
        (key, value, state) -> state.update(value));
```

WINDOWS

Mit Windows lassen sich Ereignisse in einem bestimmten Zeitfenster zusammenfassen



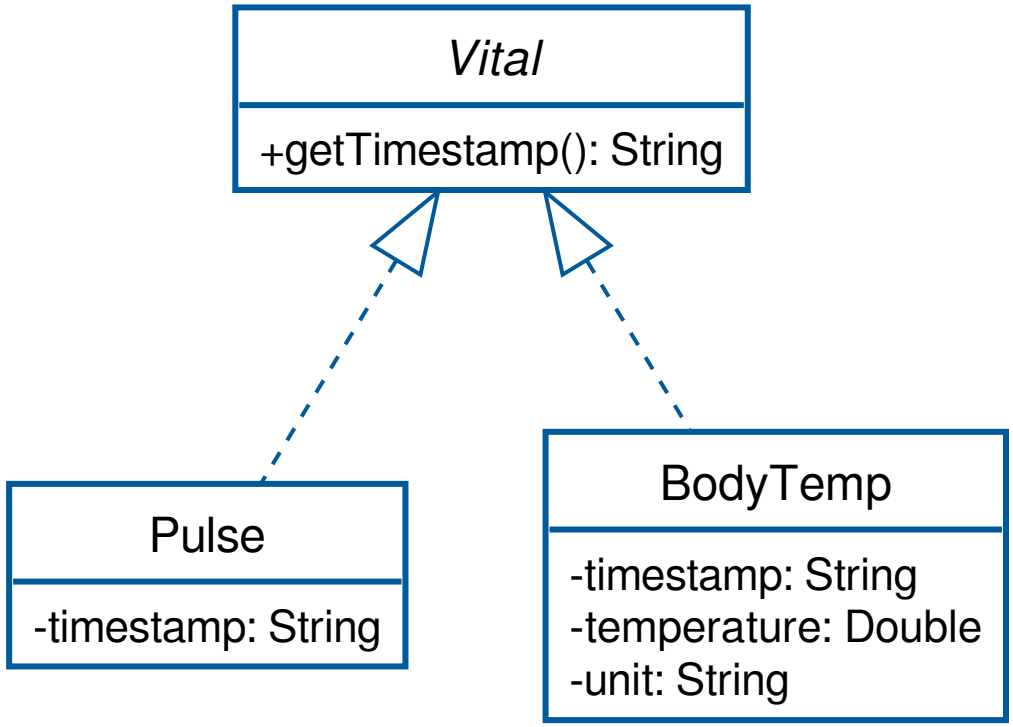
Folgende Beispiele orientieren sich an einem Herz-Sensor aus der Quelle, bei dem je Puls ein Ereignis erzeugt wird und regelmäßig die Temperatur gemessen wird.

Für die Betrachtung des Zeit-Kontext und entsprechende Fenster sind drei Zeitpunkte möglich: Ereignis-Erzeugung, -Eingang und -Verarbeitung.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Beispiel

In Anlehnung an die Quelle können Sie folgendes Beispiel verwenden. Es realisiert Pulse- und Temperatur-Überwachung.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Das Beispiel enthält eine `docker-compose.yml` für Apache Kafka und Zookeeper, welche mit `docker compose up -d` gestartet werden kann.

Nutzen Sie das bereitgestellte Projekt-Archiv apache-kafka-streams-pulse.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Verarbeitung von JSON-Daten bisher

```
KStream<String, Pulse> pulseEvents = builder
    .stream(Configurator.PULSE_EVENTS,
        Consumed.with(Serdes.String(), JsonSerdes.pulse()));
```

Bisher wurde über Serdes-Angaben konfiguriert, wie Serialisierung und Deserialisierung auf Datenströmen von Kafka Streams verwendet werden soll. Im Beispiel wird ein eigener JSON-Serdes verwendet.

```
KStream<String, Pulse> pulseEvents = builder
    .stream(Configurator.PULSE_EVENTS,
        Consumed.with(Serdes.String(), JsonSerdes.pulse())
            .withTimestampExtractor(new VitalTimestampExtractor()));
```

Über eine Ergänzung kann der Zeitbezug auf Basis einer Information in der Nachricht realisiert werden.

Timestamp Extractor

```
public class VitalTimestampExtractor implements TimestampExtractor {
    public long extract(ConsumerRecord<Object, Object> record,
            long partitionTime) {
        Vital measurement = (Vital) record.value();
        if (measurement != null && measurement.getTimestamp() != null) {
            String timestamp = measurement.getTimestamp();
            return Instant.parse(timestamp).toEpochMilli();
        }
        return partitionTime;
    }
}
```

Der Extractor muss dafür aus dem Ereignis entsprechende Information ziehen, die einen Rückschluss auf z.B. den Zeitpunkt der Erzeugung liefert.

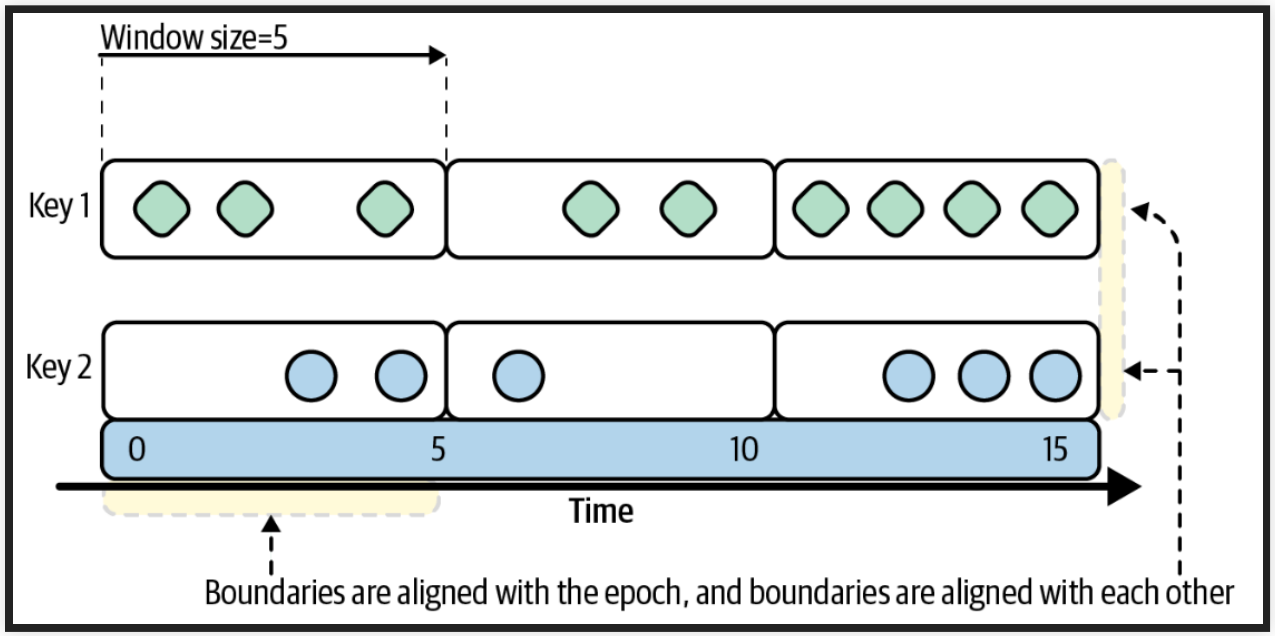
Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Tumbling Windows

