Lund&Bendsen
developing developers

Kafka Streams

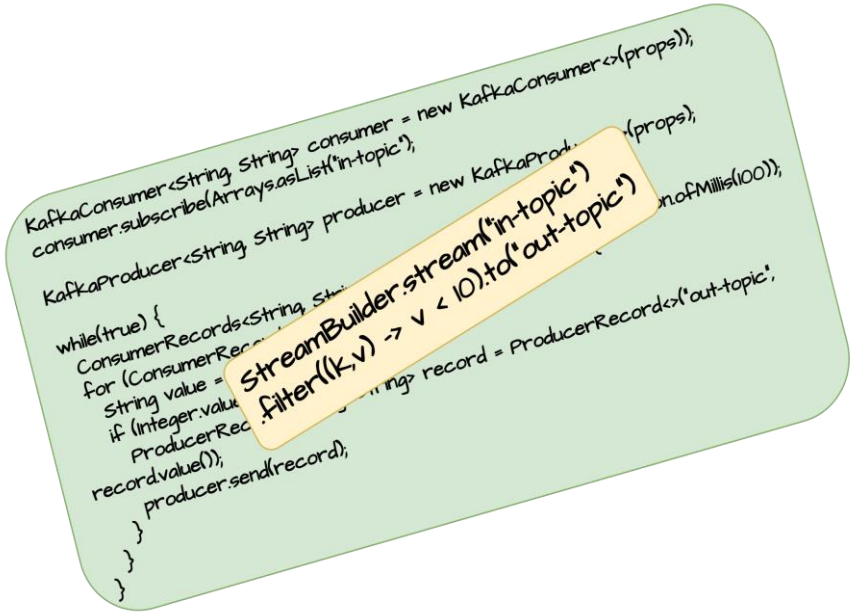
Why Kafka Streams

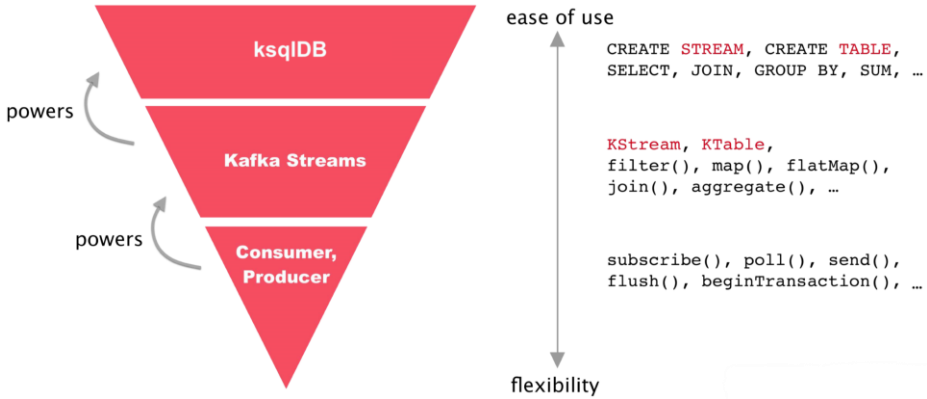
Lund&Bendsen
developing developers

The Kafka Streams API is a client-side library that sits alongside the existing Kafka client APIs. The Consumer and Producer APIs read and write to Kafka topics. Kafka Streams, which use the Consumer and Producer APIs under the hood, provides the ability to stream messages in real-time from a topic, processing and reacting to the data, before writing the transformed and enriched data to other topics.

Kafka streams provide a **declarative** approach to stream processing and provides a fluent domain specific language (DSL) that allows users to make a description of what needs to be done. Under the hood the framework takes take of all the operations needed to achieve the result. This is opposed to an **imperative** approach you would use if you were to build the same logic using vanilla Kafka consumers & producers.

Using DSL makes the code more scalable and robust and heavily reduces the number of code-lines



