Streams DSL

The Kafka Streams DSL (Domain Specific Language) is built on top of the Streams Processor API. It is the recommended for most users, especially beginners. Most data processing operations can be expressed in just a few lines of DSL code.

Overview

In comparison to the Processor API, only the DSL supports:

- Built-in abstractions for streams and tables in the form of KStream, KTable, and GlobalKTable. Having first-class support for streams and tables is crucial because, in practice, most use cases require not just either streams or databases/tables, but a combination of both. For example, if your use case is to create a customer 360-degree view that is updated in real-time, what your application will be doing is transforming many input *streams* of customer-related events into an output *table* that contains a continuously updated 360-degree view of your customers.
- Declarative, functional programming style with stateless transformations (e.g. `map` and `filter`) as well as stateful transformations such as aggregations (e.g. `count` and `reduce`), joins (e.g. `leftJoin`), and windowing (e.g. session windows).

With the DSL, you can define processor topologies (i.e., the logical processing plan) in your application. The steps to accomplish this are:

1. Specify one or more input streams that are read from Kafka topics
2. Compose transformations on these streams.
3. Write the resulting output streams back to Kafka topics, or expose the processing results of your application directly to other applications through Interactive Queries (e.g., via a REST API).

After the application is run, the defined processor topologies are continuously executed (i.e., the processing plan is put into action). A step-by-step guide for writing a stream processing application using the DSL is provided below.

Once you have built your Kafka Streams application using the DSL you can view the underlying `Topology` by first executing `StreamsBuilder#build()` which returns the `Topology` object. Then to view the `Topology` you call `Topology#desribe()`. Full details on describing a `Topology` can be found in describing a topology.

For a complete list of available API functionality, see also the Kafka Streams Javadocs.

Creating source streams from Kafka

You can easily read data from Apache Kafka® topics into your application. The following operations are supported.

Reading from Kafka	Description
Stream • *input topics* → KStream	Creates a KStream from the specified Kafka input topics and interprets the data as a record stream. A `KStream` represents a *partitioned* record stream. (details) In the case of a KStream, the local KStream instance of every application instance will be populated with data from only **a subset** of the partitions of the input topic. Collectively, across all application instances, all input topic partitions are read and processed. `import org.apache.kafka.common.serialization.Serdes;` `import org.apache.kafka.streams.StreamsBuilder;` `import org.apache.kafka.streams.kstream.KStream;` `StreamsBuilder builder = new StreamsBuilder();` `KStream<String, Long> wordCounts = builder.stream(` `"word-counts-input-topic" /* input topic */`

Reading from Kafka

```
    Consumed.with(
      Serdes.String(), /* key serde */
      Serdes.Long()   /* value serde */
    );
```

If you do not specify SerDes explicitly, the default SerDes from the configuration are used.

You **must specify SerDes explicitly** if the key or value types of the records in the Kafka input topics do not match the configured default SerDes. For information about configuring default SerDes, available SerDes, and implementing your own custom SerDes see Data Types and Serialization.

Several variants of `stream` exist, for example to specify a regex pattern for input topics to read from).

Table

- *input topic* → KTable

Reads the specified Kafka input topic into a KTable. The topic is interpreted as a changelog stream, where records with the same key are interpreted as UPSERT aka INSERT/UPDATE (when the record value is not `null`) or as DELETE (when the value is `null`) for that key. (details)

In the case of a KTable, the local KTable instance of every application instance will be populated with data from only **a subset** of the partitions of the input topic. Collectively, across all application instances, all input topic partitions are read and processed.

You must provide a name for the table (more precisely, for the internal state store that backs the table). This is required for supporting Interactive Queries against the table. When a name is not provided the table will not queryable and an internal name will be provided for the state store.

If you do not specify SerDes explicitly, the default SerDes from the configuration are used.

You **must specify SerDes explicitly** if the key or value types of the records in the Kafka input topics do not match the configured default SerDes. For information about configuring default SerDes, available SerDes, and implementing your own custom SerDes see Data Types and Serialization.

Several variants of `table` exist, for example to specify the `auto.offset.reset` policy to be used when reading from the input topic.

Global Table

- *input topic* → GlobalKTable

Reads the specified Kafka input topic into a GlobalKTable. The topic is interpreted as a changelog stream, where records with the same key are interpreted as UPSERT aka INSERT/UPDATE (when the record value is not `null`) or as DELETE (when the value is `null`) for that key. (details)

In the case of a GlobalKTable, the local GlobalKTable instance of every application instance will be populated with data from all input topic partitions. Collectively, across all application instances, all input topic partitions are consumed by all instances of the application.

You must provide a name for the table (more precisely, for the internal state store that backs the table). This is required for supporting Interactive Queries against the table. When a name is not provided the table will not queryable and an internal name will be provided for the state store.

```
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.kstream.GlobalKTable;

StreamsBuilder builder = new StreamsBuilder();

GlobalKTable<String, Long> wordCounts = builder.globalTable(
    "word-counts-input-topic",
    Materialized.<String, Long, KeyValueStore<Bytes, byte[]>>as(
      "word-counts-global-store" /* table/store name */)
      .withKeySerde(Serdes.String()) /* key serde */
      .withValueSerde(Serdes.Long()) /* value serde */
    );
```

You **must specify SerDes explicitly** if the key or value types of the records in the Kafka input topics do not match the configured default SerDes. For information about configuring default SerDes, available SerDes, and implementing your own custom SerDes see Data Types and Serialization.

Several variants of `globalTable` exist to e.g. specify explicit SerDes.

Transform a stream

The KStream and KTable interfaces support a variety of transformation operations. Each of these operations can be translated into one or more connected processors into the underlying processor topology. Since KStream and KTable are strongly typed, all of these transformation operations are defined as generic functions where users could specify the input and output data types.

Some KStream transformations may generate one or more KStream objects, for example: - `filter` and `map` on a KStream will generate another KStream - `branch` on KStream can generate multiple KStreams

Some others may generate a KTable object, for example an aggregation of a KStream also yields a KTable. This allows Kafka Streams to continuously update the computed value upon arrivals of late records after it has already been produced to the downstream transformation operators.

All KTable transformation operations can only generate another KTable. However, the Kafka Streams DSL does provide a special function that converts a KTable representation into a KStream. All of these transformation methods can be chained together to compose a complex processor topology.

These transformation operations are described in the following subsections:

- Stateless transformations
- Stateful transformations

Stateless transformations

Stateless transformations do not require state for processing and they do not require a state store associated with the stream processor. Kafka 0.11.0 and later allows you to materialize the result from a stateless `KTable` transformation. This allows the result to be queried through Interactive Queries. To materialize a `KTable`, each of the below stateless operationscan be augmented with an optional `queryableStoreName` argument.