```
TimeWindows window = TimeWindows.ofSizeWithNoGrace(
    Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey()
        .windowedBy(window)
        .count(Materialized.as(Configurator.PULSE_COUNTS));
```

Tumbling Windows sind Fenster fester Größe, die sich nie überlappen. Sie werden mit einer einzigen Eigenschaft definiert, der Fenstergröße (in Millisekunden), und haben vorhersehbare Zeitbereiche, da sie an der Epoche ausgerichtet sind.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

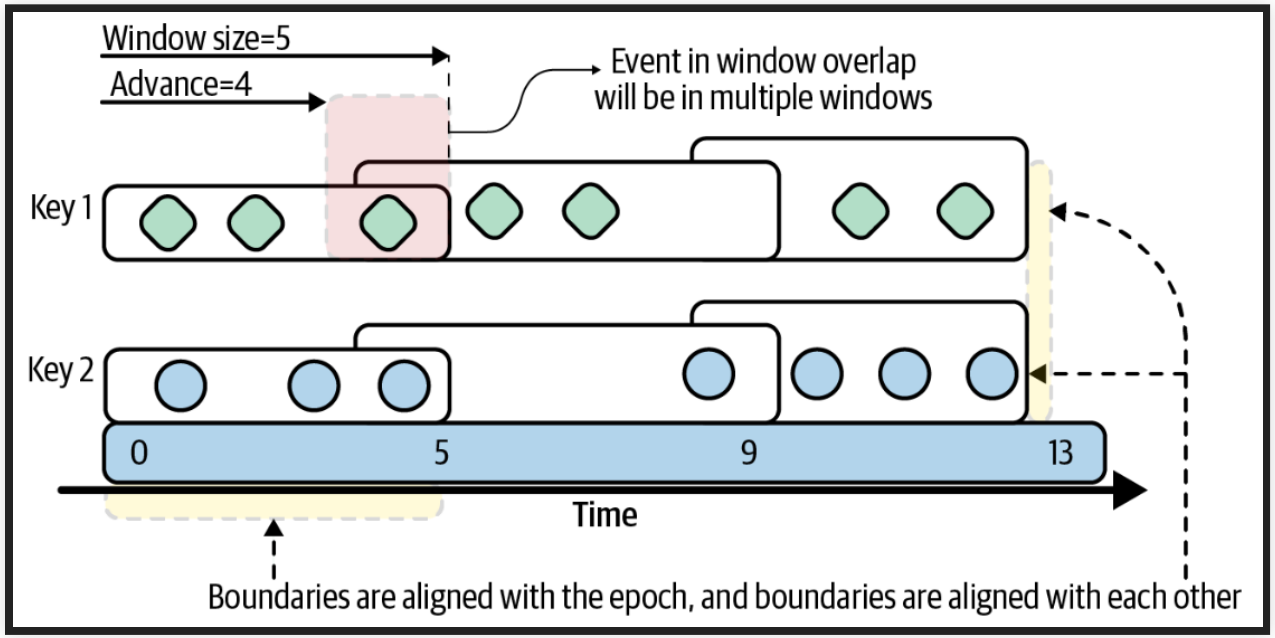
Overlapping Windows

```
TimeWindows window = TimeWindows.ofSizeWithNoGrace(
    Duration.ofSeconds(5)).advanceBy(Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey()
        .windowedBy(window)
        .count(Materialized.as(Configurator.PULSE_COUNTS));
```

Bei Overlapping Windows handelt es sich um Fenster fester Größe, die sich überlappen können.

Für die Konfiguration, müssen sowohl die Fenstergröße als auch das Vorrückungsintervall (wie weit sich das Fenster vorwärts bewegt) angegeben werden.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

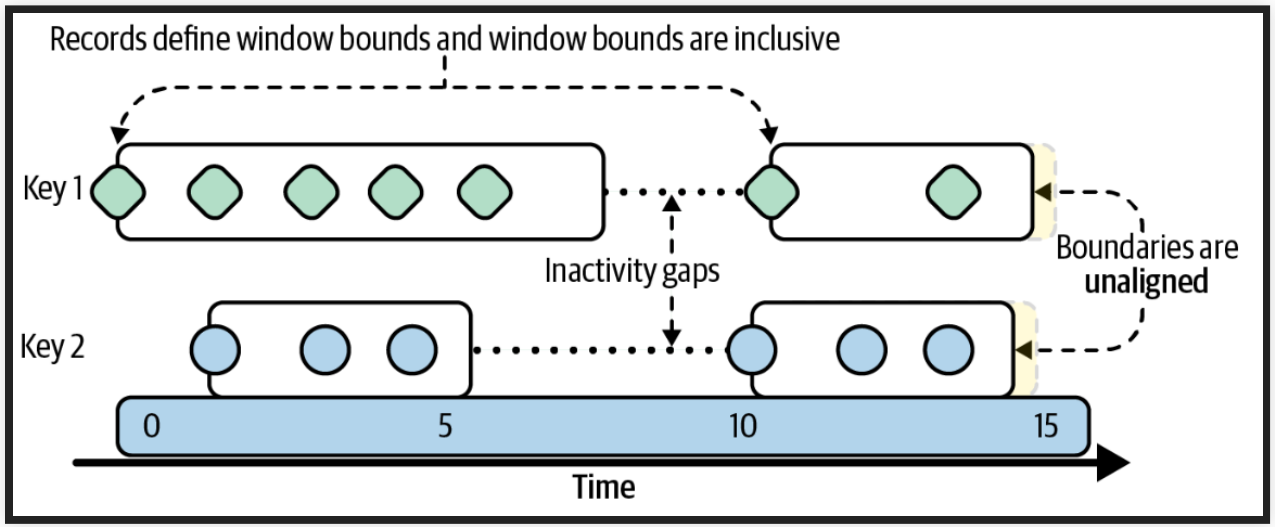
Session Windows

```
SessionWindows window = SessionWindows.with(Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey()
        .windowedBy(window)
        .count(Materialized.as(Configurator.PULSE_COUNTS));
```

Sitzungsfenster sind Fenster variabler Größe, die durch Aktivitätsperioden, gefolgt von Inaktivitätslücken, bestimmt werden.

Ein einziger Parameter, die Inaktivitätslücke, wird zur Definition eines Sitzungsfensters verwendet. Beträgt die Inaktivitätslücke fünf Sekunden, so wird jeder Datensatz, dessen Zeitstempel innerhalb von fünf Sekunden nach dem vorherigen Datensatz mit demselben Schlüssel liegt, in demselben Fenster zusammengeführt.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



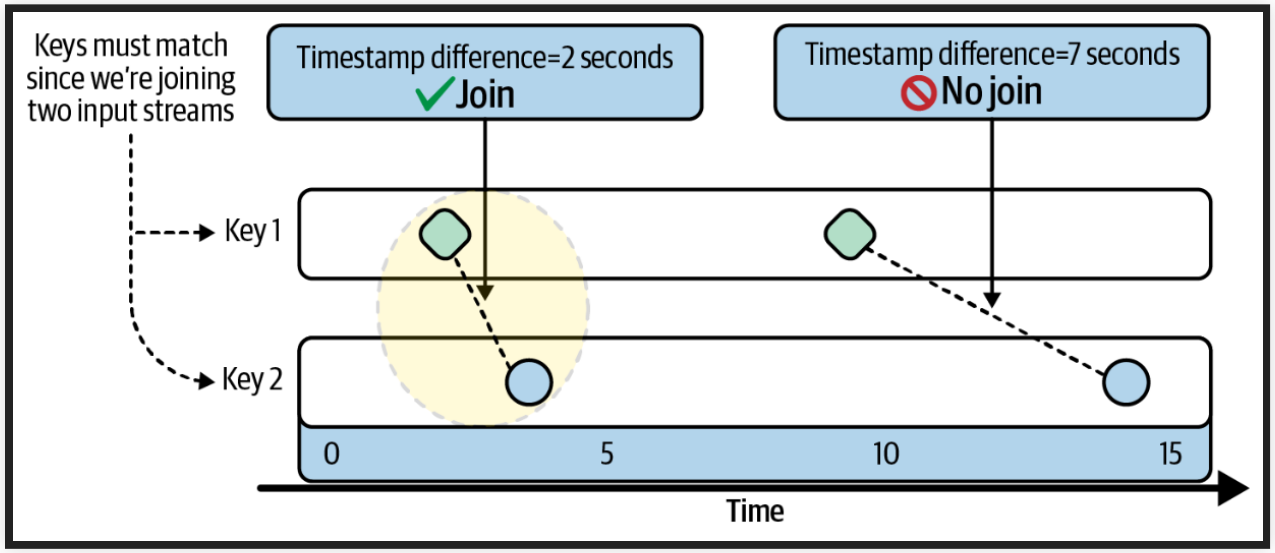
Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Sliding Join Windows