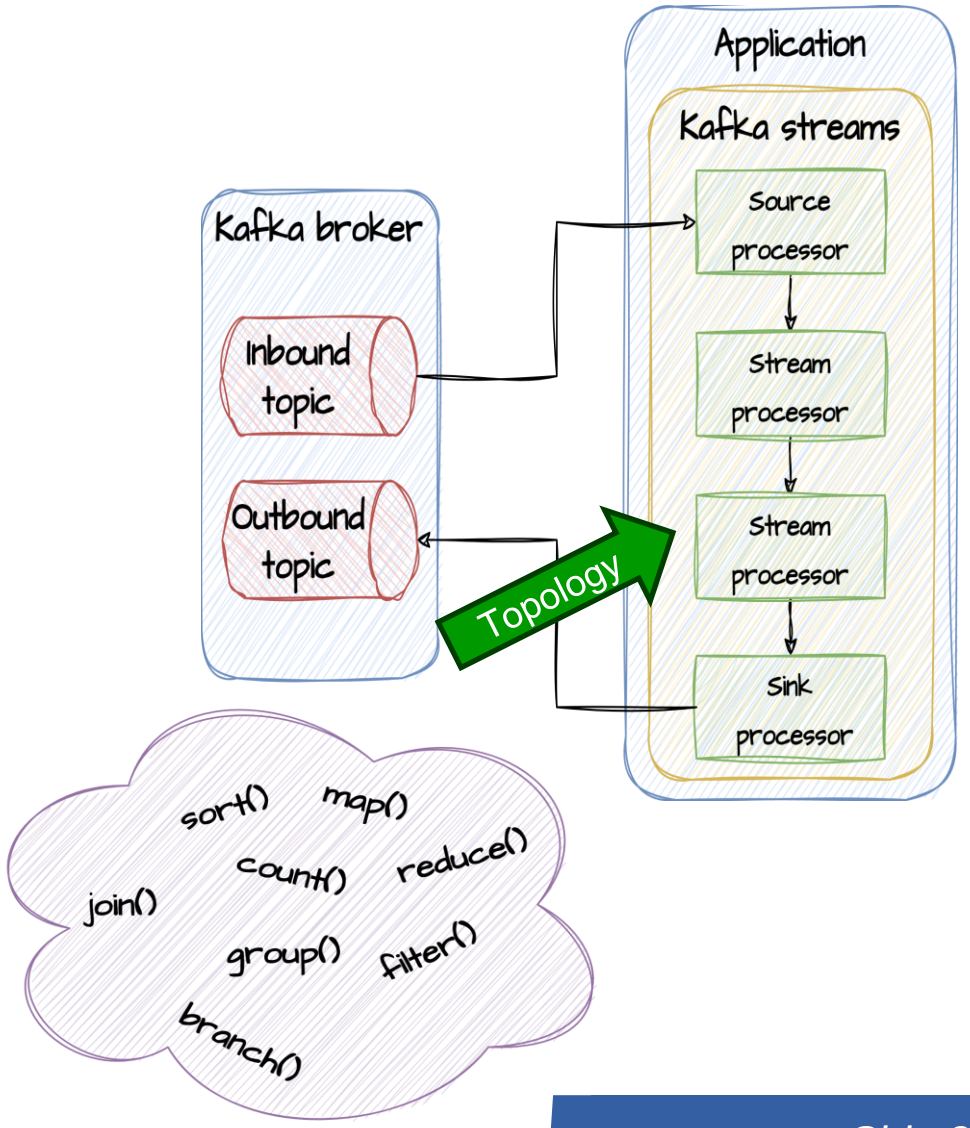


Why Kafka Streams

Lund&Bendsen
developing developers

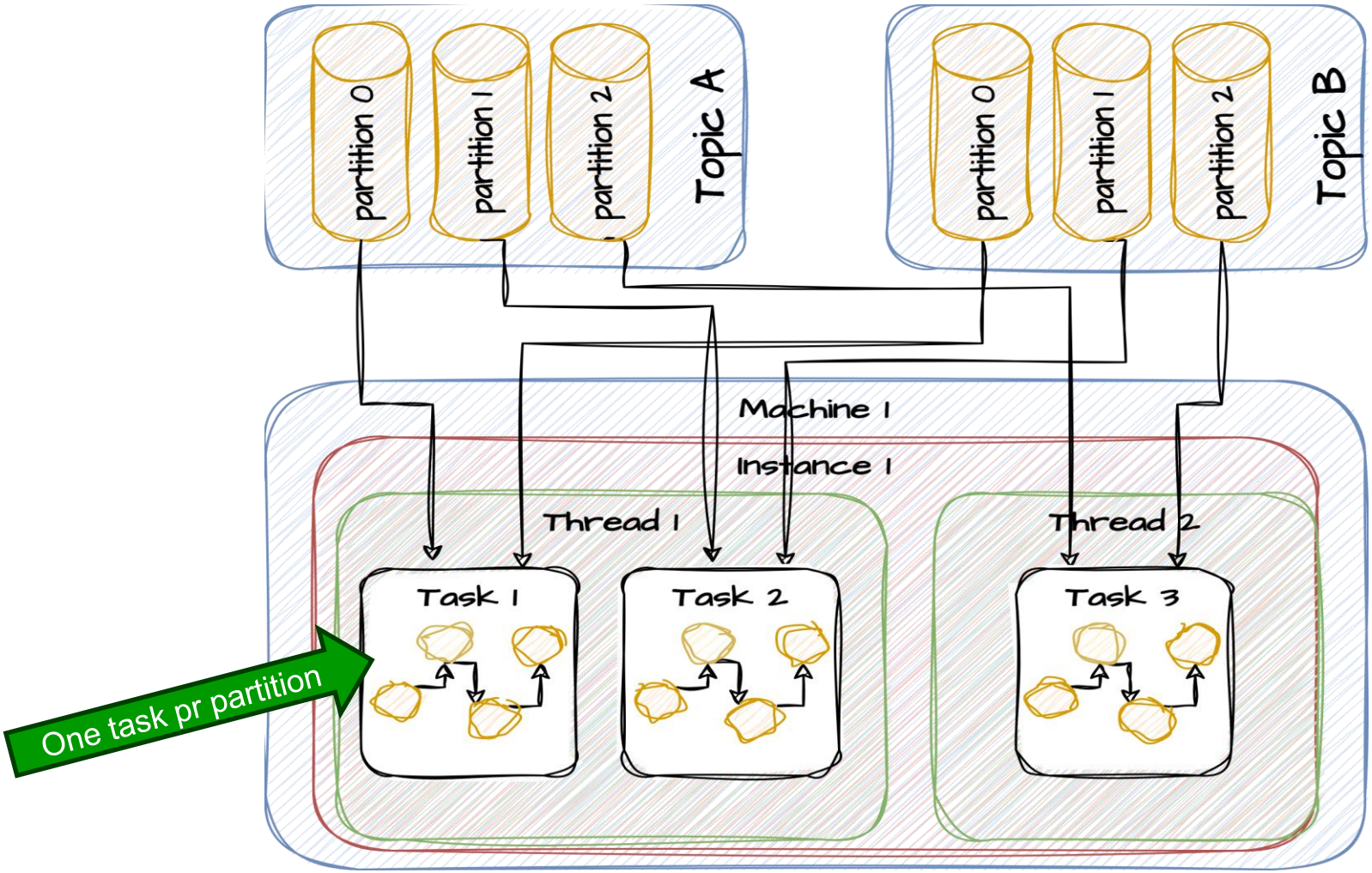
Kafka Streams is an ETL framework (**E**xtract-**T**ransform-**L**oad) that is based on the concept of processors. Processors are small building blocks that can be setup in a 'unix pipe'-like fashion. Output from one processor serves as input to the next. Processors can do filtering, projections and aggregations. This allows for Single Message Transform or stateful processing where the result depends on the history of previous events. Stateful allows us to join data from other streams or tables, group data in time windows, and aggregate related events, using a backing store to maintain state.

Each message is passed through a processor topology serially, such that each subsequent message is not processed until the previous one has completed. The topology can be divided into sub-topologies, and these sub-topologies will process messages in parallel with other sub-topologies.



Architecture

Lund&Bendsen
developing developers

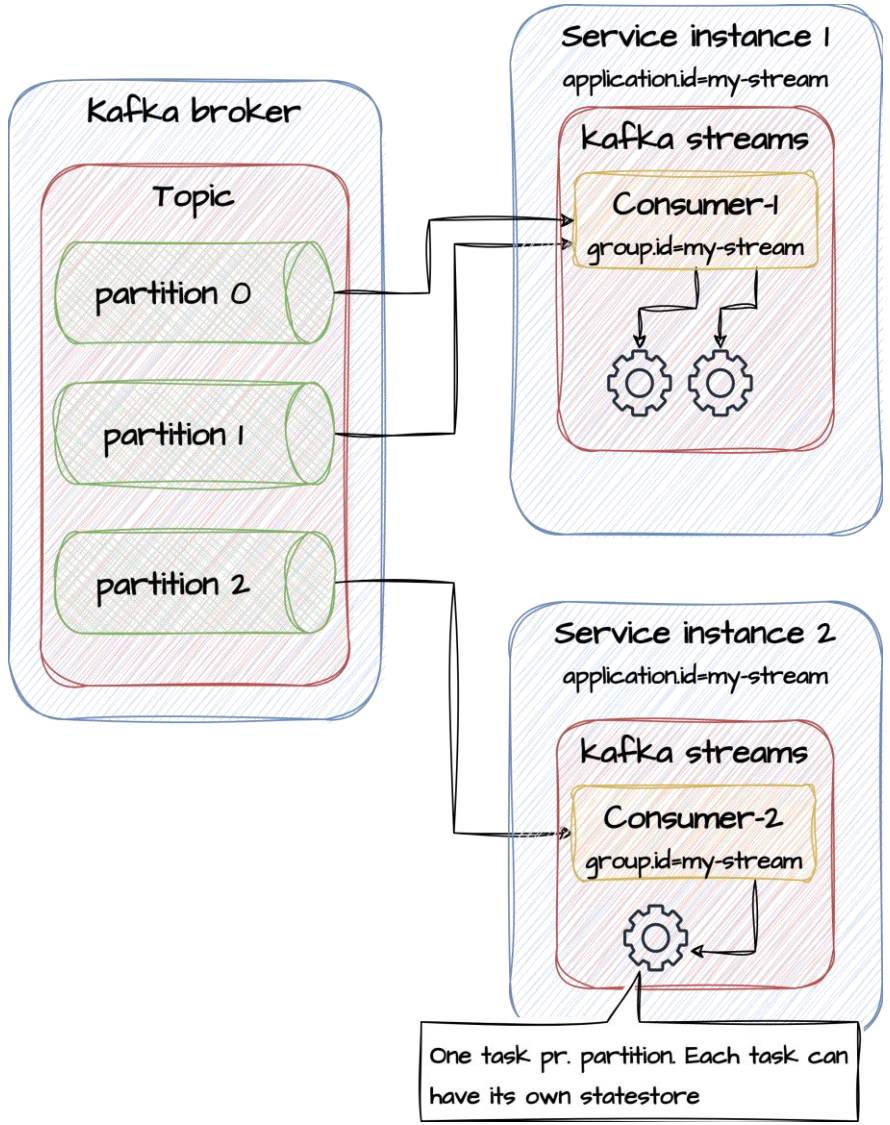


Tasks and threads

Lund&Bendsen
developing developers

Kafka Streams assigns each source topic partition to a task, and each task has its own copy of the topology.

A topic with 10 partitions will have 10 tasks. Threads, which are isolated and thread-safe, then execute the tasks. The creation of threads is managed by Kafka Streams so is not a developer concern. The number of threads is configurable, based on the **num.streams.threads** setting. This cannot exceed the task count.
If **num.streams.threads** was configured to be 5 threads, then with 10 partitions each thread would execute 2 tasks. The higher the thread count the better the utilisation that can be made of the available CPU resources.