Transformation	Description
Branch • KStream → KStream[]	Branch (or split) a `KStream` based on the supplied predicates into one or more `KStream` instances (details) Predicates are evaluated in order. A record is placed to one and only one output stream on the firs if the n-th predicate evaluates to true, the record is placed to n-th stream. If no predicate matches, record is dropped. Branching is useful, for example, to route records to different downstream topics. `KStream<String, Long> stream = ...;` `KStream<String, Long>[] branches = stream.branch(` `(key, value) -> key.startsWith("A"), /* first predicate */` `(key, value) -> key.startsWith("B"), /* second predicate */` `(key, value) -> true /* third predicate */` `);` `// KStream branches[0] contains all records whose keys start with "A"` `// KStream branches[1] contains all records whose keys start with "B"` `// KStream branches[2] contains all other records` `// Java 7 example: cf. `filter` for how to create `Predicate` instances`
Filter • KStream → KStream • KTable → KTable	Evaluates a boolean function for each element and retains those for which the function returns tru (KStream details, KTable details) `KStream<String, Long> stream = ...;` `// A filter that selects (keeps) only positive numbers`

Transformation

```
// Java 8+ example, using lambda expressions
KStream<String, Long> onlyPositives = stream.filter((key, value) -> value > 0);

// Java 7 example
KStream<String, Long> onlyPositives = stream.filter(
    new Predicate<String, Long>() {
      @Override
      public boolean test(String key, Long value) {
        return value > 0;
      }
    });
```

Inverse Filter

- KStream → KStream
- KTable → KTable

Evaluates a boolean function for each element and drops those for which the function returns tru
(KStream details, KTable details)

```
KStream<String, Long> stream = ...;

// An inverse filter that discards any negative numbers or zero
// Java 8+ example, using lambda expressions
KStream<String, Long> onlyPositives = stream.filterNot((key, value) -> value <= 0);

// Java 7 example
KStream<String, Long> onlyPositives = stream.filterNot(
    new Predicate<String, Long>() {
      @Override
      public boolean test(String key, Long value) {
        return value <= 0;
      }
    });
```

FlatMap

- KStream → KStream

Takes one record and produces zero, one, or more records. You can modify the record keys and v
including their types. (details)

Marks the stream for data re-partitioning: Applying a grouping or a join after `flatMap` will result
partitioning of the records. If possible use `flatMapValues` instead, which will not cause data re-
partitioning.

```
KStream<Long, String> stream = ...;
KStream<String, Integer> transformed = stream.flatMap(
     // Here, we generate two output records for each input record.
     // We also change the key and value types.
     // Example: (345L, "Hello") -> ("HELLO", 1000), ("hello", 9000)
    (key, value) -> {
      List<KeyValue<String, Integer>> result = new LinkedList<>();
      result.add(KeyValue.pair(value.toUpperCase(), 1000));
      result.add(KeyValue.pair(value.toLowerCase(), 9000));
      return result;
    }
  );

// Java 7 example: cf. `map` for how to create `KeyValueMapper` instances
```

FlatMap (values only)

- KStream → KStream

Takes one record and produces zero, one, or more records, while retaining the key of the original r
You can modify the record values and the value type. (details)

`flatMapValues` is preferable to `flatMap` because it will not cause data re-partitioning. However, y
cannot modify the key or key type like `flatMap` does.

```
// Split a sentence into words.
KStream<byte[], String> sentences = ...;
KStream<byte[], String> words = sentences.flatMapValues(value -> Arrays.asList(value.split("\\

// Java 7 example: cf. `mapValues` for how to create `ValueMapper` instances
```

Foreach

- KStream → void
- KStream → void
- KTable → void

Terminal operation. Performs a stateless action on each record. (details)

You would use `foreach` to cause *side effects* based on the input data (similar to `peek`) and then
further processing of the input data (unlike `peek`, which is not a terminal operation).

Note on processing guarantees: Any side effects of an action (such as writing to external system
not trackable by Kafka, which means they will typically not benefit from Kafka's processing guara

```
KStream<String, Long> stream = ...;
```

Transformation	

```
KStream<String, Long> stream = ...;

// Print the contents of the KStream to the local console.
// Java 8+ example, using lambda expressions
stream.foreach((key, value) -> System.out.println(key + " => " + value));

// Java 7 example
stream.foreach(
    new ForeachAction<String, Long>() {
      @Override
      public void apply(String key, Long value) {
        System.out.println(key + " => " + value);
      }
    });
```

GroupByKey

- KStream → KGroupedStream

Groups the records by the existing key. (details)

Grouping is a prerequisite for aggregating a stream or a table and ensures that data is properly pa ("keyed") for subsequent operations.

When to set explicit SerDes: Variants of `groupByKey` exist to override the configured default SerD your application, which **you must do** if the key and/or value types of the resulting `KGroupedStream` match the configured default SerDes.

> **Note**
>
> **Grouping vs. Windowing:** A related operation is windowing, which lets you control how to "sub- group" the grouped records *of the same key* into so-called *windows* for stateful operations suc windowed aggregations or windowed joins.

Causes data re-partitioning if and only if the stream was marked for re-partitioning. `groupByKey` preferable to `groupBy` because it re-partitions data only if the stream was already marked for re- partitioning. However, `groupByKey` does not allow you to modify the key or key type like `groupBy`

```
KStream<byte[], String> stream = ...;

// Group by the existing key, using the application's configured
// default serdes for keys and values.
KGroupedStream<byte[], String> groupedStream = stream.groupByKey();

// When the key and/or value types do not match the configured
// default serdes, we must explicitly specify serdes.
KGroupedStream<byte[], String> groupedStream = stream.groupByKey(
    Grouped.with(
      Serdes.ByteArray(), /* key */
      Serdes.String())     /* value */
  );
```

GroupBy

- KStream → KGroupedStream
- KTable → KGroupedTable

Groups the records by a *new* key, which may be of a different key type. When grouping a table, you also specify a new value and value type. `groupBy` is a shorthand for `selectKey(...).groupByKey()` (KStream details, KTable details)

Grouping is a prerequisite for aggregating a stream or a table and ensures that data is properly pa ("keyed") for subsequent operations.

When to set explicit SerDes: Variants of `groupBy` exist to override the configured default SerDes application, which **you must do** if the key and/or value types of the resulting `KGroupedStream` or `KGroupedTable` do not match the configured default SerDes.

> **Note**
>
> **Grouping vs. Windowing:** A related operation is windowing, which lets you control how to "sub- group" the grouped records *of the same key* into so-called *windows* for stateful operations suc windowed aggregations or windowed joins.

Always causes data re-partitioning: `groupBy` always causes data re-partitioning. If possible use `groupByKey` instead, which will re-partition data only if required.

```
KStream<byte[], String> stream = ...;
```

Transformation	
	<pre>KTable<byte[], String> table = ...; // Java 8+ examples, using lambda expressions // Group the stream by a new key and key type KGroupedStream<String, String> groupedStream = stream.groupBy((key, value) -> value, Grouped.with(Serdes.String(), /* key (note: type was modified) */ Serdes.String()) /* value */); // Group the table by a new key and key type, and also modify the value and value type. KGroupedTable<String, Integer> groupedTable = table.groupBy((key, value) -> KeyValue.pair(value, value.length()), Grouped.with(Serdes.String(), /* key (note: type was modified) */ Serdes.Integer()) /* value (note: type was modified) */); // Java 7 examples // Group the stream by a new key and key type KGroupedStream<String, String> groupedStream = stream.groupBy(new KeyValueMapper<byte[], String, String>>() { @Override public String apply(byte[] key, String value) { return value; } }, Grouped.with(Serdes.String(), /* key (note: type was modified) */ Serdes.String()) /* value */); // Group the table by a new key and key type, and also modify the value and value type. KGroupedTable<String, Integer> groupedTable = table.groupBy(new KeyValueMapper<byte[], String, KeyValue<String, Integer>>() { @Override public KeyValue<String, Integer> apply(byte[] key, String value) { return KeyValue.pair(value, value.length()); } }, Grouped.with(Serdes.String(), /* key (note: type was modified) */ Serdes.Integer()) /* value (note: type was modified) */);</pre>
Map • KStream → KStream	Takes one record and produces one record. You can modify the record key and value, including th types. (details) **Marks the stream for data re-partitioning:** Applying a grouping or a join after `map` will result in re partitioning of the records. If possible use `mapValues` instead, which will not cause data re-partitio <pre>KStream<byte[], String> stream = ...; // Java 8+ example, using lambda expressions // Note how we change the key and the key type (similar to `selectKey`) // as well as the value and the value type. KStream<String, Integer> transformed = stream.map((key, value) -> KeyValue.pair(value.toLowerCase(), value.length())); // Java 7 example KStream<String, Integer> transformed = stream.map(new KeyValueMapper<byte[], String, KeyValue<String, Integer>>() { @Override public KeyValue<String, Integer> apply(byte[] key, String value) { return new KeyValue<>(value.toLowerCase(), value.length()); } });</pre>
Map (values only) • KStream → KStream • KTable → KTable	Takes one record and produces one record, while retaining the key of the original record. You can the record value and the value type. (KStream details, KTable details) `mapValues` is preferable to `map` because it will not cause data re-partitioning. However, it does nc you to modify the key or key type like `map` does. Note that it is possible though to get read-only ac the input record key if you use `ValueMapperWithKey` instead of `ValueMapper`. <pre>KStream<byte[], String> stream = ...; // Java 8+ example, using lambda expressions KStream<byte[], String> uppercased = stream.mapValues(value -> value.toUpperCase());</pre>