```
JoinWindows window = JoinWindows.of(Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey()
        .windowedBy(window)
        .count(Materialized.as(Configurator.PULSE_COUNTS));
```

Sind Fenster fester Größe, die für Joins verwendet und mit der Klasse JoinWindows erstellt werden. Zwei Datensätze fallen in dasselbe Fenster, wenn der Unterschied zwischen ihren Zeitstempeln kleiner oder gleich der Fenstergröße ist.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.



Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Grace

```
TimeWindows window = TimeWindows.ofSizeAndGrace(
    Duration.ofSeconds(60), Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey()
        .windowedBy(window)
        .count(Materialized.as(Configurator.PULSE_COUNTS));
```

Ereignisse können verzögert im Stream empfangen werden. Um Ereignissen die Möglichkeit zu geben in einem Zeitfenster aufgenommen zu werden, müssten ggf. auftretende Latenzen berücksichtigen werden.

Mittels Grace-Angaben lassen sich die nachgelagerten Wartezeiten, bis das Zeitfenster geschlossen, wird angeben. Hier wird 5 Sekunden länger gewartet.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Suppression

```
TimeWindows window = TimeWindows.ofSizeAndGrace(
    Duration.ofSeconds(60), Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey()
        .windowedBy(window)
        .count(Materialized.as(Configurator.PULSE_COUNTS));
        .suppress(Suppressed.untilWindowCloses(
            BufferConfig.unbounded().shutDownWhenFull()))
```

Verhindert, dass jedes Ereignis einzeln weitergegeben wird. Hier wird dafür gesorgt, dass am Ende eines Fensters das Gesamtergebnis bereitgestellt wird.

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Buffer Configuration

- BufferConfig.maxBytes(): Eingrenzung des Speichers durch Anzahl von Bytes
- BufferConfig.maxRecords(): Eingrenzung des Speichers durch Anzahl von Keys
- BufferConfig.unbounded(): Verwendung von Speicher solange wie möglich, wenn kein Heap-Speicher mehr verfügbar wird eine OutOfMemory-Exception ausgelöst

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

Buffer Full Strategies

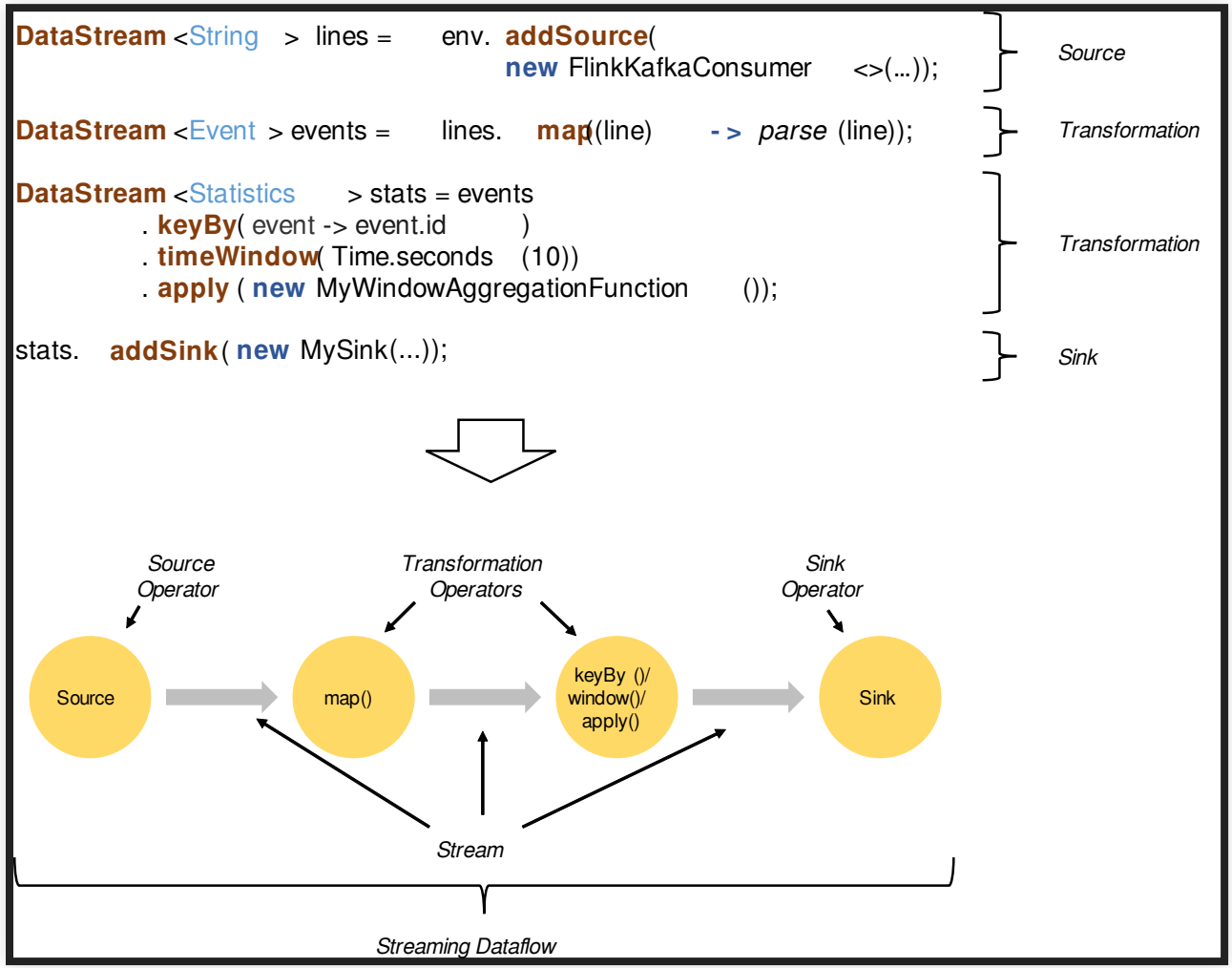
- shutDownWhenFull: Die Anwendung wird ordnungsgemäß beendet, wenn der Puffer voll ist; es werden keine Zwischenberechnungen im Fenster gezeigt
- emitEarlyWhenFull: Gibt die ältesten Ergebnisse aus, wenn der Puffer voll ist, anstatt die Anwendung abzuschalten; Bei dieser Strategie werden möglicherweise noch Zwischenberechnungen im Fenster gezeigt

Quelle: Mitch Seymour. Mastering Kafka Streams and ksqlDB - Building Real-Time Data Systems by Example. O'Reilly, 2021.

5.3 APACHE FLINK

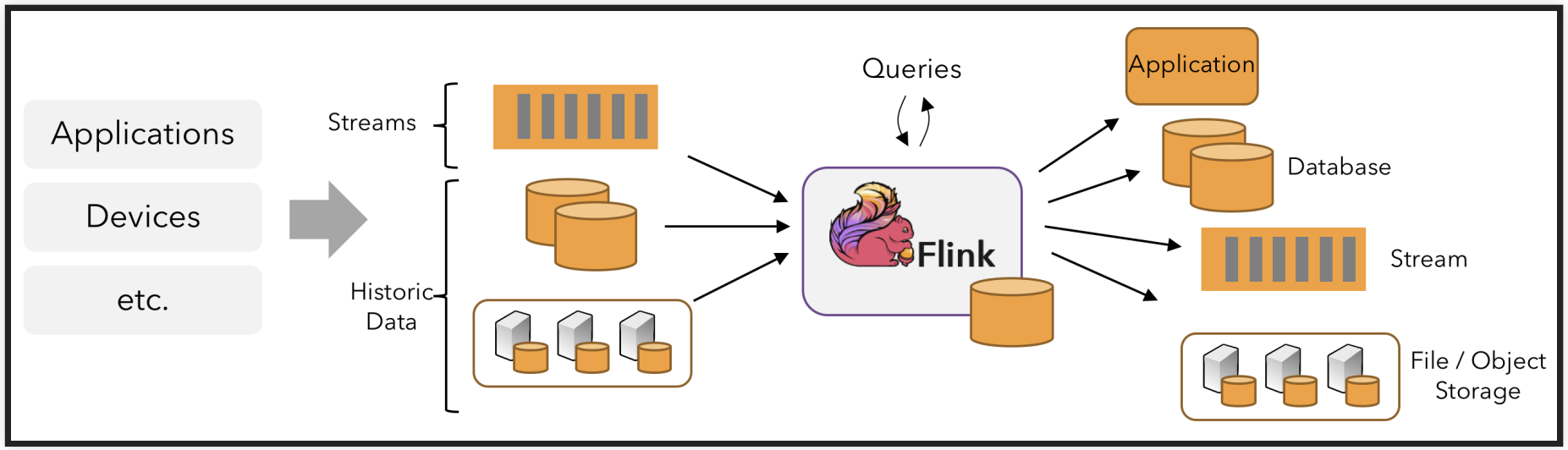
Neben Kafka Stream gibt es Platformen wie **Apache Flink**, Apache Spark und Apache Storm zur parallen Verarbeitung von Datenströmen.

In Apache Flink bestehen Anwendungen wie in Kafka Stream aus Datenströmen die durch (benutzerdefinierte) **Operatoren** transformiert werden können. Diese Datenflüsse bilden erneut gerichtete Graphen, die mit einer oder mehreren **Quellen** beginnen und in einer oder mehreren **Senken** enden.