Topology

Lund&Bendsen
developing developers

A Kafka Streams topology is a directed acyclic graph (DAG) that represents a Kafka Streams application's processing logic. It defines how data flows within your application, including the input sources, the transformations, and the output sinks. In other words, it specifies the sequence of processing steps and operations that your Kafka Streams application performs on data as it flows through the system.

Source Nodes: These nodes represent the starting point of your processing pipeline. Source nodes read data from Kafka topics and convert them into Kafka Streams, which are represented as KStream or KTable objects.

Processor Nodes: These nodes represent the intermediate processing steps in your application. Processors apply various transformations to the input data, such as filtering, mapping, aggregating, and joining. Each processor typically takes input from one or more parent nodes (source nodes or other processors) and produces output data that can be consumed by downstream processors.

Sink Nodes: Sink nodes represent the endpoint of your processing pipeline. They typically write the processed data to Kafka topics, external databases, or other output destinations.

State Stores: State stores are used for maintaining and accessing stateful data in your Kafka Streams application. They allow you to store and retrieve data for operations like windowed aggregations or maintaining changelogs.

Define topology with DSL

Lund&Bendsen
developing developers

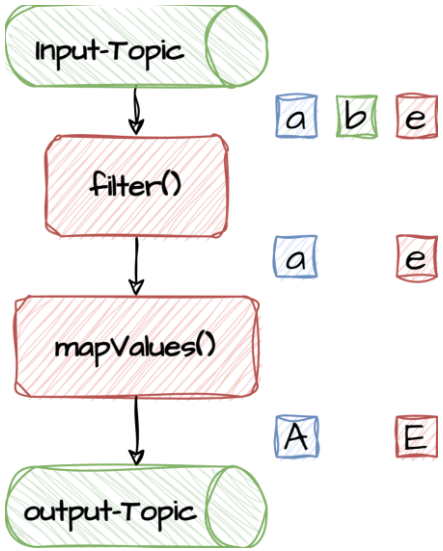
```
Properties props = ...

// Define topology - begin
StreamsBuilder builder = new StreamsBuilder();

builder.stream("inputTopic")
.filter((k,v) -> v.color() != green)
.mapValues(v -> v.label().toUpperCase())
.to("outputTopic", Produced.with(..., ...));
Topology topology = builder.build();
// Define topology - end

// Build and start stream
KafkaStreams streams = new KafkaStreams(topology, props);
streams.start();
```

Source processor

Filter processor

Map processor

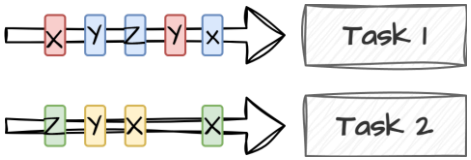
Sink processor



Creating source streams

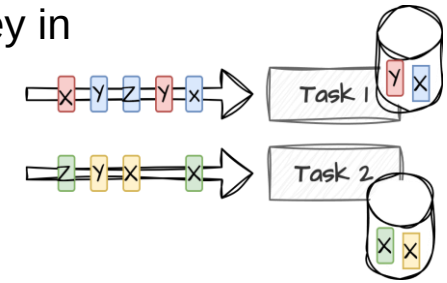
Kstream is an abstraction of a continuous stream of events that can be build from a kafka topic.

```
KStream<String, String> myKStream =
builder.stream("inputTopic");
```



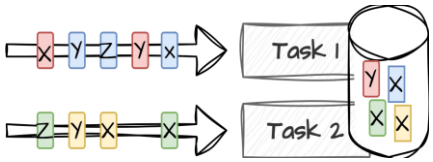
Ktable is a stateful table like structure that holds the latest value for each key in the processed partition

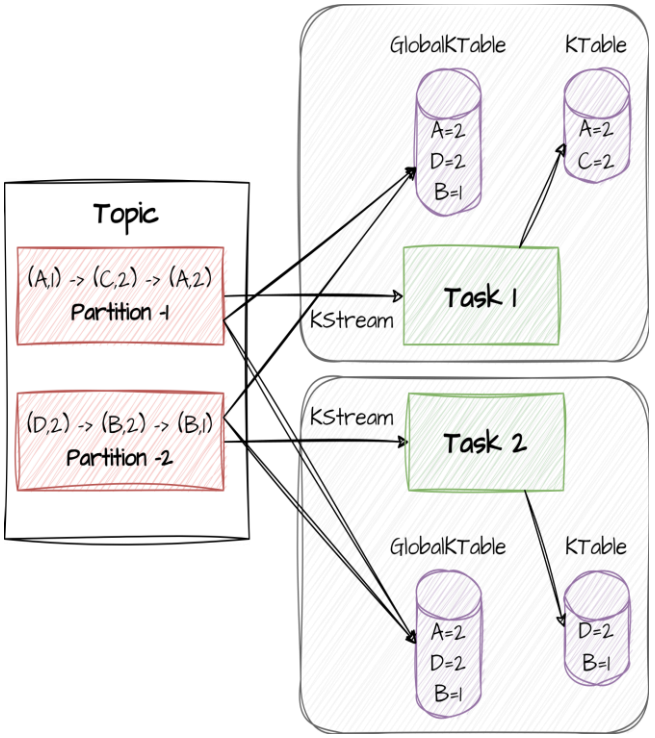
```
KTable<String, String> myKTable =
builder.table("inputTopic");
```



GlobalKtable is a stateful table like structure that holds the latest value for each key in the processed topic

```
GlobalKTable<String, String> myGlobalKTable
= builder.globalTable(inputTopic);
```





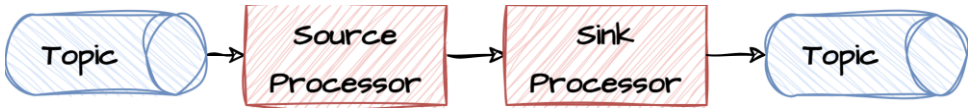
Exercise

Lund&Bendsen
developing developers

8.1

Create simple stream

- Create an input topic named `inputTopic` and an output topic named `outputTopic`
- Write a topology that read messages from input and write them to output
- Start the stream application
- Start a producer that produces messages to the input topic like this:
- Open PowerShell window and navigate to <project>/docker folder
- Type **`.\kafka-producer-perf-test.bat --topic inputTopic --num-records 10000 --throughput 1 --record-size 5`**
- Open a new PowerShell window and navigate to <project>/docker folder:
- Type **.\kafka-consumer.bat --topic outputTopic --from-beginning --property print.partition=true --property print.offset=true**
- Verify that messages are being sent to the output topic



Stateless processors

Lund&Bendsen
developing developers

branch() splits the stream into one or more streams based on predicates

```
KStream<String, String> stream = ...;
Map<String, KStream<String, String>> branches =
  stream.split(Named.as("Branch-"))
  .branch((key, value) -> key.startsWith("A"),  /* first predicate  */ Branched.as("A"))
  .branch((key, value) -> key.startsWith("B"),  /* second predicate */ Branched.as("B"))
  .defaultBranch(Branched.as("C"))
);
```

Or:

```
KStream<K, V> stream = ...;
stream.filter((key, value) -> key.startsWith("A")).to(“Branch-A");
stream.filter((key, value) -> key.startsWith(“B")).to(“Branch-B");
stream.filter((key, value) -> !key.startsWith("A") && !key.startsWith(“B")).to(“Branch-C");
```

Stateless processors

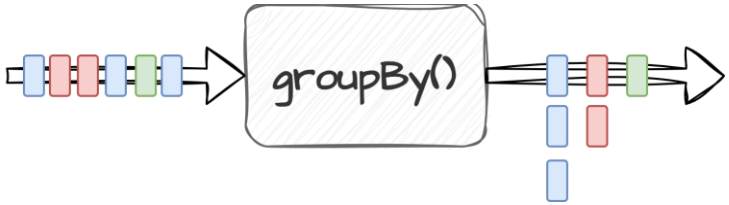
Lund&Bendsen
developing developers

filter() removes records from a stream based on a predicates

```
KStream<K, V> stream = ...;
stream.filter((key, value) -> key.startsWith("A"));
```

groupBy() groups records by a new key

```
KStream<byte[], String> stream = ...;
KGroupedStream<String, String> groupedStream = stream.groupBy((key, value) -> key,
Grouped.with(Serdes.String(), Serdes.String()));
```