Transformation	

```
// Java 7 example
KStream<byte[], String> uppercased = stream.mapValues(
    new ValueMapper<String>() {
      @Override
      public String apply(String s) {
        return s.toUpperCase();
      }
    });
```

Peek

- KStream → KStream

Performs a stateless action on each record, and returns an unchanged stream. (details)

You would use `peek` to cause *side effects* based on the input data (similar to `foreach`) and *con processing* the input data (unlike `foreach`, which is a terminal operation). `peek` returns the input as-is; if you need to modify the input stream, use `map` or `mapValues` instead.

`peek` is helpful for use cases such as logging or tracking metrics or for debugging and troublesh

Note on processing guarantees: Any side effects of an action (such as writing to external system not trackable by Kafka, which means they will typically not benefit from Kafka's processing guara

```
KStream<byte[], String> stream = ...;

// Java 8+ example, using lambda expressions
KStream<byte[], String> unmodifiedStream = stream.peek(
    (key, value) -> System.out.println("key=" + key + ", value=" + value));

// Java 7 example
KStream<byte[], String> unmodifiedStream = stream.peek(
    new ForeachAction<byte[], String>() {
      @Override
      public void apply(byte[] key, String value) {
        System.out.println("key=" + key + ", value=" + value);
      }
    });
```

Print

- KStream → void

Terminal operation. Prints the records to `System.out` or into a file. (details)

Calling `print(Printed.toSysOut())` is the same as calling `foreach((key, value) -> System.out.println(key + ", " + value))`

```
KStream<byte[], String> stream = ...;
// print to sysout
stream.print(Printed.toSysOut());

// print to file with a custom label
stream.print(Printed.toFile("streams.out").withLabel("streams"));
```

SelectKey

- KStream → KStream

Assigns a new key -- possibly of a new key type -- to each record. (details)

Calling `selectKey(mapper)` is the same as calling `map((key, value) -> mapper(key, value), value)`

Marks the stream for data re-partitioning: Applying a grouping or a join after `selectKey` will resu partitioning of the records.

```
KStream<byte[], String> stream = ...;

// Derive a new record key from the record's value.  Note how the key type changes, too.
// Java 8+ example, using lambda expressions
KStream<String, String> rekeyed = stream.selectKey((key, value) -> value.split(" ")[0])

// Java 7 example
KStream<String, String> rekeyed = stream.selectKey(
    new KeyValueMapper<byte[], String, String>() {
      @Override
      public String apply(byte[] key, String value) {
        return value.split(" ")[0];
      }
    });
```

Table to Stream

- KTable →

Get the changelog stream of this table. (details)

```
KTable<byte[], String> table = ...;
```

KStream **Transformation**	

```
// Also, a variant of `toStream` exists that allows you
// to select a new key for the resulting stream.
KStream<byte[], String> stream = table.toStream();
```

Stateful transformations

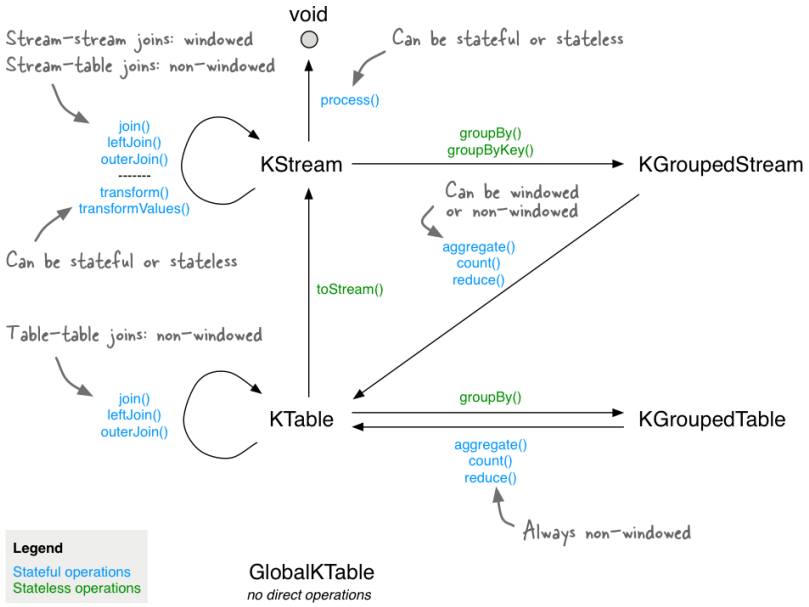
Stateful transformations depend on state for processing inputs and producing outputs and require a state store associated with the stream processor. For example, in aggregating operations, a windowing state store is used to collect the latest aggregation results per window. In join operations, a windowing state store is used to collect all of the records received so far within the defined window boundary.

Note, that state stores are fault-tolerant. In case of failure, Kafka Streams guarantees to fully restore all state stores prior to resuming the processing. See Fault Tolerance for further information.

Available stateful transformations in the DSL include:

- Aggregating
- Joining
- Windowing (as part of aggregations and joins)
- Applying custom processors and transformers, which may be stateful, for Processor API integration

The following diagram shows their relationships:



Stateful transformations in the DSL.

Here is an example of a stateful application: the WordCount algorithm.

WordCount example in Java 8+, using lambda expressions (seeWordCountLambdaIntegrationTest for the full code):

```
// Assume the record values represent lines of text.  For the sake of this example, you can ignore
// whatever may be stored in the record keys.
KStream<String, String> textLines = ...;

KStream<String, Long> wordCounts = textLines
    // Split each text line, by whitespace, into words.  The text lines are the record
    // values, i.e. you can ignore whatever data is in the record keys and thus invoke
    // `flatMapValues` instead of the more generic `flatMap`.
    .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
    // Group the stream by word to ensure the key of the record is the word.
    .groupBy((key, word) -> word)
    // Count the occurrences of each word (record key).
    //
    // This will change the stream type from `KGroupedStream<String, String>` to
    // `KTable<String, Long>` (word -> count).
    .count()
    // Convert the `KTable<String, Long>` into a `KStream<String, Long>`.
    .toStream();
```

WordCount example in Java 7:

```
// Code below is equivalent to the previous Java 8+ example above.
KStream<String, String> textLines = ...;

KStream<String, Long> wordCounts = textLines
    .flatMapValues(new ValueMapper<String, Iterable<String>>() {
        @Override
        public Iterable<String> apply(String value) {
            return Arrays.asList(value.toLowerCase().split("\\W+"));
        }
    })
    .groupBy(new KeyValueMapper<String, String, String>() {
        @Override
        public String apply(String key, String word) {
            return word;
        }
    })
    .count()
    .toStream();
```

Aggregating

After records are grouped by key via `groupByKey` or `groupBy` -- and thus represented as either a `KGroupedStream` or a `KGroupedTable`, they can be aggregated via an operation such as `reduce`. Aggregations are key-based operations, which means that they always operate over records (notably record values) of the same key. You can perform aggregations on windowed or non-windowed data.