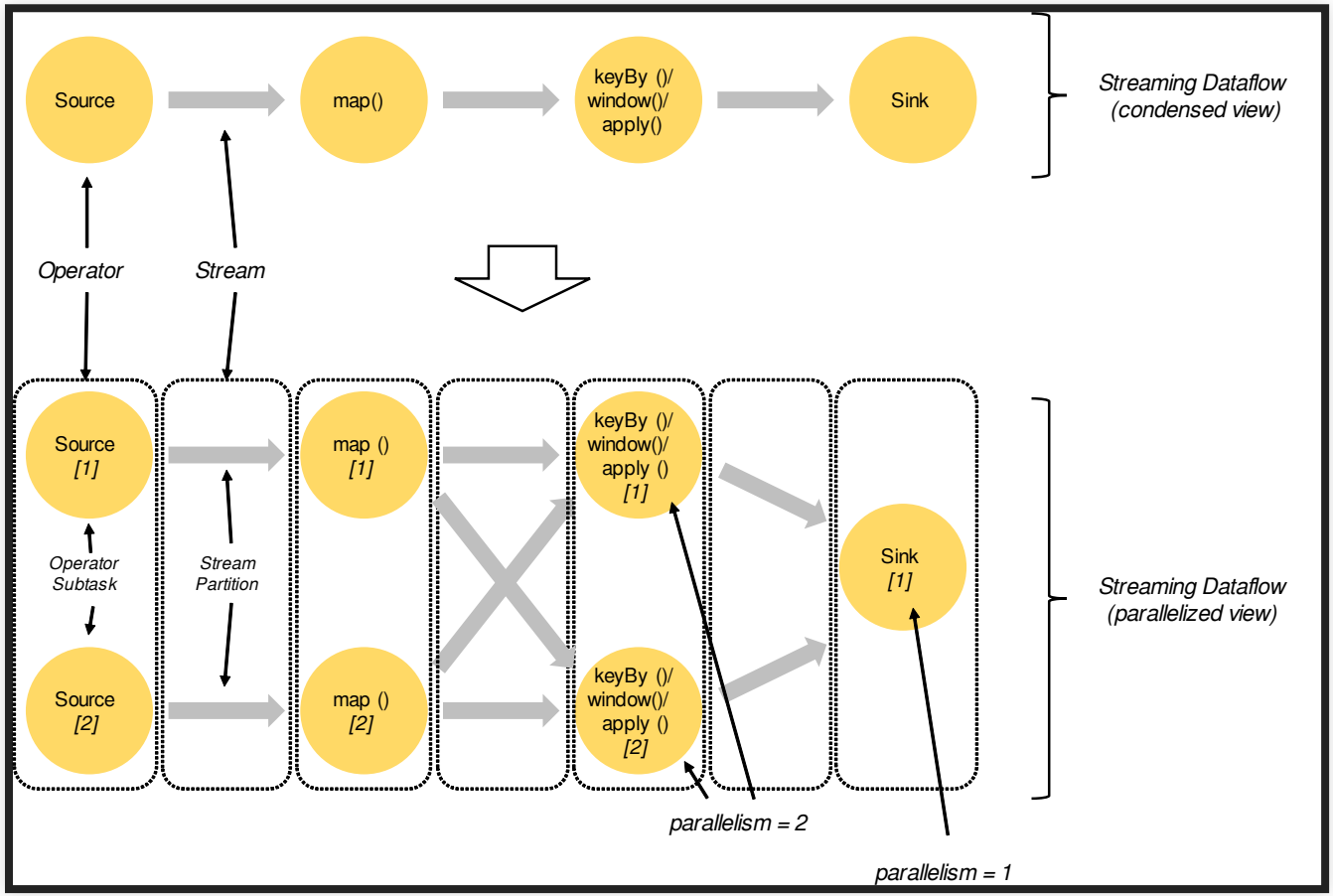
Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/overview/

Eine Anwendung in Apache Flink kann Echtzeitdaten aus Streaming-Quellen wie Message Queues oder verteilten Protokollen wie Apache Kafka nutzen. Apache Flink kann aber auch gebundene, historische Daten aus einer Vielzahl von anderen Datenquellen nutzen. In ähnlicher Weise können die von einer Flink-Anwendung erzeugten Ergebnisströme an eine Vielzahl von Systemen gesendet werden, die als Senken angeschlossen werden können.



Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/overview/

Parallele Datenstromverarbeitung



Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/overview/

- Programme in Flink sind parallel und verteilt
- Während der Ausführung hat ein Stream eine oder mehrere Stream-Partitionen, und jeder Operator hat eine oder mehrere Operator-Subtasks
- Die Operator-Subtasks sind unabhängig voneinander und werden in verschiedenen Threads und möglicherweise auf verschiedenen Maschinen oder Containern ausgeführt.
- Die Anzahl der Operator-Subtasks ist die Parallelität des jeweiligen Operators.
- Verschiedene Operatoren desselben Programms können einen unterschiedlichen Grad an Parallelität aufweisen.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/overview/

Streams können Daten zwischen zwei Operatoren in einem 1:1-Muster (oder Weiterleitung) oder in einem Redistributing-Pattern transportieren:

1:1-Streams (z. B. zwischen dem Source- und dem `map()` -Operator in der obigen Abbildung) behalten die Aufteilung und Reihenfolge der Elemente bei. Das bedeutet, dass Subtask (vgl. [1] in Abbildung) des map()-Operators die gleichen Elemente in der gleichen Reihenfolge sieht, wie sie von Subtask (vgl. [1] in Abbildung) des Source-Operators erzeugt wurden.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/overview/

Das **Redistributing** von Datenströmen (wie oben zwischen `map()` und `keyBy`/`window` sowie zwischen `keyBy`/`window` und Sink) ändert die Aufteilung der Datenströme. Jede Operator-Teilaufgabe sendet Daten an verschiedene Ziel-Teilaufgaben, abhängig von der gewählten Transformation.

Beispiele hierfür sind `keyBy()` (das die Partitionierung durch Hashing des Schlüssels vornimmt), `broadcast()` oder `rebalance()` (das die Partitionierung nach dem Zufallsprinzip vornimmt). Bei einem Umverteilungsaustausch wird die Reihenfolge der Elemente nur innerhalb jedes Paares von sendenden und empfangenden Teilaufgaben beibehalten (z. B. Subtask[1] von `map()` und Subtask[2] von `keyBy`/`window`). So führt beispielsweise die oben gezeigte Umverteilung zwischen den Operatoren keyBy/window und Sink zu einem Nichtdeterminismus hinsichtlich der Reihenfolge, in der die aggregierten Ergebnisse für verschiedene Schlüssel in der Senke ankommen.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/overview/

HELLO-WORLD EXAMPLE

Folgendes Beispiel zeigt in Anlehnung an Kafka Streams die Verarbeitung eines Topics und die Veröffentlichung von Ergebnissen