Exercise

Lund&Bendsen
developing developers

8.2

Create filtered stream

- Create an input topic named `inputTopic` and an output topic named `outputTopic`
- Write a topology that read messages from input topic, filter off all messages not containing the letter ‘a’ or ‘A’ (uppercase) and write them to output topic
- Start the stream application
- Start a producer that produces messages to the input topic like this:
- Open powershell window and navigate to <project>/docker folder
- Type **`.\kafka-producer-perf-test.bat --topic inputTopic --num-records 10000 --throughput 1 --record-size 5`**
- Open a new PowerShell window and navigate to <project>/docker folder:
- Type **.\kafka-consumer.bat --topic outputTopic --from-beginning --property print.partition=true --property print.offset=true**
- Verify that only messages containing the letter ‘a’ are being sent to the output topic

Stateless processors

Lund&Bendsen
developing developers

map() / mapValues() takes one record, transforms it and produces a new record

```
KStream<K, V> stream = ...;
KStream<String, String> mappedStream = input.mapValues(value -> value.toUpperCase());
```

flatMap() takes one record and produces zero, one or more records

```
KStream<K, V> stream = ...;
stream.flatMapValues(value -> Arrays.asList(value.toLowerCase().split(" ")));
```

Exercise

Lund&Bendsen
developing developers

8.3

Create mapped stream

- Create an input topic named `inputTopic` and an output topic named `outputTopic`
- Write a topology that read messages from input topic, replace all ‘a’ and ‘A’ characters with an hyphen(-) and write them to output topic
- Start the stream application
- Start a producer that produces messages to the input topic like this:
- Open powershell window and navigate to <project>/docker folder
- Type **`.\kafka-producer-perf-test.bat --topic inputTopic --num-records 10000 --throughput 1 --record-size 5`**
- Open a new powershell window and navigate to <project>/docker folder:
- Type **.\kafka-consumer.bat --topic outputTopic --from-beginning --property print.partition=true --property print.offset=true**
- Verify that only messages have the letter ‘a’ or ‘A’ replaced with ‘-’

Stateless processors

Lund&Bendsen
developing developers

merge() merges records from two streams into a new stream

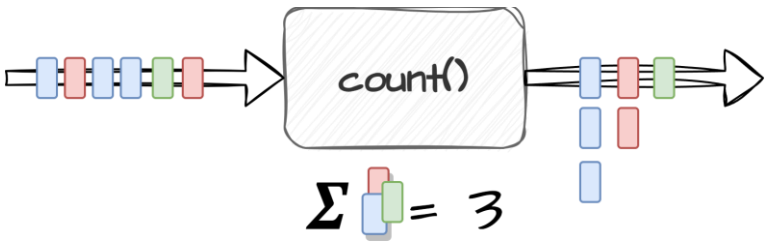
```
KStream<K, V> streamA = ...;
KStream<K, V> streamB = ...;
KStream<String, String> mergedStream = streamA.merge(streamB);
```

Stateful processors

Lund&Bendsen
developing developers

count() counts the number of records by the grouped key

```
KStream<K, V> stream = ...;
stream.groupBy((key, value) -> value)
.count()
```

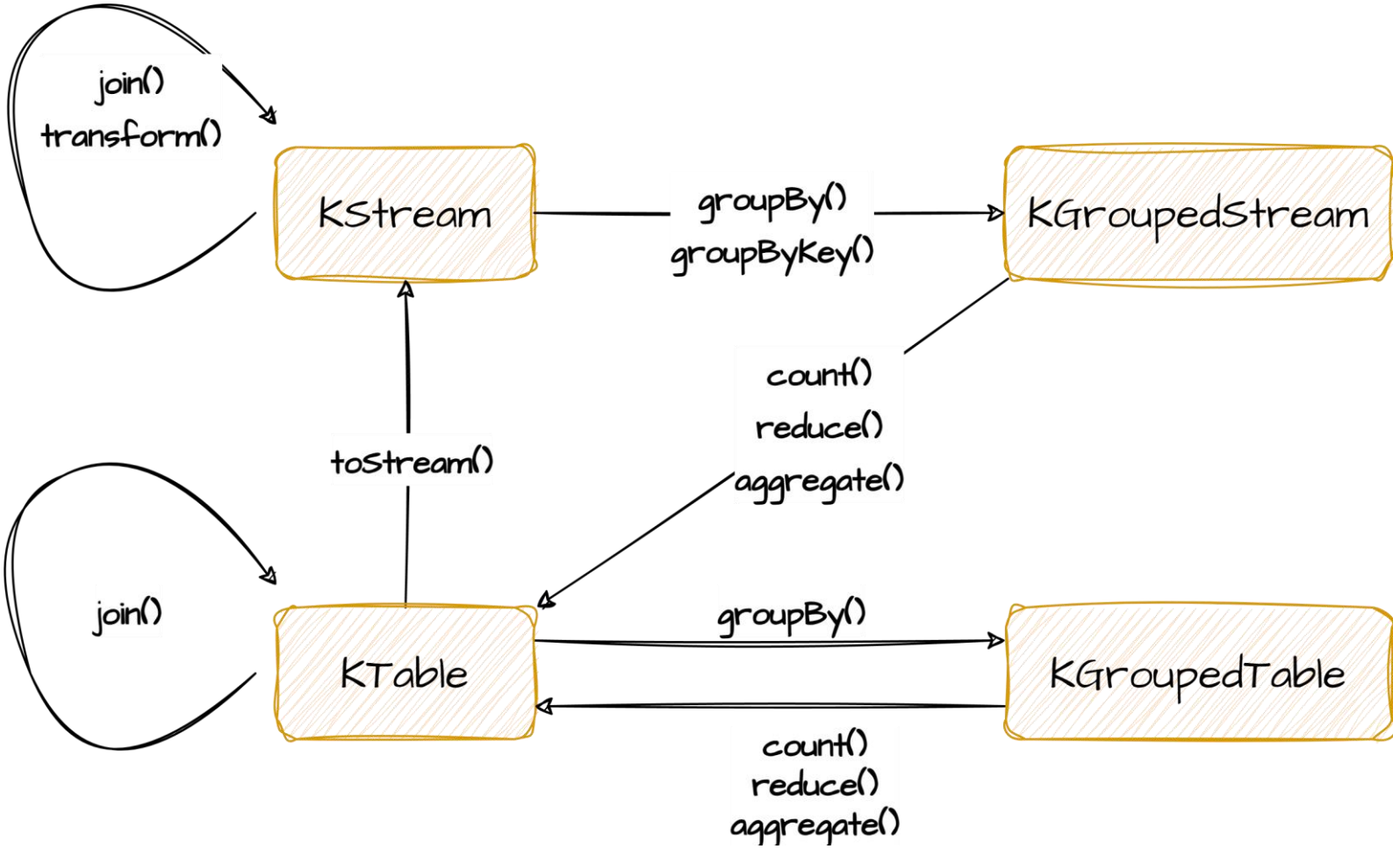


join() merges records from two streams into a new stream. Co-partitioning needs to be in place

```
KStream<K, V> streamA = ...;
KStream<K, V> streamB = ...;

KStream<String, String> joinedStream =
  streamA.join(streamB, (valueA, valueB) -> valueA + "-" + valueB,
    JoinWindows.of(10000),
    Materialized.with(Serdes.String(), Serdes.String());
```