Important

To support fault tolerance and avoid undesirable behavior, the initializer and aggregator must be stateless. The aggregation results should be passed in the return value of the initializer and aggregator. Do not use class member variables because that data can potentially get lost in case of failure.

Transformation	Description
Aggregate • KGroupedStream → KTable • KGroupedTable → KTable	**Rolling aggregation.** Aggregates the values of (non-windowed) records by the grouped key. Aggre and allows, for example, the aggregate value to have a different type than the input values. (KGrou details) When aggregating a *grouped stream*, you must provide an initializer (e.g., `aggValue = 0`) and an " `aggValue + curValue`). When aggregating a *grouped table*, you must provide a "subtractor" aggreg Several variants of `aggregate` exist, see Javadocs for details. `KGroupedStream<byte[], String> groupedStream = ...;` `KGroupedTable<byte[], String> groupedTable = ...;` `// Java 8+ examples, using lambda expressions` `// Aggregating a KGroupedStream (note how the value type changes from String to Long)` `KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(` `() -> 0L, /* initializer */` `(aggKey, newValue, aggValue) -> aggValue + newValue.length(), /* adder */` `Materialized.as("aggregated-stream-store") /* state store name */` `.withValueSerde(Serdes.Long()); /* serde for aggregate value */`

Transformation	

```
// Aggregating a KGroupedTable (note how the value type changes from String to Long)
KTable<byte[], Long> aggregatedTable = groupedTable.aggregate(
    () -> 0L, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue.length(), /* adder */
    (aggKey, oldValue, aggValue) -> aggValue - oldValue.length(), /* subtractor */
    Materialized.as("aggregated-table-store") /* state store name */
 .withValueSerde(Serdes.Long()) /* serde for aggregate value */


// Java 7 examples

// Aggregating a KGroupedStream (note how the value type changes from String to Long)
KTable<byte[], Long> aggregatedStream = groupedStream.aggregate(
    new Initializer<Long>() { /* initializer */
      @Override
      public Long apply() {
        return 0L;
      }
    },
    new Aggregator<byte[], String, Long>() { /* adder */
      @Override
      public Long apply(byte[] aggKey, String newValue, Long aggValue) {
        return aggValue + newValue.length();
      }
    },
    Materialized.as("aggregated-stream-store")
        .withValueSerde(Serdes.Long());

// Aggregating a KGroupedTable (note how the value type changes from String to Long)
KTable<byte[], Long> aggregatedTable = groupedTable.aggregate(
    new Initializer<Long>() { /* initializer */
      @Override
      public Long apply() {
        return 0L;
      }
    },
    new Aggregator<byte[], String, Long>() { /* adder */
      @Override
      public Long apply(byte[] aggKey, String newValue, Long aggValue) {
        return aggValue + newValue.length();
      }
    },
    new Aggregator<byte[], String, Long>() { /* subtractor */
      @Override
      public Long apply(byte[] aggKey, String oldValue, Long aggValue) {
        return aggValue - oldValue.length();
      }
    },
    Materialized.as("aggregated-stream-store")
        .withValueSerde(Serdes.Long());
```

Detailed behavior of `KGroupedStream`:

- Input records with `null` keys are ignored.
- When a record key is received for the first time, the initializer is called (and called before the ac
- Whenever a record with a non-`null` value is received, the adder is called.

Detailed behavior of `KGroupedTable`:

- Input records with `null` keys are ignored.
- When a record key is received for the first time, the initializer is called (and called before the ac contrast to `KGroupedStream`, over time the initializer may be called more than once for a key as tombstone records for that key (see below).
- When the first non-`null` value is received for a key (e.g., INSERT), then only the adder is calle
- When subsequent non-`null` values are received for a key (e.g., UPDATE), then (1) the subtract stored in the table and (2) the adder is called with the new value of the input record that was ju for the subtractor and adder is not defined.
- When a tombstone record -- i.e. a record with a `null` value -- is received for a key (e.g., DELET Note that, whenever the subtractor returns a `null` value itself, then the corresponding key is r that happens, any next input record for that key will trigger the initializer again.

See the example at the bottom of this section for a visualization of the aggregation semantics.

Aggregate (windowed)

- KGroupedStream → KTable

Windowed aggregation. Aggregates the values of records, per window, by the grouped key. Aggre and allows, for example, the aggregate value to have a different type than the input values. (TimeW SessionWindowedKStream details)

You must provide an initializer (e.g., `aggValue = 0`), "adder" aggregator (e.g., `aggValue + curValue`

Transformation

based on sessions, you must additionally provide a "session merger" aggregator (e.g., `mergedAggV`

Description

The windowed `aggregate` turns a `TimeWindowedKStream<K, V>` or `SessionWindowdKStream<K, V>` into a `KTable<Windowed<K>, V>`.

Several variants of `aggregate` exist, see Javadocs for details.

```
import java.time.Duration;

KGroupedStream<String, Long> groupedStream = ...;

// Java 8+ examples, using lambda expressions

// Aggregating with time-based windowing (here: with 5-minute tumbling windows)
KTable<Windowed<String>, Long> timeWindowedAggregatedStream = groupedStream.windowedBy(TimeWir
    .aggregate(
      () -> 0L, /* initializer */
      (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */
      Materialized.<String, Long, WindowStore<Bytes, byte[]>>as("time-windowed-aggregated-stre
        .withValueSerde(Serdes.Long())); /* serde for aggregate value */

// Aggregating with session-based windowing (here: with an inactivity gap of 5 minutes)
KTable<Windowed<String>, Long> sessionizedAggregatedStream = groupedStream.windowedBy(SessionW
    aggregate(
     () -> 0L, /* initializer */
     (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */
     (aggKey, leftAggValue, rightAggValue) -> leftAggValue + rightAggValue, /* session merger
     Materialized.<String, Long, SessionStore<Bytes, byte[]>>as("sessionized-aggregated-stream
        .withValueSerde(Serdes.Long())); /* serde for aggregate value */

// Java 7 examples

// Aggregating with time-based windowing (here: with 5-minute tumbling windows)
KTable<Windowed<String>, Long> timeWindowedAggregatedStream = groupedStream.windowedBy(TimeWir
    .aggregate(
        new Initializer<Long>() { /* initializer */
            @Override
            public Long apply() {
                return 0L;
            }
        },
        new Aggregator<String, Long, Long>() { /* adder */
            @Override
            public Long apply(String aggKey, Long newValue, Long aggValue) {
                return aggValue + newValue;
            }
        },
        Materialized.<String, Long, WindowStore<Bytes, byte[]>>as("time-windowed-aggregated-st
          .withValueSerde(Serdes.Long()));

// Aggregating with session-based windowing (here: with an inactivity gap of 5 minutes)
KTable<Windowed<String>, Long> sessionizedAggregatedStream = groupedStream.windowedBy(SessionW
    aggregate(
        new Initializer<Long>() { /* initializer */
            @Override
            public Long apply() {
                return 0L;
            }
        },
        new Aggregator<String, Long, Long>() { /* adder */
            @Override
            public Long apply(String aggKey, Long newValue, Long aggValue) {
                return aggValue + newValue;
            }
        },
        new Merger<String, Long>() { /* session merger */
            @Override
            public Long apply(String aggKey, Long leftAggValue, Long rightAggValue) {
                return rightAggValue + leftAggValue;
            }
        },
        Materialized.<String, Long, SessionStore<Bytes, byte[]>>as("sessionized-aggregated-str
          .withValueSerde(Serdes.Long()));
```