```
public class HelloWorldJob {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecu
        // ...
        env.execute("Hello World");
    }
}
```

Grundlage ist die `StreamExecutionEnvironment`, welche es erlaubt Datenstromverarbeitungen zu konfigurieren.

Einfaches Beispiel

```
public class HelloWorldJob {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecu

        env.fromCollection(Arrays.asList("abc", "test", "unknown"))
                .map((value) -> "Hello, " + value)
                .print();

        env.execute("Hello World");
    }
}
```

- Mittels `fromCollection` kann eine beliebige Collection als Source angeschlossen werden
- Über Operatoren wie `map` werden Ereignisse verarbeitet
- Mittels `print` werden Ergebnisse in der Konsole platziert

Erweitertes Beispiel

```
public class HelloWorldJob {
    public static void main(String[] args) throws Exception {
        StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecu

        env.fromSource(...)
            .map((value) -> "Hello, " + value)
            .sinkTo(sink);

        env.execute("Hello World");
    }
}
```

- Mittels `fromSource` kann eine beliebige Quelle angeschlossen werden
- Über Operatoren wie `map` werden Ereignisse verarbeitet
- Mittels `sinkTo` werden Ergebnisse in einer Senke platziert

```java
public class HelloWorldJob {
    public static void main(String[] args) throws Exception {
        // ...
        KafkaSource<String> source = KafkaSource.<String>builder()
                .setBootstrapServers("localhost:9090")
                .setTopics("hello-world")
                .setGroupId("flink-foo")
                .setStartingOffsets(OffsetsInitializer.earliest())
                .setValueOnlyDeserializer(new SimpleStringSchema()).build();

        DataStream<String> text = env
            .fromSource(source, WatermarkStrategy.noWatermarks(),
                "hello-world") // ...
    }
}
```

- Eine Kafka-Quelle kann auf unterschiedlichen Wegen konfiguriert werden, eine Option ist die `KafkaSource`
- Notwendige Informationen sind wie bisher GroupID, Topics und Bootstrap Servers
- Anschließend kann die Quelle für die Konfiguration der Verarbeitungspipeline verwendet werden, hier ist eine WatermarkStrategy notwendig und ein Name für die Quelle

Dieses und folgende Beispiel finden sich in folgendem Archiv

Das Beispiel enthält ein Java-Projekt, welches direkt genutzt und gestartet werden kann. Für Beispiele mit Kafka muss Kafka zzgl. Zookeeper verfügbar sein (`docker compose up -d kafka zookeeper`). Ergänzend ist ein Flink Cluster konfiguiert.

Nutzen Sie das bereitgestellte Projekt-Archiv apache-flink-hello-world.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

DATENFORMATE

Mit den DataStream-APIs von Flink können verschiedene Formate verarbeitet werden. Flink's eigener Serialisierer wird verwendet für:

- Grundtypen, d.h. String, Long, Integer, Boolean, Array
- zusammengesetzte Typen: Tuples, POJOs, und Scala Fallklassen

... und für andere Typen greift Flink auf Kryo zurück. Es ist auch möglich, andere Serialisierer mit Flink zu verwenden. Insbesondere Avro wird gut unterstützt.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Tuples

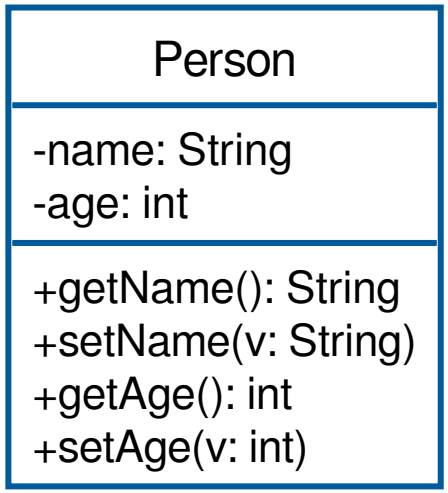
```
Tuple2<String, Integer> person = Tuple2.of("Fred", 35);

// zero based index!
String name = person.f0;
Integer age = person.f1;
```

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

POJOs

Flink erkennt einen Datentyp als POJO-Typ (und erlaubt die "by-name"-Feldreferenzierung), wenn die folgenden Bedingungen erfüllt sind:



Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Dabei muss gelten:

- Die Klasse ist public und standalone (keine nicht-statische innere Klasse)
- Die Klasse hat einen öffentlichen Konstruktor ohne Argumente (Default-Konstruktor)
- Alle nicht-statischen, nicht-transienten Felder in der Klasse (und allen Oberklassen) sind entweder öffentlich (und nicht-final) oder haben öffentliche Getter- und Setter-Methoden

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Beispiel mit POJO

```
public class Example {
    public static void main(String[] args) throws Exception {
        final StreamExecutionEnvironment env =
                StreamExecutionEnvironment.getExecutionEnvironment();

        DataStream<Person> flintstones = env.fromElements(
                new Person("Fred", 35),
                new Person("Wilma", 35),
                new Person("Pebbles", 2));

        flintstones
            .filter((Person person) -> person.age >= 18)
            .print();

        env.execute();
    }
}
```

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

JSON

Zur Serialisierung / Deserialisierung von sonstigen Formaten sind eigene Implementierungen möglich. Als Beispiel folgend eine Implementierung zur Verwendung von Jackson mit Apache Flink:

```
public class HelloWorld {
    private String name;
    public String getName() {}
    public void setName(String name) {}
}
```

```
public class HelloWorldSchema extends AbstractDeserializationSchema<HelloWor
    private static final ObjectMapper OBJECT_MAPPER = new ObjectMapper();

    public HelloWorld deserialize(byte[] message) throws IOException {
        return OBJECT_MAPPER.readValue(message, HelloWorld.class);
    }
}
```

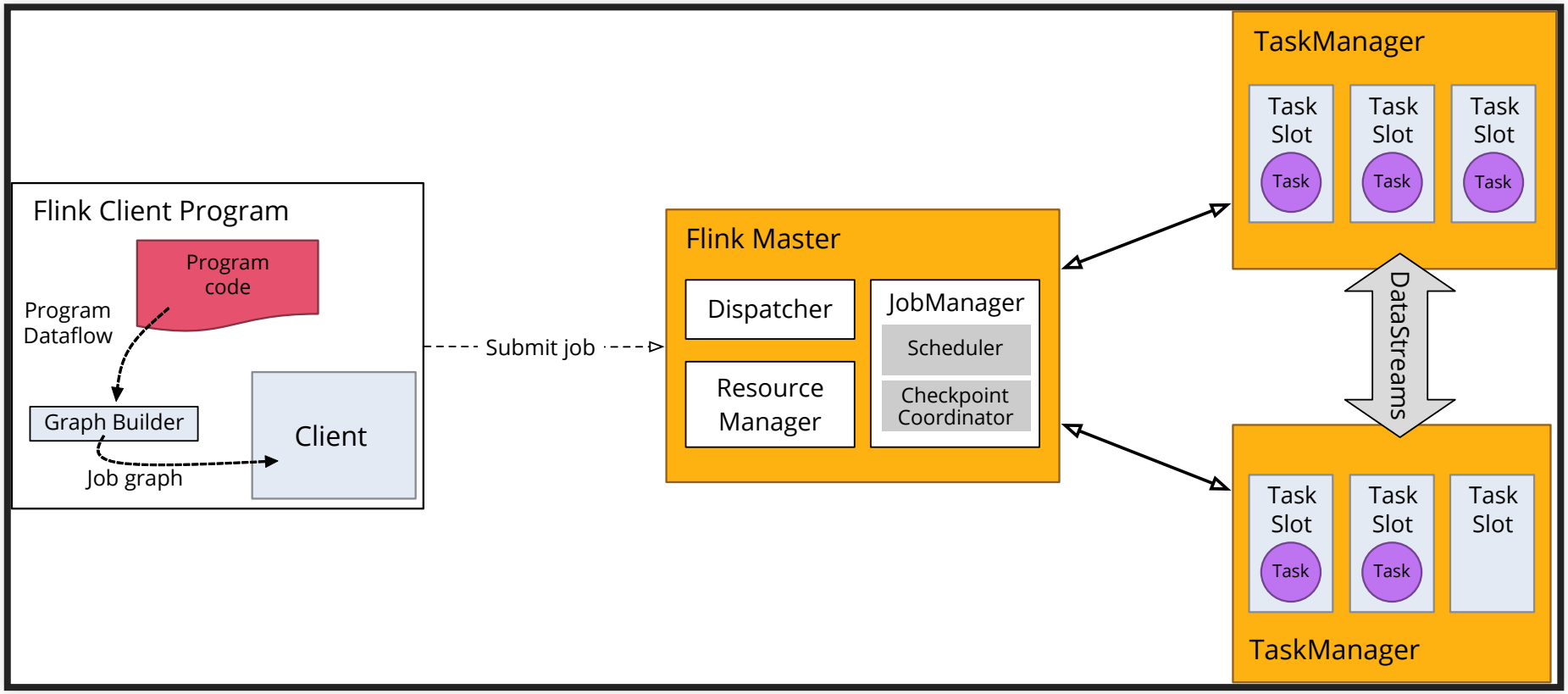
Anpassung der Konfiguration der KafkaSource