Transformation relationships

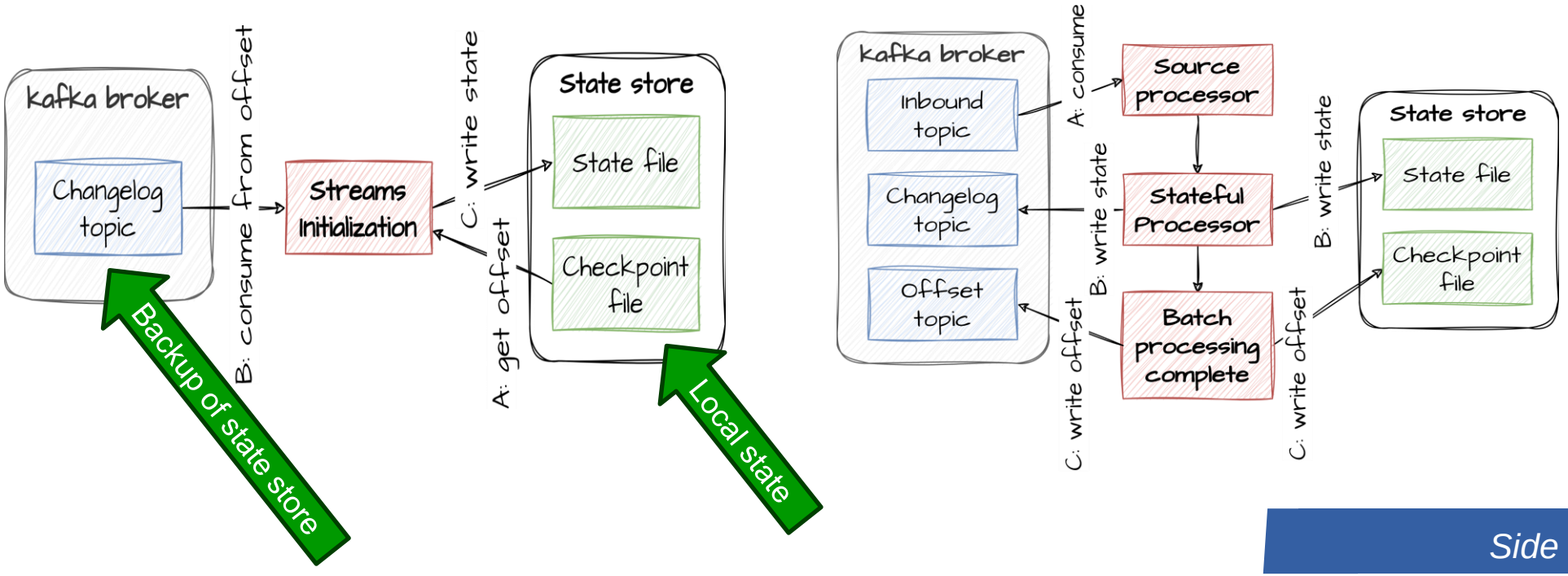
Lund&Bendsen
developing developers



State stores

Lund&Bendsen
developing developers

When processing infinite streams, we cannot do join or aggregation processing without maintaining state of precious records in the stream. In Kafka streams this is handled through state stores. Each task has its own state store. A state store is a local cache that is backed by a lightweight database called Rocks DB. Alternative implementations can be chosen such as in-mem, Redis, etc. Because tasks can be ephemeral or temporarily unavailable (crashes, network issues etc), Kafka maintains a backup of the state store in a compacted Kafka changelog topic. This allows new tasks to take over and proceed from where old tasks left without having to replay the entire event stream. New tasks will simply rebuild its state store from the change-log topic during initialization

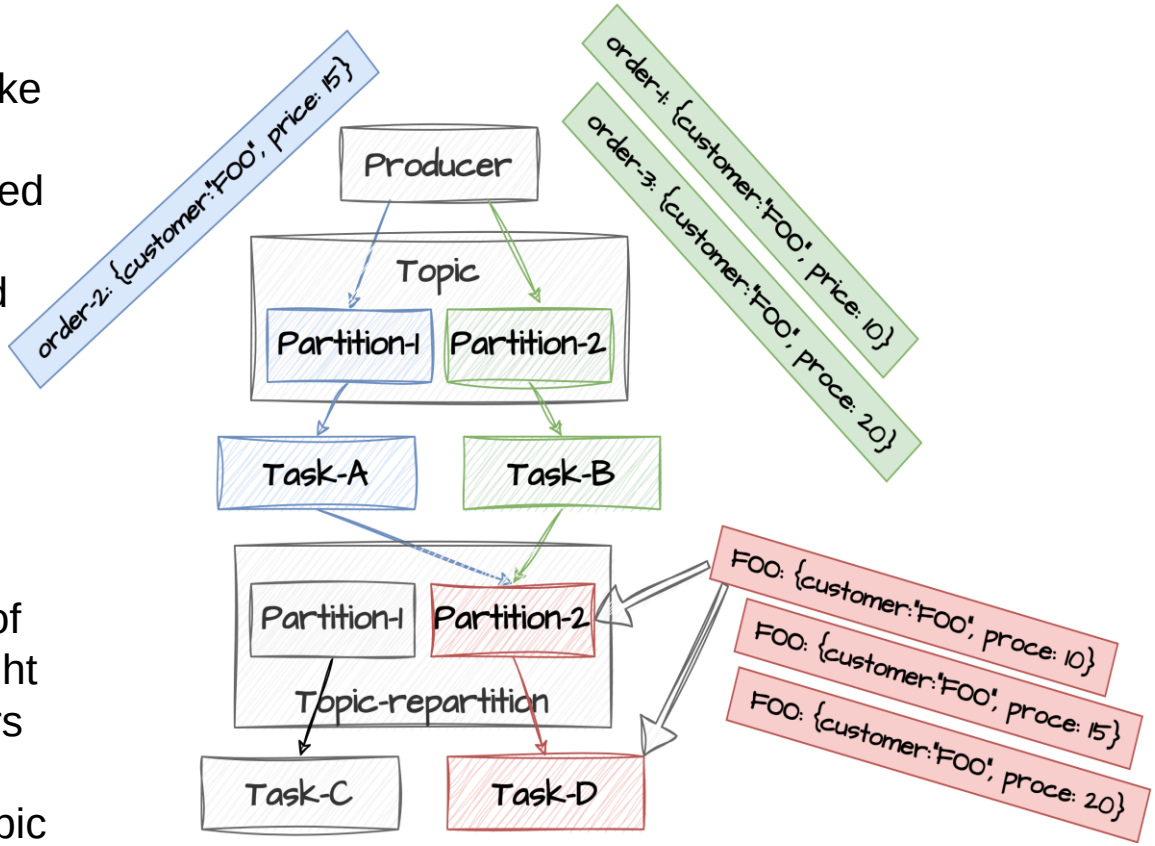


Re-partition

Lund&Bendsen
developing developers

Sometimes you need to change keys while processing. This happens with operations like `map()`, `selectKey()` etc. Typically you do this when the message has been transformed / mapped and the new message has a new logical identity that need to be synchronized with the other Stream tasks.

In that case Kafka streams uses an intermediate repartitioning topic to consolidate all messages in the right partitions based on their newly assigned keys. This ensures that further processing of these re-keyed messages end up on the right tasks. When creating topologies that triggers re-keying of messages Kafka Streams automatically creates the new repartition topic and new tasks that listen to these repartition-topics.