Detailed behavior:

- The windowed aggregate behaves similar to the rolling aggregate described above. The additi
  per window.
- Input records with `null` keys are ignored in general.
- When a record key is received for the first time for a given window, the initializer is called (and
- Whenever a record with a non-`null` value is received for a given window, the adder is called. (
  Kafka 0.11.0.0, the adder is currently also called for `null` values. You can work around this, fo
  `null` values prior to grouping the stream.)
- When using session windows: the session merger is called whenever two sessions are being r

Transformation	Description
Count • KGroupedStream → KTable • KGroupedTable → KTable	See the example at the bottom of this section for a visualization of the aggregation semantics. **Rolling aggregation.** Counts the number of records by the grouped key. (KGroupedStream details Several variants of `count` exist, see Javadocs for details. ``` KGroupedStream<String, Long> groupedStream = ...; KGroupedTable<String, Long> groupedTable = ...; // Counting a KGroupedStream KTable<String, Long> aggregatedStream = groupedStream.count(); // Counting a KGroupedTable KTable<String, Long> aggregatedTable = groupedTable.count(); ``` Detailed behavior for `KGroupedStream`: • Input records with `null` keys or values are ignored. Detailed behavior for `KGroupedTable`: • Input records with `null` keys are ignored. Records with `null` values are not ignored but inter corresponding key, which indicate the deletion of the key from the table.
Count (windowed) • KGroupedStream → KTable	**Windowed aggregation.** Counts the number of records, per window, by the grouped key. (TimeWi SessionWindowedKStream details) The windowed `count` turns a `TimeWindowedKStream<K, V>` or `SessionWindowedKStream<K, V>` into a Several variants of `count` exist, see Javadocs for details. ``` import java.time.Duration; KGroupedStream<String, Long> groupedStream = ...; // Counting a KGroupedStream with time-based windowing (here: with 5-minute tumbling windows) KTable<Windowed<String>, Long> aggregatedStream = groupedStream.windowedBy(TimeWindows.of(Duration.ofMinutes(5))) /* time-based window */ .count(); // Counting a KGroupedStream with session-based windowing (here: with 5-minute inactivity gap KTable<Windowed<String>, Long> aggregatedStream = groupedStream.windowedBy(SessionWindows.with(Duration.ofMinutes(5))) /* session window */ .count(); ``` Detailed behavior: • Input records with `null` keys or values are ignored. (Note: As a result of a known bug in Kafka are not ignored yet. You can work around this, for example, by manually filtering out `null` valu
Reduce • KGroupedStream → KTable • KGroupedTable → KTable	**Rolling aggregation.** Combines the values of (non-windowed) records by the grouped key. The cu the last reduced value, and a new reduced value is returned. The result value type cannot be chan (KGroupedStream details, KGroupedTable details) When reducing a *grouped stream*, you must provide an "adder" reducer (e.g., `aggValue + curValue` you must additionally provide a "subtractor" reducer (e.g., `aggValue - oldValue`). Several variants of `reduce` exist, see Javadocs for details. ``` KGroupedStream<String, Long> groupedStream = ...; KGroupedTable<String, Long> groupedTable = ...; // Java 8+ examples, using lambda expressions // Reducing a KGroupedStream KTable<String, Long> aggregatedStream = groupedStream.reduce((aggValue, newValue) -> aggValue + newValue /* adder */); // Reducing a KGroupedTable KTable<String, Long> aggregatedTable = groupedTable.reduce((aggValue, newValue) -> aggValue + newValue, /* adder */ ``` (aggValue, oldValue) -> aggValue - oldValue /* subtractor */);

Transformation

```
    (aggValue, oldValue) -> aggValue - oldValue /* subtractor */);

// Java 7 examples

// Reducing a KGroupedStream
KTable<String, Long> aggregatedStream = groupedStream.reduce(
    new Reducer<Long>() { /* adder */
      @Override
      public Long apply(Long aggValue, Long newValue) {
        return aggValue + newValue;
      }
    });

// Reducing a KGroupedTable
KTable<String, Long> aggregatedTable = groupedTable.reduce(
    new Reducer<Long>() { /* adder */
      @Override
      public Long apply(Long aggValue, Long newValue) {
        return aggValue + newValue;
      }
    },
    new Reducer<Long>() { /* subtractor */
      @Override
      public Long apply(Long aggValue, Long oldValue) {
        return aggValue - oldValue;
      }
    });
```

Detailed behavior for `KGroupedStream`:

- Input records with `null` keys are ignored in general.
- When a record key is received for the first time, then the value of that record is used as the init
- Whenever a record with a non-`null` value is received, the adder is called.

Detailed behavior for `KGroupedTable`:

- Input records with `null` keys are ignored in general.
- When a record key is received for the first time, then the value of that record is used as the init contrast to `KGroupedStream`, over time this initialization step may happen more than once for a input tombstone records for that key (see below).
- When the first non-`null` value is received for a key (e.g., INSERT), then only the adder is calle
- When subsequent non-`null` values are received for a key (e.g., UPDATE), then (1) the subtrac stored in the table and (2) the adder is called with the new value of the input record that was ju for the subtractor and adder is not defined.
- When a tombstone record -- i.e. a record with a `null` value -- is received for a key (e.g., DELET Note that, whenever the subtractor returns a `null` value itself, then the corresponding key is r that happens, any next input record for that key will re-initialize its aggregate value.

See the example at the bottom of this section for a visualization of the aggregation semantics.

Reduce (windowed)

- KGroupedStream → KTable

Windowed aggregation. Combines the values of records, per window, by the grouped key. The cu the last reduced value, and a new reduced value is returned. Records with `null` key or value are i be changed, unlike `aggregate`. (TimeWindowedKStream details, SessionWindowedKStream detai

The windowed `reduce` turns a turns a `TimeWindowedKStream<K, V>` or a `SessionWindowedKStream<K, KTable<Windowed<K>, V>`.