```
KafkaSource<HelloWorld> source = KafkaSource.<HelloWorld>builder()
    .setBootstrapServers(bootstrapServers)
    .setTopics("hello-world")
    .setGroupId("flink-foo")
    .setStartingOffsets(OffsetsInitializer.earliest())
    .setValueOnlyDeserializer(new HelloWorldSchema()).build();
```

Die Instanz des DeserializationSchema wird in der Source-Konfiguration bereitgestellt.

STREAM EXECUTION ENVIRONMENT

Jede Flink-Anwendung benötigt eine Ausführungsumgebung, in den Beispielen war dies `env` über die Klasse `StreamExecutionEnvironment`.



Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

- Die DataStream-API-Aufrufe erstellen einen Auftragsgraphen, der an die `StreamExecutionEnvironment` angehängt wird
- Wenn `env.execute()` aufgerufen wird, wird dieser Graph verpackt und an den JobManager gesendet, der den Auftrag parallelisiert und Teile davon an die Task-Manager zur Ausführung verteilt
- Jedes parallele Teilstück des Auftrags wird in einem Aufgabenslot ausgeführt.

Hinweis: Beachten Sie, dass die Anwendung nicht ausgeführt wird, wenn execute() nicht aufrufen wird.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

STATELESS TRANSFORMATIONS

Zustandslose Transformationen speichern keinen Zustand in der Ausführungsumgebung bei der Transformation. Typischerweise betrifft dies Operatoren wie `flatMap` und `map`.

```
env.fromSource(...)
    .map((value) -> "Hello, " + value)
    .sinkTo(sink);
```

FlatMap-Beispiel

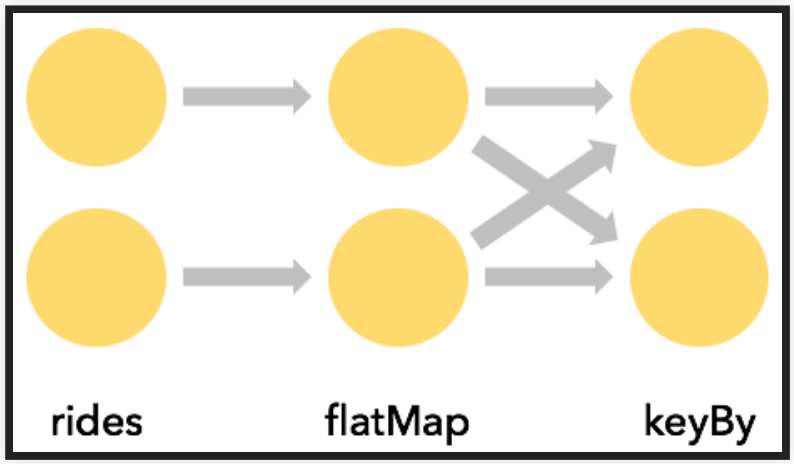
```
env.fromSource(...)
        .<String>flatMap((value, out) ->
            Stream.of(value.split(",")).forEach(out::collect))
        .returns(Types.STRING)
        .map((value) -> value.trim())
        .map((value) -> "Hello, " + value)
        .sinkTo(sink);
```

Im Beispiel wird ein Split auf dem String im Datenstrom durchgeführt, die resultierenden Elemente werden einzeln weitergegeben. Weitergegeben werden diese über den Parameter out, welcher eine Methode `collect` anbietet.

Für Lambda-Ausdrücke ist in Flink die Angabe des Types notwendig, alternativ können anonyme innere Klassen verwendet werden.

KEYED STREAMS

Ein Datenstrom kann anhand eines seiner Attribute partitioniert werden, so dass alle Ereignisse mit demselben Wert dieses Attributs in Gruppen zusammengefasst werden.



Zum Beispiel eine Suche nach längsten Taxifahrten, die einer bestimmten Grid-Zelle beginnen. Der keyBy-Operator würde eine Art Gruppierung (vgl. SQL) ermöglichen.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Der Operator `keyBy` teilt einen Datenstrom logisch in disjunkte Partitionen auf.

```
env.fromSource(...)
    .<String>flatMap((value, out) -> Stream.of(value.split(",")).forEach(out
    .returns(Types.STRING)
    .map((value) -> value.trim())
    .keyBy((value) -> value)
    .map((value) -> "Hello, " + value)
    .sinkTo(sink);
```

Alle Datensätze mit dem gleichen Schlüssel werden der gleichen Partition zugewiesen. Intern ist keyBy() mit Hash-Partitionierung implementiert.

STATEFUL TRANSFORMATIONS

Anwendungen können Zustände verwenden, ohne dass Flink in die Verwaltung dieser Zustände involviert ist - aber Flink bietet Funktionen für den von ihm verwalteten Zustand.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Zustandsverwaltung in Flink:

- **lokal**: Der Flink-Status wird lokal auf der Maschine gehalten, die ihn verarbeitet, und kann mit Speichergeschwindigkeit abgerufen werden.
- **dauerhaft**: Der Flink-Status ist fehlertolerant, d.h. er wird in regelmäßigen Abständen automatisch überprüft und bei einem Ausfall wiederhergestellt.
- **vertikal skalierbar**: Der Flink-Status kann in eingebetteten RocksDB-Instanzen gehalten werden, die durch Hinzufügen weiterer lokaler Festplatten skalierbar sind.
- **Horizontal skalierbar**: Der Flink-Status wird umverteilt, wenn Ihr Cluster wächst und schrumpft.
- **abfragbar**: Der Flink-Status kann von außen über die Queryable State API abgefragt werden. In diesem Abschnitt erfahren Sie, wie Sie mit den APIs von Flink arbeiten, die den Keyed State verwalten.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Beispiel: Jeder Name nur einmal

```
env.fromSource(source, WatermarkStrategy.noWatermarks(), "hello-world")
    .<String>flatMap((value, out) -> Stream.of(value.split(",")).forEach(out
    .returns(Types.STRING)
    .map((value) -> value.trim())
    .keyBy(value -> value)
    .flatMap(new Deduplicator())
    .map((value) -> "Hello, " + value)
    .sinkTo(sink);
```

Im Beispiel wird mittels `keyBy` partitioniert, und anschließend ein `flatMap`-Operator genutzt. Der `flatMap`-Operator speichert, ob der Name schon einmal empfangen wurde.