Exercise

Lund&Bendsen
developing developers

8.4

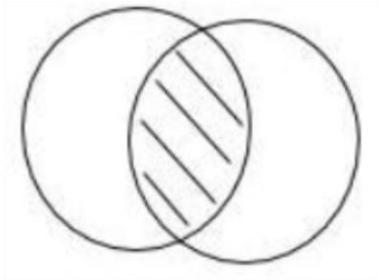
Create groupby+count streams

- Create one input topics `inputTopic`
- Write a topology that reads from the inputStream, re-key the stream to use the first letter in the value of the record to determine if the key should be v (vocal) or c (consonant).
- Count how many records that are vocals and how many are consonants. Output the result to a new topic called outputTopic
- Start the stream application
- Open a PowerShell window and navigate to <project>/docker folder
- Start a producer that produces messages to the input topic like this:
- Type **`.\kafka-producer-perf-test.bat --topic inputTopic --num-records 1000 --throughput 10 --record-size 1`** in one window
- Open a new PowerShell window and navigate to <project>/docker folder:
- Type **.\kafka-consumer.bat --topic outputTopic --property print.key=true --property print.value=true --property value.deserializer=org.apache.kafka.common.serialization.LongDeserializer**
- Verify that a stream of counts is produced

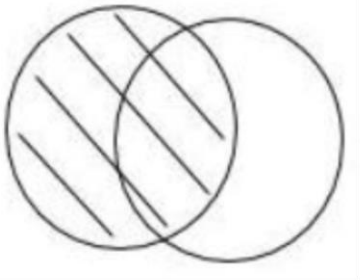
Joins

Lund&Bendsen
developing developers

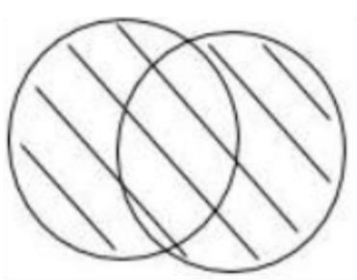
Its possible to join streams from multiple topics in Kafka. Three different types of joins are supported depending on the type of streams that you join:



Inner:
KStream-KStream
KTable-KTable
KStream-KTable
KStream-GlobalKTable



Inner:
KStream-KStream
KTable-KTable
KStream-KTable
KStream-GlobalKTable



Outer:
KStream-KStream
KTable-KTable

- **Inner Joins:** Emits an output when both input sources have records with the same key.
- **Left Joins:** Emits an output for each record in the left or primary input source. If the other source does not have a value for a given key, it is set to null.
- **Outer Joins:** Emits an output for each record in either input source. If only one source contains a key, the other is null.

Windowing

Lund&Bendsen
developing developers

When doing join operations on two infinite KStreams you need to narrow equality of keys based on a time window.

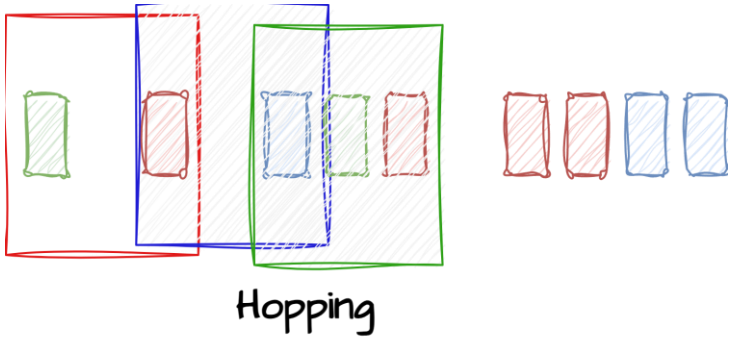
A windows is defined by a fixed width and an incremental factor (how it moves forward in time). If two events happens inside the same window they can be joined.

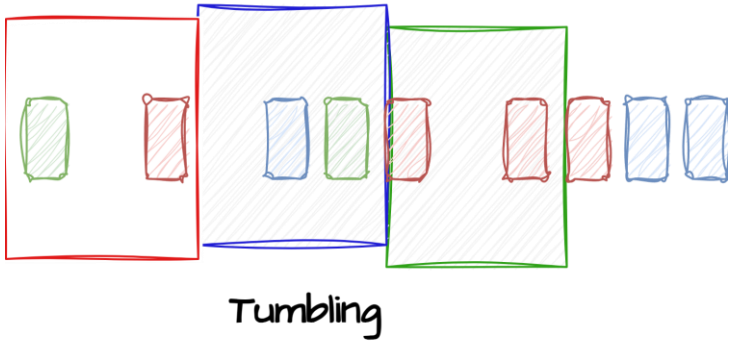
There are different definitions of time windows

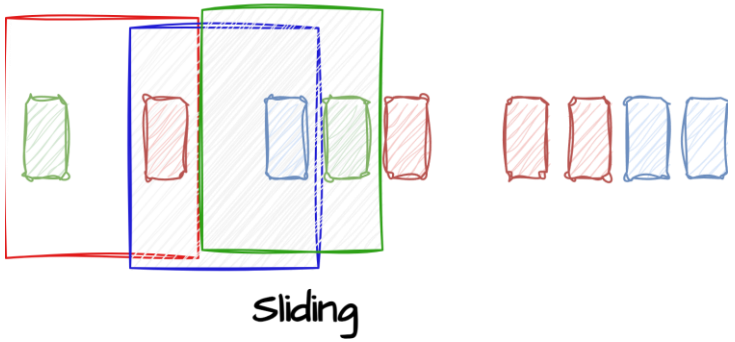
Hopping, incremental step is fixed and less than the width of the window

Tumbling, incremental step is fixed and equal to the width of the window

Sliding, incremental step depends on the arrival of events



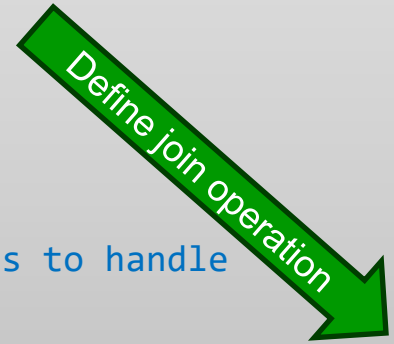




Join example

Lund&Bendsen
developing developers

```
KStream<String, String> leftStream = builder.stream("topic-A");
KStream<String, String> rightStream = builder.stream("topic-B");

// ValueJoiner is applied whenever there is identical keys in both streams.
// Returns the result of the join operations
ValueJoiner<String, String, String> valueJoiner = (leftValue, rightValue) -> {
    return leftValue + rightValue;
};

// Do the join between the two KStreams and narrow by a
// timewindow. A state store is created behind the scenes to handle
// the windowed state between the two streams
Kstream joinedStream<String, String> = leftStream.join(rightStream, valueJoiner,
JoinWindows.of(Duration.ofSeconds(10)));
```



co-partition

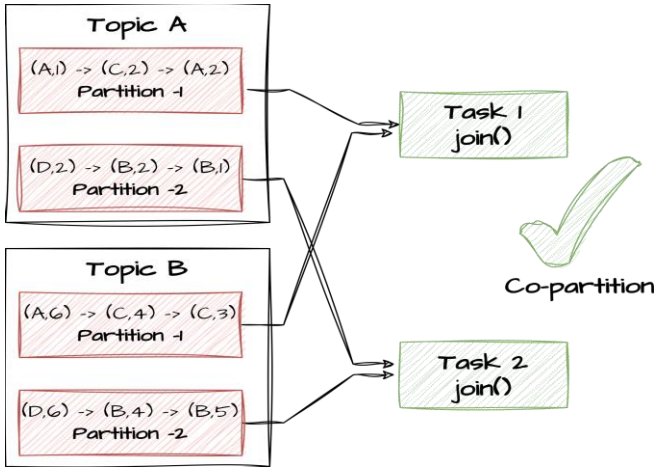
Lund&Bendsen
developing developers

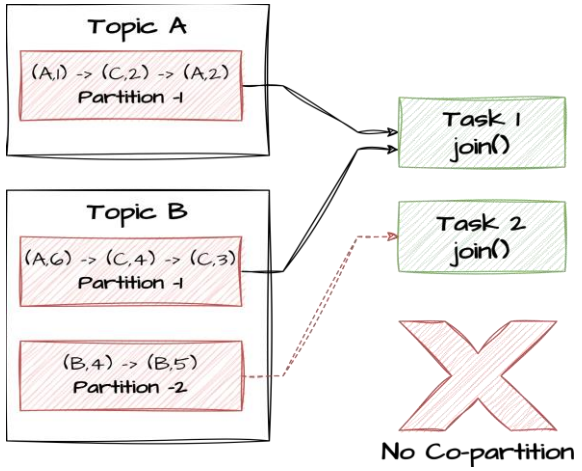
Kafka stream joins work on key level. Keys that are identical can be joined.