Several variants of `reduce` exist, see Javadocs for details.

```
import java.time.Duration;
KGroupedStream<String, Long> groupedStream = ...;

// Java 8+ examples, using lambda expressions

// Aggregating with time-based windowing (here: with 5-minute tumbling windows)
KTable<Windowed<String>, Long> timeWindowedAggregatedStream = groupedStream.windowedBy(
  TimeWindows.of(Duration.ofMinutes(5)) /* time-based window */)
  .reduce(
    (aggValue, newValue) -> aggValue + newValue /* adder */
  );

// Aggregating with session-based windowing (here: with an inactivity gap of 5 minutes)
KTable<Windowed<String>, Long> sessionzedAggregatedStream = groupedStream.windowedBy(
  SessionWindows.with(Duration.ofMinutes(5))) /* session window */
  .reduce(
    (aggValue, newValue) -> aggValue + newValue /* adder */
```

Transformation	

```
  );


// Java 7 examples

// Aggregating with time-based windowing (here: with 5-minute tumbling windows)
KTable<Windowed<String>, Long> timeWindowedAggregatedStream = groupedStream.windowedBy(
  TimeWindows.of(Duration.ofMinutes(5)) /* time-based window */)
  .reduce(
    new Reducer<Long>() { /* adder */
      @Override
      public Long apply(Long aggValue, Long newValue) {
        return aggValue + newValue;
      }
    });

// Aggregating with session-based windowing (here: with an inactivity gap of 5 minutes)
KTable<Windowed<String>, Long> timeWindowedAggregatedStream = groupedStream.windowedBy(
  SessionWindows.with(Duration.ofMinutes(5))) /* session window */
  .reduce(
    new Reducer<Long>() { /* adder */
      @Override
      public Long apply(Long aggValue, Long newValue) {
        return aggValue + newValue;
      }
    });
```

Detailed behavior:

- The windowed reduce behaves similar to the rolling reduce described above. The additional tw
  window.
- Input records with `null` keys are ignored in general.
- When a record key is received for the first time for a given window, then the value of that recor
  value.
- Whenever a record with a non-`null` value is received for a given window, the adder is called. (
  Kafka 0.11.0.0, the adder is currently also called for `null` values. You can work around this, fo
  `null` values prior to grouping the stream.)

See the example at the bottom of this section for a visualization of the aggregation semantics.

Example of semantics for stream aggregations: A `KGroupedStream` → `KTable` example is shown below. The streams and the table are initially empty. Bold font is used in the column for "KTable `aggregated`" to highlight changed state. An entry such as `(hello, 1)` denotes a record with key `hello` and value `1`. To improve the readability of the semantics table you can assume that all records are processed in timestamp order.

```
// Key: word, value: count
KStream<String, Integer> wordCounts = ...;

KGroupedStream<String, Integer> groupedStream = wordCounts
    .groupByKey(Grouped.with(Serdes.String(), Serdes.Integer()));

KTable<String, Integer> aggregated = groupedStream.aggregate(
    () -> 0, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */
    Materialized.<String, Long, KeyValueStore<Bytes, byte[]>as("aggregated-stream-store" /* state store name */)
      .withKeySerde(Serdes.String()) /* key serde */
      .withValueSerde(Serdes.Integer()); /* serde for aggregate value */
```

Note

Impact of record caches: For illustration purposes, the column "KTable `aggregated`" below shows the table's state changes over time in a very granular way. In practice, you would observe state changes in such a granular way only when record caches are disabled (default: enabled). When record caches are enabled, what might happen for example is that the output results of the rows with timestamps 4 and 5 would be compacted, and there would only be a single state update for the key `kafka` in the KTable (here: from `(kafka, 1)` directly to `(kafka, 3)`. Typically, you should only disable record caches for testing or debugging purposes -- under normal circumstances it is better to leave record caches enabled.

	KStream `wordCounts`		KGroupedStream `groupedStream`		KTable `aggregated`
Timestamp	**Input record**	**Grouping**	**Initializer**	**Adder**	**State**
1	(hello, 1)	(hello, 1)	0 (for hello)	(hello, 0 + 1)	**(hello, 1)**
2	(kafka, 1)	(kafka, 1)	0 (for kafka)	(kafka, 0 + 1)	(hello, 1) **(kafka, 1)**
3	(streams, 1)	(streams, 1)	0 (for streams)	(streams, 0 + 1)	(hello, 1) (kafka, 1) **(streams, 1)**
4	(kafka, 1)	(kafka, 1)		(kafka, 1 + 1)	(hello, 1) (kafka, **2**) (streams, 1)
5	(kafka, 1)	(kafka, 1)		(kafka, 2 + 1)	(hello, 1) (kafka, **3**) (streams, 1)
6	(streams, 1)	(streams, 1)		(streams, 1 + 1)	(hello, 1) (kafka, 3) (streams, **2**)

Example of semantics for table aggregations: A `KGroupedTable` → `KTable` example is shown below. The tables are initially empty. Bold font is used in the column for "KTable `aggregated`" to highlight changed state. An entry such as `(hello, 1)` denotes a record with key `hello` and value `1`. To improve the readability of the semantics table you can assume that all records are processed in timestamp order.

```
// Key: username, value: user region (abbreviated to "E" for "Europe", "A" for "Asia")
KTable<String, String> userProfiles = ...;

// Re-group `userProfiles`.  Don't read too much into what the grouping does:
// its prime purpose in this example is to show the *effects* of the grouping
// in the subsequent aggregation.
KGroupedTable<String, Integer> groupedTable = userProfiles
    .groupBy((user, region) -> KeyValue.pair(region, user.length()), Serdes.String(), Serdes.Integer());

KTable<String, Integer> aggregated = groupedTable.aggregate(
    () -> 0, /* initializer */
    (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */
    (aggKey, oldValue, aggValue) -> aggValue - oldValue, /* subtractor */
    Materialized.<String, Long, KeyValueStore<Bytes, byte[]>as("aggregated-table-store" /* state store name */)
      .withKeySerde(Serdes.String()) /* key serde */
      .withValueSerde(Serdes.Integer()); /* serde for aggregate value */
```

Note

Impact of record caches: For illustration purposes, the column "KTable `aggregated`" below shows the table's state changes over time in a very granular way. In practice, you would observe state changes in such a granular way only when record caches are disabled (default: enabled). When record caches are enabled, what might happen for example is that the output results of the rows with timestamps 4 and 5 would be compacted, and there would only be a single state update for the key `kafka` in the KTable (here: from `(kafka 1)` directly to `(kafka, 3)`. Typically, you should only disable record caches for testing or debugging purposes -- under normal circumstances it is better to leave record caches enabled.

	KTable `userProfiles`			KGroupedTable `groupedTable`			KTable `aggregated`
Timestamp	**Input record**	**Interpreted as**	**Grouping**	**Initializer**	**Adder**	**Subtractor**	**State**
1	(alice, E)	INSERT alice	(E, 5)	0 (for E)	(E, 0 + 5)		**(E, 5)**
2	(bob, A)	INSERT bob	(A, 3)	0 (for A)	(A, 0 + 3)		**(A, 3)** (E, 5)
3	(charlie, A)	INSERT charlie	(A, 7)		(A, 3 + 7)		(A, **10**) (E, 5)
4	(alice, A)	UPDATE alice	(A, 5)		(A, 10 + 5)	(E, 5 - 5)	(A, **15**) (E, **0**)
5	(charlie, null)	DELETE charlie	(null, 7)			(A, 15 - 7)	(A, **8**) (E, 0)
6	(null, E)	*ignored*					(A, 8) (E, 0)
7	(bob, E)	UPDATE bob	(E, 3)		(E, 0 + 3)	(A, 8 - 3)	(A, **5**) (E, **3**)

Joining

Streams and tables can also be joined. Many stream processing applications in practice are coded as streaming joins. For example, applications backing an online shop might need to access multiple, updating database tables (e.g. sales prices, inventory, customer information) in order to enrich a new data record (e.g. customer transaction) with context information. That is, scenarios where you need to perform table lookups at very large scale and with a low processing latency. Here, a popular pattern is to make the information in the databases available in Kafka through so-called *change data capture* in combination with Kafka's Connect API, and then implementing applications that leverage the Streams API to perform very fast and efficient local joins of such tables and streams, rather than requiring the application to make a query to a remote database over the network for each record. In this example, the KTable concept in Kafka Streams would enable you to track the latest state (e.g., snapshot) of each table in a local state store, thus greatly reducing the processing latency as well as reducing the load of the remote databases when doing such streaming joins.