```
public static class Deduplicator
        extends RichFlatMapFunction<String, String> {
    private ValueState<Boolean> keyHasBeenSeen;
    public void open(Configuration conf) {
        ValueStateDescriptor<Boolean> desc =
            new ValueStateDescriptor<>("keyHasBeenSeen", Types.BOOLEAN);
        keyHasBeenSeen = getRuntimeContext().getState(desc);
    }
    public void flatMap(String event, Collector<String> out)
            throws Exception {
        if (keyHasBeenSeen.value() == null) {
            out.collect(event);
            keyHasBeenSeen.update(true);
        }
    }
}
```

Mit jeder Verarbeitung von `flatMap` wird erst die `open`-Methode aufgerufen.

Anschließend wird die eigentliche `flatMap`-Methode aufgerufen, diese enthält aktuell eine Referenz auf den zugehörigen Zustand der Partition.

WATERMARKS

Gegeben sei folgendes Beispiel einer Ereignisreihe mit Zeitstempeln:

```
--- 4 2 7 11 9 15 12 13 17 14 21 24 22 19 23 -->
                                        arrivial time
```

Die angezeigten Zahlen sind Zeitstempel, die angeben, wann diese Ereignisse eingetreten sind. Das erste Ereignis, das eintrifft, ist zur Zeit 4, gefolgt von einem Ereignis, das früher, zur Zeit 2, eingetroffen ist, und so weiter:

Angenommen es ist ein Stream-Sorter zu erstellen, welches jedes Ereignis aus einem Datenstrom verarbeitet, sobald es eintrifft. Der Sorter soll einen neuen Datenstrom ausgeben, der dieselben Ereignisse enthält, die aber nach ihren Zeitstempeln geordnet sind.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Beobachtungen:

- Das erste Element, das Ihr Stream Sorter sieht, ist die 4, es ist aber unbekannt ob es das erste eingetretene Ereignis. Eine Pufferung wäre notwendig.
- Wie lange soll gepuffert werden? Was wenn nie eine 1 kommt?
- Notwendig ist eine Art Richtlinie, die festlegt, wann man bei einem bestimmten Ereignis mit Zeitstempel nicht mehr auf das Eintreffen früherer Ereignisse warten soll.

In Flink werden hierfür Wasserzeichen verwendet - sie definieren, wann das Warten auf frühere Ereignisse beendet werden soll.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Die Zeitverarbeitung von Ereignissen in Flink hängt von Wasserzeichen-Generatoren ab, die spezielle zeitgestempelte Elemente in den Stream einfügen, die Wasserzeichen genannt werden. Ein Wasserzeichen für den Zeitpunkt t ist eine Behauptung, dass der Stream (wahrscheinlich) bis zum Zeitpunkt t vollständig ist.

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Working with Watermarks

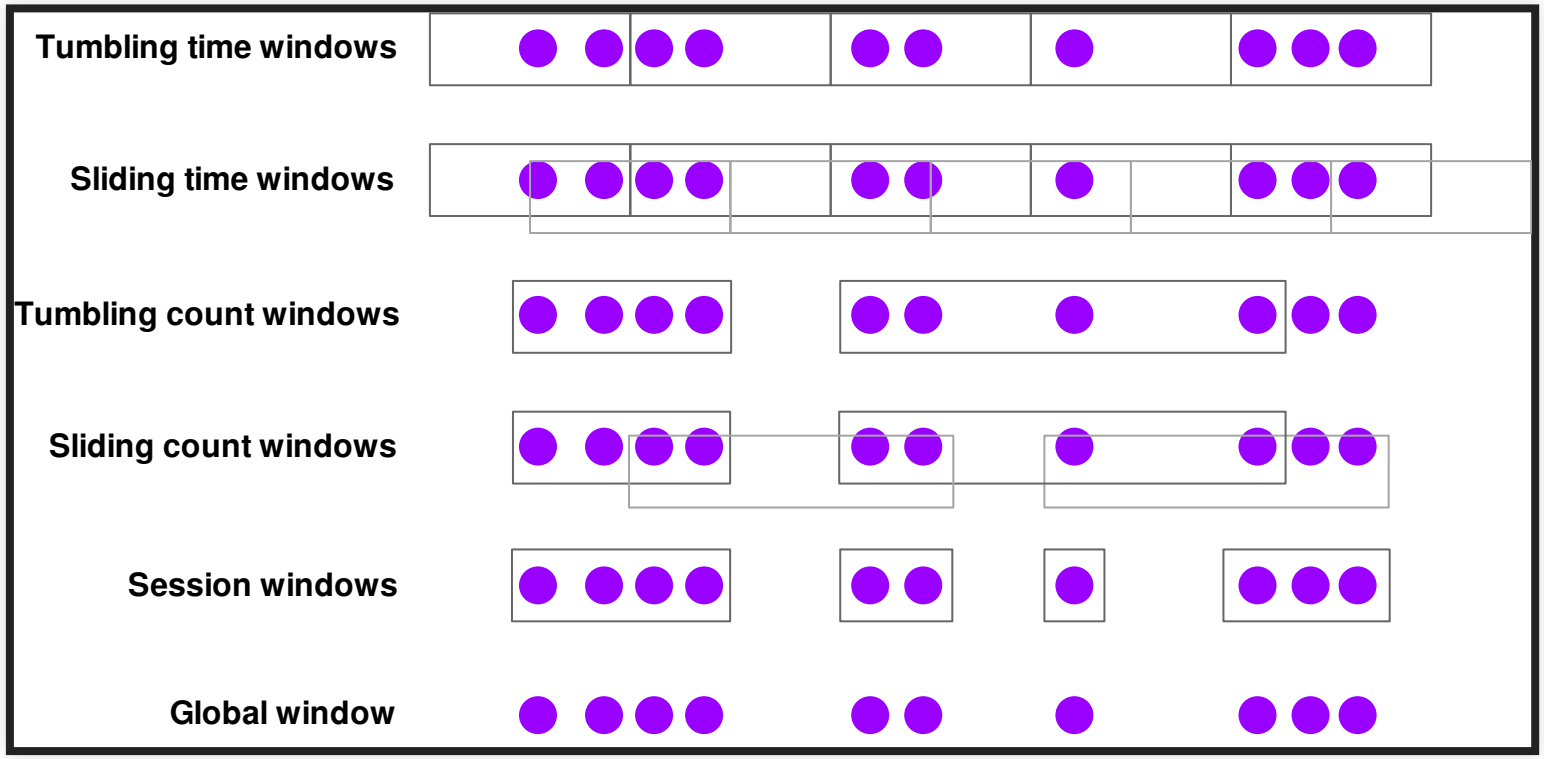
```
WatermarkStrategy<Event> strategy = WatermarkStrategy
        .<Event>forBoundedOutOfOrderness(Duration.ofSeconds(20))
        .withTimestampAssigner((event, timestamp) -> event.timestamp);

DataStream<Event> withTimestampsAndWatermarks =
    stream.assignTimestampsAndWatermarks(strategy);
```

Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

WINDOWS

Flink verfügt über mehrere eingebaute Typen von Fensterzuweisern



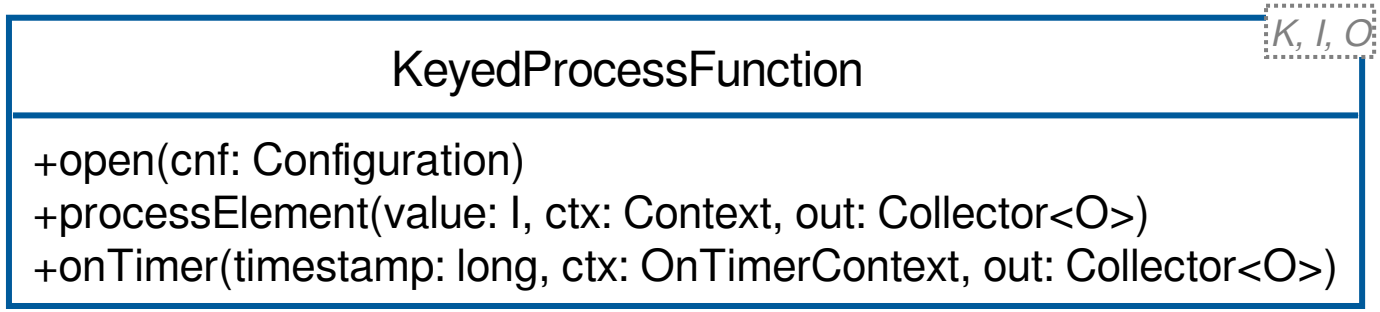
Quelle: https://nightlies.apache.org/flink/flink-docs-release-1.17/docs/learn-flink/datastream_api/

Beispiel zu Tumbling Windows