When joining two KStreams, the two source topics must be co-partitioned, meaning that:

- The keys must be semantically the same.
- The topics must have the same number of partitions
- The partition-algorithm must work the same way on the producer side.

This is important because otherwise the different tasks won't get the right records





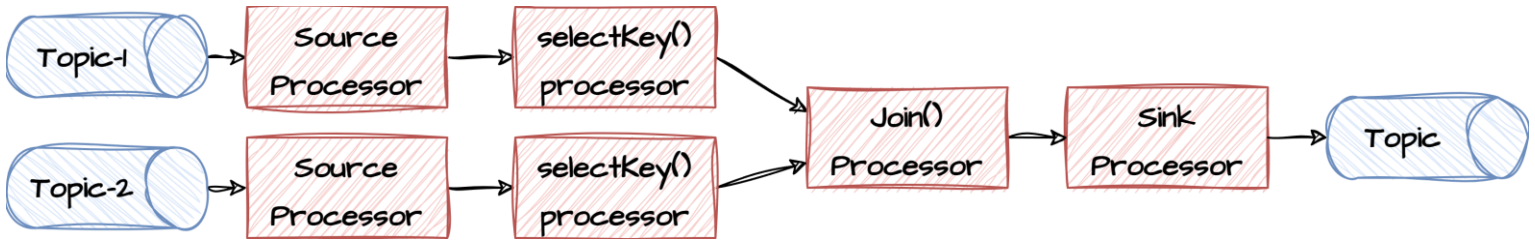
Exercise

Lund&Bendsen
developing developers

8.5

Create joined streams

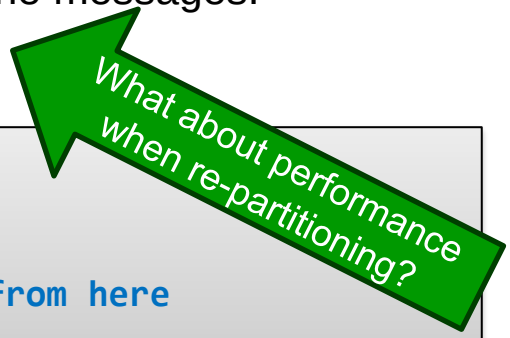
- Create two input topics `inputTopic-1 & inputTopic-2`
- Write a topology that read messages from the two topics, joins them (concat values) and write the output to an output topic. Use a timewindow of 10 seconds
- Before joining the two streams make sure to re-key the streams to be able to join the streams. The key could be the first character in the message value
- Start the stream application
- Start a producer that produces messages to the input topic like this:
- Open two PowerShell window and navigate to <project>/docker folder
- Type **`.\kafka-producer-perf-test.bat --topic inputTopic-1 --num-records 10000 --throughput 1 --record-size 5`** in one window
- Type **`.\kafka-producer-perf-test.bat --topic inputTopic-2 --num-records 10000 --throughput 1 --record-size 5`** in another window
- Open a new powershell window and navigate to <project>/docker folder:
- Type **.\kafka-consumer.bat --topic outputTopic --from-beginning --property print.partition=true --property print.offset=true**
- Verify that a stream of joined streams is produced



Sub-topologies

Lund&Bendsen
developing developers

Sub-topologies is a way of parallelizing your stream processing. Simply put the degree of parallelism depends on the number of tasks that can be run simultaneously which again depends on the number of partitions in the input topics. By breaking your topology into smaller chunks, you can increase the number of running tasks by introducing intermediate topics in the processing. This can be done explicitly using the `repartition()` operation or implicitly through re-keying the messages.



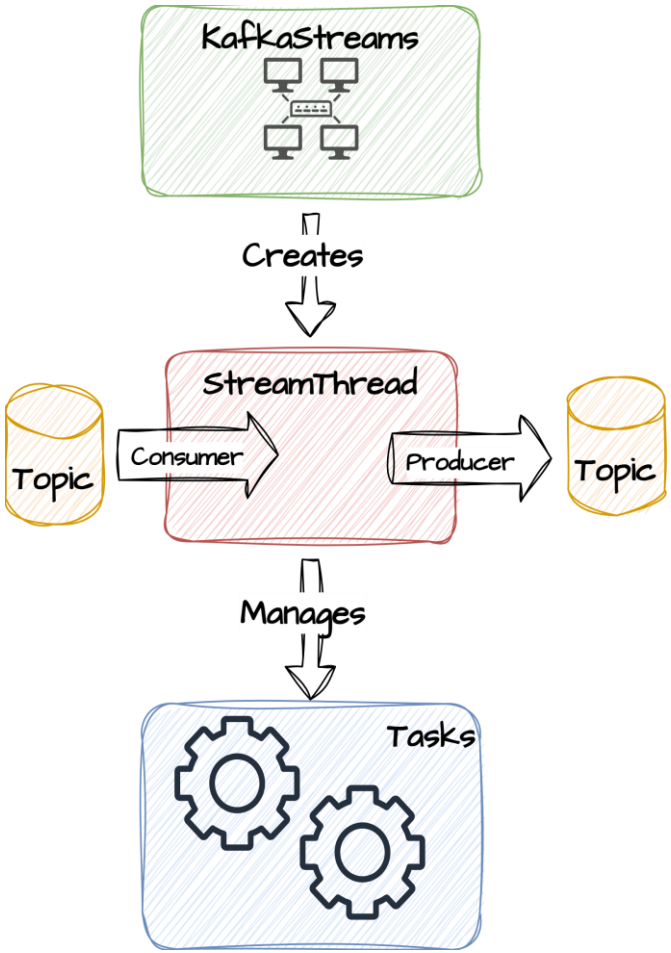

```
new StreamsBuilder()
    .stream("orders") // Create stream from topic
    .flatMap(Ops::split)
    .repartition() // write result to a topic and create a new stream from here
    .filter(Ops::excludeArchivedValues)
    .to("active-order-items"); // write result to topic
```

Construction the topologies

Lund&Bendsen
developing developers

During startup of the `KafkaStream` instance these are roughly the steps that happen:

- The Topology is built based on the DSL
- A `KafkaStreams` instance is created using the specified configuration
- `KafkaStreams` creates `StreamsConfig.NUM_STREAM_THREADS_CONFIG` instances of `StreamThread`. Each `StreamThread` has its own producer and consumer with the application.id used as the consumergroup.id
- A `StreamThread` is a wrapper of 3 underlying threads. (Producer network thread, Coordinator heartbeat thread and the processor thread)
- One or more tasks are assigned to each `StreamThread`
- Using the `StreamPartitionAssignor` interface, one of the consumers is appointed the role of group leader. The leader ensures that partitions are evenly assigned among the different `StreamThreads` and that co-located partitions are processed by the same consumer
- All `StreamThreads` are started when the `KafkaStreams` instance is started.