The following join operations are supported, see also the diagram in the overview section of Stateful Transformations. Depending on the operands, joins are either windowed joins or non-windowed joins.

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN	Demo application
KStream-to-KStream	Windowed	Supported	Supported	Supported	StreamToStreamJoinIntegrationTest
KTable-to-KTable	Non-windowed	Supported	Supported	Supported	TableToTableJoinIntegrationTest
KStream-to-KTable	Non-windowed	Supported	Supported	Not Supported	StreamToTableJoinIntegrationTest
KStream-to-GlobalKTable	Non-windowed	Supported	Supported	Not Supported	GlobalKTablesExample
KTable-to-GlobalKTable	N/A	Not Supported	Not Supported	Not Supported	N/A

Each case is explained in more detail in the subsequent sections.

Join co-partitioning requirements

Input data must be co-partitioned when joining. This ensures that input records with the same key, from both sides of the join, are delivered to the same stream task during processing. **It is the responsibility of the user to ensure data co-partitioning when joining**

Tip

If possible, consider using global tables (`GlobalKTable`) for joining because they do not require data co-partitioning.

The requirements for data co-partitioning are:

- The input topics of the join (left side and right side) must have the **same number of partitions**
- All applications that *write* to the input topics must have the **same partitioning strategy** so that records with the same key are delivered to same partition number. In other words, the keyspace of the input data must be distributed across partitions in the same manner. This means that, for example, applications that use Kafka's Java Producer API must use the same partitioner (cf. the producer setting `"partitioner.class"` aka `ProducerConfig.PARTITIONER_CLASS_CONFIG`), and applications that use the Kafka's Streams API must use the same `StreamPartitioner` for operations such as `KStream#to()`. The good news is that, if you happen to use the default partitioner-related settings across all applications, you do not need to worry about the partitioning strategy.

Why is data co-partitioning required? Because KStream-KStream, KTable-KTable, and KStream-KTable joins are performed based on the keys of records (e.g., `leftRecord.key == rightRecord.key`), it is required that the input streams/tables of a join are co-partitioned by key.

The only exception are KStream-GlobalKTable joins. Here, co-partitioning is it not required because *all* partitions of the `GlobalKTable`'s underlying changelog stream are made available to each `KafkaStreams` instance, i.e. each instance has a full copy of the changelog stream. Further, a `KeyValueMapper` allows for non-key based joins from the `KStream` to the `GlobalKTable`.

Note

Kafka Streams partly verifies the co-partitioning requirement: During the partition assignment step, i.e. at runtime, Kafka Streams verifies whether the number of partitions for both sides of a join are the same. If they are not, a `TopologyBuilderException` (runtime exception) is being thrown. Note that Kafka Streams cannot verify whether the partitioning strategy matches between the input streams/tables of a join -- it is up to the user to ensure that this is the case.

Ensuring data co-partitioning: If the inputs of a join are not co-partitioned yet, you must ensure this manually. You may follow a procedure such as outlined below.

1. Identify the input KStream/KTable in the join whose underlying Kafka topic has the smaller number of partitions. Let's call this stream/table "SMALLER", and the other side of the join "LARGER". To learn about the number of partitions of a Kafka topic you can use, for example, the CLI tool `bin/kafka-topics` with the `--describe` option.
2. Pre-create a new Kafka topic for "SMALLER" that has the same number of partitions as "LARGER". Let's call this new topic "repartitioned-topic-for-smaller". Typically, you'd use the CLI tool `bin/kafka-topics` with the `--create` option for this.
3. Within your application, re-write the data of "SMALLER" into the new Kafka topic. You must ensure that, when writing the data with `to` or `through`, the same partitioner is used as for "LARGER".
   - If "SMALLER" is a KStream: `KStream#to("repartitioned-topic-for-smaller")`.
   - If "SMALLER" is a KTable: `KTable#to("repartitioned-topic-for-smaller")`.
4. Within your application, re-read the data in "repartitioned-topic-for-smaller" into a new KStream/KTable.
   - If "SMALLER" is a KStream: `StreamsBuilder#stream("repartitioned-topic-for-smaller")`.
   - If "SMALLER" is a KTable: `StreamsBuilder#table("repartitioned-topic-for-smaller")`.
5. Within your application, perform the join between "LARGER" and the new stream/table.

KStream-KStream Join

KStream-KStream joins are alwayswindowed joins, because otherwise the size of the internal state store used to perform the join -- e.g., a sliding window or "buffer" -- would grow indefinitely. For stream-stream joins it's important to highlight that a new input record on one side will produce a join output *for each* matching record on the other side, and there can be*multiple* such matching records in a given join window (cf. the row with timestamp 15 in the join semantics table below, for example).

Join output records are effectively created as follows, leveraging the user-supplied `ValueJoiner`:

```
KeyValue<K, LV> leftRecord = ...;
KeyValue<K, RV> rightRecord = ...;
ValueJoiner<LV, RV, JV> joiner = ...;

KeyValue<K, JV> joinOutputRecord = KeyValue.pair(
    leftRecord.key, /* by definition, leftRecord.key == rightRecord.key */
    joiner.apply(leftRecord.value, rightRecord.value)
  );
```

Transformation	Description
Inner Join (windowed) • (KStream, KStream) → KStream	Performs an INNER JOIN of this stream with another stream. Even though this operation is windowed, the joined stream will be of type `KStream<K, ...>` rather than `KStream<Windowed<K>, ...>`. (details) **Data must be co-partitioned**: The input data for both sides must be co-partitioned. **Causes data re-partitioning of a stream if and only if the stream was marked for re-partitioning (if both are marked, both are re-partitioned).** Several variants of `join` exists, see the Javadocs for details. `import java.time.Duration;` `KStream<String, Long> left = ...;` `KStream<String, Double> right = ...;` `// Java 8+ example, using lambda expressions` `KStream<String, String> joined = left.join(right,` `  (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue, /* ValueJoiner */` `  JoinWindows.of(Duration.ofMinutes(5)),` `  Joined.with(` `  Serdes.String(), /* key */` `  Serdes.Long(), /* left value */` `  Serdes.Double()) /* right value */` `);` `// Java 7 example` `KStream<String, String> joined = left.join(right,` `  new ValueJoiner<Long, Double, String>() {` `  @Override` `  public String apply(Long leftValue, Double rightValue) {` `  return "left=" + leftValue + ", right=" + rightValue;` `  }` `  },` `  JoinWindows.of(Duration.ofMinutes(5)),`

Transformation	

```
    Joined.with(
      Serdes.String(), /* key */
      Serdes.Long(),   /* left value */
      Serdes.Double())  /* right value */
  );
```

Detailed behavior:

- The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`, and *window-based*, i.e. two input records are joined if and only if their timestamps are "close" to each other as defined by the user-supplied `JoinWindows`, i.e. the window defines an additional join predicate over the record timestamps.
- The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records.
  - Input records with a `null` key or a `null` value are ignored and do not trigger the join.

See the semantics overview at the bottom of this section for a detailed description.

Left Join (windowed)

- (KStream, KStream) → KStream

Performs a LEFT JOIN of this stream with another stream. Even though this operation is windowed, the joined stream will be of type `KStream<K, ...>` rather than `KStream<Windowed<K>, ...>`. (details)

Data must be co-partitioned: The input data for both sides must be co-partitioned.

Causes data re-partitioning of a stream if and only if the stream was marked for re-partitioning (if both are marked, both are re-partitioned).