```
env.fromCollection(asList)
        .assignTimestampsAndWatermarks(strategy)
        .keyBy(value -> 1)
        .window(TumblingEventTimeWindows.of(Time.seconds(5)))
        .aggregate(new CountAggregator())
        .print();
```

PROCESS FUNCTIONS

Eine Process Function kombiniert die Ereignisverarbeitung mit Zeitgebern und Zuständen. Dies ist die Grundlage für die Erstellung ereignisgesteuerter Anwendungen mit Flink. Sie ist einer RichFlatMapFunction sehr ähnlich, allerdings mit dem Zusatz des Zeitbezugs.



Nutzen Sie das bereitgestellte Projekt-Archiv apache-flink-fraud.zip, entpacken Sie dieses und öffnen Sie den Ordner in Visual Studio Code.

5.4 VERGLEICH

Für Datenstromverarbeitung gibt es verschiedene Frameworks, die bekanntesten sind Apache Storm, Spark, Flink sowie Kafka Stream. Während alle sich für die Verarbeitung von Datenströmen eignen, können Sie hinsichtlich verschiedener Aspekte unterschieden werden:

- Delivery Guarantees
- Fehlertoleranz
- Zustandsverwaltung
- Leistung
- Erweiterte Funktionen

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Delivery Guarantees

Dies bedeutet, dass ein bestimmter eingehender Datensatz in einer Streaming-Engine auf jeden Fall verarbeitet wird. Es kann entweder Atleast-once (wird mindestens einmal verarbeitet, auch im Falle von Fehlern), Atmost-once (wird im Falle von Fehlern nicht verarbeitet) oder Exactly-once (wird genau einmal verarbeitet, auch im Falle von Fehlern) sein. Exactly-once ist natürlich wünschenswert, aber in verteilten Systemen schwer zu erreichen und geht mit Kompromissen bei der Leistung einher.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Fehlertoleranz

Im Falle von Ausfällen wie Knotenausfällen, Netzwerkausfällen usw. sollte das System in der Lage sein, sich zu erholen und die Verarbeitung an dem Punkt fortzusetzen, an dem es aufgehört hat. Dies wird dadurch erreicht, dass der Zustand des Streaming von Zeit zu Zeit in einem persistenten Speicher abgelegt wird. z.B. werden Kafka-Offsets in Zookeeper abgelegt, nachdem ein Datensatz von Kafka abgerufen und verarbeitet wurde.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Zustandsverwaltung

Im Falle von zustandsabhängigen Verarbeitungsanforderungen, bei denen wir einen gewissen Zustand beibehalten müssen (z.B. Zählung jedes einzelnen Wortes in Datensätzen), sollte das Framework in der Lage sein, einen Mechanismus zur Erhaltung und Aktualisierung von Zustandsinformationen bereitzustellen.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Leistung

Dazu gehören Latenz (wie schnell ein Datensatz verarbeitet werden kann), Durchsatz (verarbeitete Datensätze/Sekunde) und Skalierbarkeit. Die Latenzzeit sollte so gering wie möglich sein, während der Durchsatz so hoch wie möglich sein sollte. Es ist schwierig, beides gleichzeitig zu erreichen.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Erweiterte Funktionen: Ereigniszeitverarbeitung, Wasserzeichen, Fensterung

Diese Funktionen werden benötigt, wenn die Anforderungen an die Stream-Verarbeitung komplex sind. Zum Beispiel die Verarbeitung von Datensätzen basierend auf der Zeit, zu der sie an der Quelle erzeugt wurden (Ereigniszeitverarbeitung).

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Reifegrad

Wie gut haben sich Framework bereits bewährt und wurden von Unternehmen in großem Maßstab getestet. Es erhöht die Wahrscheinlichkeit, dass gute Unterstützung von der Community und Hilfe auf Stackoverflow geboten wird.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Apache Storm

Storm ist das Hadoop der Streaming-Welt. Es ist das älteste Open-Source-Streaming-Framework und eines der ausgereiftesten und zuverlässigsten.

Vorteile

- Sehr niedrige Latenz, echtes Streaming, ausgereift und hoher Durchsatz
- Hervorragend geeignet für nicht komplizierte Streaming-Anwendungsfälle

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Nachteile

- Keine Zustandsverwaltung
- Keine erweiterten Funktionen wie Ereigniszeitverarbeitung, Aggregation, Windowing, Sitzungen, Wasserzeichen usw.
- Mindestens-einmal-Garantie

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Apache Spark Streaming

Spark hat sich als echter Nachfolger von Hadoop im Bereich der Batch-Verarbeitung herauskristallisiert und ist das erste Framework, das die Lambda-Architektur vollständig unterstützt (wo sowohl Batch als auch Streaming implementiert sind; Batch für Korrektheit, Streaming für Geschwindigkeit).

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Vorteile

- Unterstützt Lambda-Architektur, wird kostenlos mit Spark geliefert
- Hoher Durchsatz, gut für viele Anwendungsfälle, bei denen eine geringe Latenzzeit nicht erforderlich ist
- Standardmäßige Fehlertoleranz aufgrund der Mikro-Batch-Natur
- Einfach zu verwendende APIs auf höherer Ebene
- Große Community und aggressive Verbesserungen
- Exactly Once

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Nachteile

- Kein echtes Streaming, nicht geeignet für niedrige Latenzanforderungen
- Zu viele Parameter zum Abstimmen. Schwer, es richtig hinzubekommen. Ich habe einen Beitrag über meine persönlichen Erfahrungen beim Tuning von Spark Streaming geschrieben
- Von Natur aus zustandslos
- Bleibt in vielen fortgeschrittenen Funktionen hinter Flink zurück

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Apache Flink

Auch Flink hat einen ähnlichen akademischen Hintergrund wie Spark. Während Spark von der UC Berkley stammt, kommt Flink von der TU Berlin. Wie Spark unterstützt es auch die Lambda-Architektur.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Vorteile

- Innovationsführer in der Open-Source-Streaming-Landschaft
- Erstes echtes Streaming-Framework mit allen fortschrittlichen Funktionen wie Ereigniszeitverarbeitung, Wasserzeichen usw.
- Niedrige Latenz mit hohem Durchsatz, konfigurierbar je nach Anforderung
- Automatische Anpassung, nicht zu viele Parameter zum Einstellen
- Exactly Once
- Wird von großen Unternehmen wie Uber und Alibaba in großem Umfang genutzt.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Nachteile

- Etwas spät im Spiel, anfangs fehlte die Akzeptanz
- Die Community ist nicht so groß wie die von Spark, wächst aber schnell
- Bislang keine Annahme von Flink Batch bekannt, nur beliebt für Streaming.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Kafka Streams

Kafka Streams ist, im Gegensatz zu anderen Streaming-Frameworks, eine leichtgewichtige Bibliothek. Sie ist nützlich, um Daten aus Kafka zu streamen, zu transformieren und dann zurück zu Kafka zu senden. Wir können es als eine Bibliothek verstehen, die dem Java Executor Service Thread Pool ähnelt, aber mit eingebauter Unterstützung für Kafka. Sie kann gut in jede Anwendung integriert werden und ist sofort einsatzbereit.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

Vorteile

- Sehr leichtgewichtige Bibliothek, gut für Microservices, IOT-Anwendungen
- Benötigt keinen dedizierten Cluster
- Vererbt alle guten Eigenschaften von Kafka
- Unterstützt Stream-Joins, verwendet intern rocksDb für die Statusverwaltung
- Exactly Once (ab Kafka 0.11)

Nachteile

- Eng mit Kafka gekoppelt, kann nicht ohne Kafka verwendet werden
- Ziemlich neu im Anfangsstadium, muss noch in großen Unternehmen getestet werden
- Nicht für schwere Arbeiten wie Spark Streaming, Flink.

Quelle: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

5.5 COMPLEX EVENT PROCESSING

ToDo