Visualize topology

Lund&Bendsen
developing developers

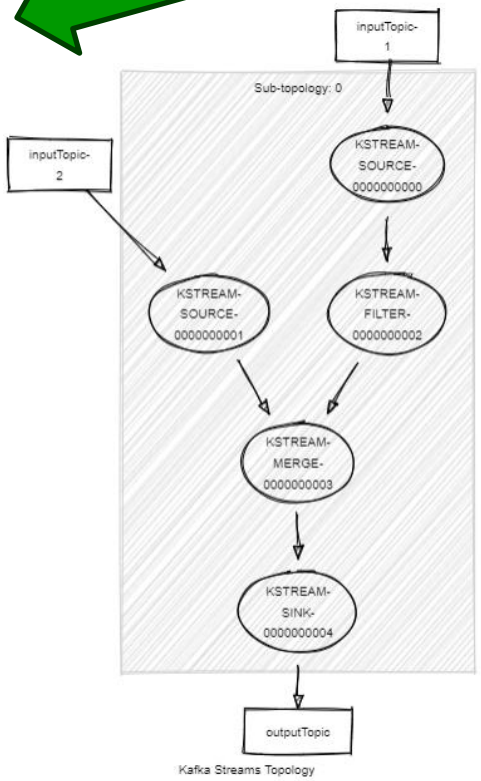
When building the topology, you can get a description of how the internal topology is build by calling

`Topology#describe()`. Using this online tool https://zz85.github.io/kafka-streams-viz/ you can get a pretty visualization of the topology:

Try this

```
KStream<String, String> inputstream1 = builder.stream("inputTopic-1");
KStream<String, String> inputstream2 = builder.stream("inputTopic-2");
KStream mergedstream = inputstream1.filter((k, v) -> v.length() >
10).merge(inputstream2);
mergedstream.to("outputTopic", Produced.with(Serdes.String(),
Serdes.Long()));
```

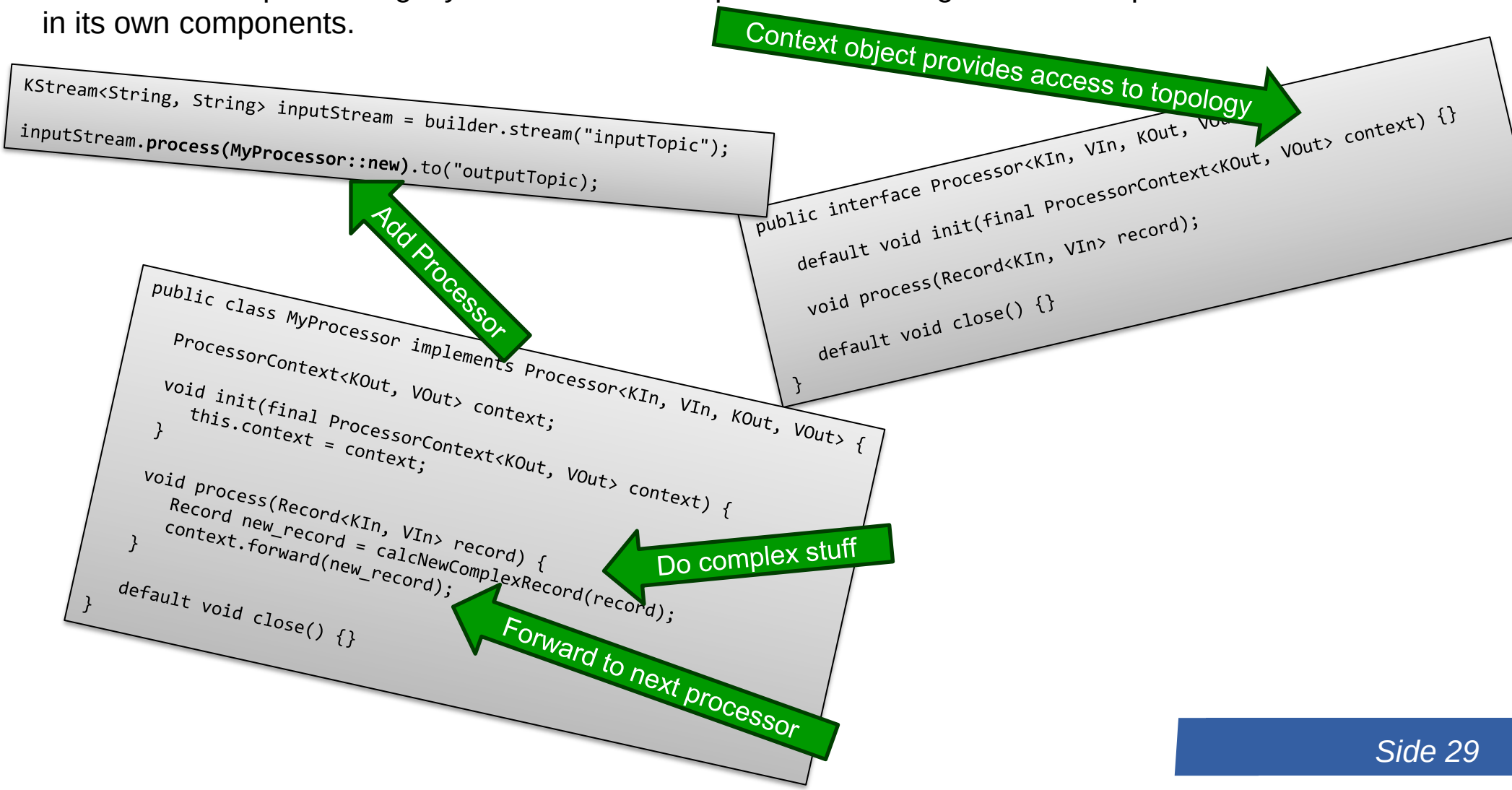
```
Topologies:
   Sub-topology: 0
    Source: KSTREAM-SOURCE-0000000000 (topics: [inputTopic-1])
      --> KSTREAM-FILTER-0000000002
    Processor: KSTREAM-FILTER-0000000002 (stores: [])
      --> KSTREAM-MERGE-0000000003
      <-- KSTREAM-SOURCE-0000000000
    Source: KSTREAM-SOURCE-0000000001 (topics: [inputTopic-2])
      --> KSTREAM-MERGE-0000000003
    Processor: KSTREAM-MERGE-0000000003 (stores: [])
      --> KSTREAM-SINK-0000000004
      <-- KSTREAM-FILTER-0000000002, KSTREAM-SOURCE-0000000001
    Sink: KSTREAM-SINK-0000000004 (topic: outputTopic)
      <-- KSTREAM-MERGE-0000000003
```



Custom Processor

Lund&Bendsen
developing developers

Its possible to specify custom processors and add them to the topology. This allows for more advanced processing if you have more complex business-logic that is incapsulated in its own components.



Exercise / demo

Lund&Bendsen
developing developers

8.6

Create custom processor

- Create one input topics `inputTopic`
- Write a topology that reads from the inputStream, holds back all records until a timer is triggered (3 sec interval) in a custom processor. Then concatenate all values and pass the new value downstream
- Start the stream application
- Open a PowerShell window and navigate to <project>/docker folder
- Start a producer that produces messages to the input topic like this:
- Type **`.\kafka-producer-perf-test.bat --topic inputTopic --num-records 1000 --throughput 10 --record-size 1`** in one window
- Open a new PowerShell window and navigate to <project>/docker folder:
- Type **.\kafka-consumer.bat --topic outputTopic**
- Verify that a stream of concatenated values are produced

Testing

Lund&Bendsen
developing developers

```
public class SimpleStreamTopology {


    public static Topology getTopology() {
        StreamsBuilder builder = new StreamsBuilder();
        var sourceStream = builder.stream(SimpleStreamConstants.SOURCE_TOPIC, ...);
        sourceStream.filter((k, v) -> v%2 == 0)
                .to(SimpleStreamConstants.FILTERED_TOPIC, ...);
        return builder.build();
    }
}
```

```
public class SimpleStreamTopologyTest {
    private TestInputTopic<String, Integer> inputTopic;
    private TestOutputTopic<String, Integer> outputTopic;

    @BeforeEach
    public void init() {
        Properties props = new Properties();
        TopologyTestDriver testDriver = new TopologyTestDriver(SimpleStreamTopology.getTopology(),
props);
        inputTopic = testDriver.createInputTopic(SimpleStreamConstants.SOURCE_TOPIC,...);
        outputTopic = testDriver.createOutputTopic(SimpleStreamConstants.FILTERED_TOPIC,...);
    }

    @Test
    public void shouldFilterEvenvalues() {
        for (int i = 0; i < 10; i++) inputTopic.pipeInput("" + i, i);
        var outputList = outputTopic.readValuesToList();
        assertEquals(5, outputList.size());
    }
}
```

TestDriver to run the tests