Several variants of `leftJoin` exists, see the Javadocs for details.

```
import java.time.Duration;

KStream<String, Long> left = ...;
KStream<String, Double> right = ...;

// Java 8+ example, using lambda expressions
KStream<String, String> joined = left.leftJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue, /* ValueJoiner */

    JoinWindows.of(Duration.ofMinutes(5)),
    Joined.with(
      Serdes.String(), /* key */
      Serdes.Long(),   /* left value */
      Serdes.Double())  /* right value */
  );

// Java 7 example
KStream<String, String> joined = left.leftJoin(right,
    new ValueJoiner<Long, Double, String>() {
      @Override
      public String apply(Long leftValue, Double rightValue) {
        return "left=" + leftValue + ", right=" + rightValue;
      }
    },
    JoinWindows.of(Duration.ofMinutes(5)),
    Joined.with(
      Serdes.String(), /* key */
      Serdes.Long(),   /* left value */
      Serdes.Double())  /* right value */
  );
```

Detailed behavior:

- The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`, and *window-based*, i.e. two input records are joined if and only if their timestamps are "close" to each other as defined by the user-supplied `JoinWindows`, i.e. the window defines an additional join predicate over the record timestamps.
- The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records.
  - Input records with a `null` key or a `null` value are ignored and do not trigger the join.
- For each input record on the left side that does not have any match on the right side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null)`; this explains the row with timestamp=3 in the table below, which lists `[A, null]` in the LEFT JOIN column.

Transformation	Description
Outer Join (windowed) • (KStream, KStream) → KStream	See the semantics overview at the bottom of this section for a detailed description. Performs an OUTER JOIN of this stream with another stream. Even though this operation is windowed, the joined stream will be of type `KStream<K, ...>` rather than `KStream<Windowed<K>, ...>`. (details) **Data must be co-partitioned**: The input data for both sides must be co-partitioned. **Causes data re-partitioning of a stream if and only if the stream was marked for re-partitioning (if both are marked, both are re-partitioned).** Several variants of `outerJoin` exists, see the Javadocs for details. `import java.time.Duration;` `KStream<String, Long> left = ...;` `KStream<String, Double> right = ...;` `// Java 8+ example, using lambda expressions` `KStream<String, String> joined = left.outerJoin(right,` `(leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue, /* ValueJoiner */` `JoinWindows.of(Duration.ofMinutes(5)),` `Joined.with(` `Serdes.String(), /* key */` `Serdes.Long(), /* left value */` `Serdes.Double()) /* right value */` `);` `// Java 7 example` `KStream<String, String> joined = left.outerJoin(right,` `new ValueJoiner<Long, Double, String>() {` `@Override` `public String apply(Long leftValue, Double rightValue) {` `return "left=" + leftValue + ", right=" + rightValue;` `}` `},` `JoinWindows.of(Duration.ofMinutes(5)),` `Joined.with(` `Serdes.String(), /* key */` `Serdes.Long(), /* left value */` `Serdes.Double()) /* right value */` `);` Detailed behavior: • The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`, and *window-based*, i.e. two input records are joined if and only if their timestamps are "close" to each other as defined by the user-supplied `JoinWindows`, i.e. the window defines an additional join predicate over the record timestamps. • The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records. ◦ Input records with a `null` key or a `null` value are ignored and do not trigger the join. • For each input record on one side that does not have any match on the other side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null)` or `ValueJoiner#apply(null, rightRecord.value)`, respectively; this explains the row with timestamp=3 in the table below, which lists `[A, null]` in the OUTER JOIN column (unlike LEFT JOIN, `[null, x]` is possible, too, but no such example is shown in the table). See the semantics overview at the bottom of this section for a detailed description.

Semantics of stream-stream joins: The semantics of the various stream-stream join variants are explained below. To improve the readability of the table, assume that (1) all records have the same key (and thus the key in the table is omitted), (2) all records belong to a single join window, and (3) all records are processed in timestamp order. The columns INNER JOIN, LEFT JOIN, and OUTER JOIN denote what is passed as arguments to the user-supplied ValueJoiner for the `join`, `leftJoin`, and `outerJoin` methods, respectively, whenever a new input record is received on either side of the join. An empty table cell denotes that the `ValueJoiner` is not called at all.

Timestamp	Left (KStream)	Right (KStream)	(INNER) JOIN	LEFT JOIN	OUTER JOIN
1	null				
2		null			
3	A			[A, null]	[A, null]
4		a	[A, a]	[A, a]	[A, a]
5	B		[B, a]	[B, a]	[B, a]
6		b	[A, b], [B, b]	[A, b], [B, b]	[A, b], [B, b]
7	null				
8		null			
9	C		[C, a], [C, b]	[C, a], [C, b]	[C, a], [C, b]
10		c	[A, c], [B, c], [C, c]	[A, c], [B, c], [C, c]	[A, c], [B, c], [C, c]
11		null			
12	null				
13		null			
14		d	[A, d], [B, d], [C, d]	[A, d], [B, d], [C, d]	[A, d], [B, d], [C, d]
15	D		[D, a], [D, b], [D, c], [D, d]	[D, a], [D, b], [D, c], [D, d]	[D, a], [D, b], [D, c], [D, d]

KTable-KTable Join

KTable-KTable joins are always *non-windowed* joins. They are designed to be consistent with their counterparts in relational databases. The changelog streams of both KTables are materialized into local state stores to represent the latest snapshot of their table duals. The join result is a new KTable that represents the changelog stream of the join operation.

Join output records are effectively created as follows, leveraging the user-supplied `ValueJoiner`:

```
KeyValue<K, LV> leftRecord = ...;
KeyValue<K, RV> rightRecord = ...;
ValueJoiner<LV, RV, JV> joiner = ...;

KeyValue<K, JV> joinOutputRecord = KeyValue.pair(
    leftRecord.key, /* by definition, leftRecord.key == rightRecord.key */
    joiner.apply(leftRecord.value, rightRecord.value)
  );
```

Transformation	Description
Inner Join • (KTable, KTable) → KTable	Performs an INNER JOIN of this table with another table. The result is an ever-updating KTable that represents the "current" result of the join. (details) **Data must be co-partitioned**: The input data for both sides must be co-partitioned. `KTable<String, Long> left = ...;` `KTable<String, Double> right = ...;` `// Java 8+ example, using lambda expressions` `KTable<String, String> joined = left.join(right,` `  (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue /* ValueJoiner */` `  );`

Transformation	
	`// Java 7 example` `KTable<String, String> joined = left.join(right,` `  new ValueJoiner<Long, Double, String>() {` `  @Override` `  public String apply(Long leftValue, Double rightValue) {` `  return "left=" + leftValue + ", right=" + rightValue;` `  }` `  });` Detailed behavior: • The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`. • The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records. ◦ Input records with a `null` key are ignored and do not trigger the join. ◦ Input records with a `null` value are interpreted as *tombstones* for the corresponding key, which indicate the deletion of the key from the table. Tombstones do not the join. When an input tombstone is received, then an output tombstone is forwarded directly to the join result KTable if required (i.e. only if the corresponding key actually exists already in the join result KTable). See the semantics overview at the bottom of this section for a detailed description.
Left Join • (KTable, KTable) → KTable	Performs a LEFT JOIN of this table with another table. (details) **Data must be co-partitioned**: The input data for both sides must be co-partitioned. `KTable<String, Long> left = ...;` `KTable<String, Double> right = ...;` `// Java 8+ example, using lambda expressions` `KTable<String, String> joined = left.leftJoin(right,` `  (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue /* ValueJoiner */` `  );` `// Java 7 example` `KTable<String, String> joined = left.leftJoin(right,` `  new ValueJoiner<Long, Double, String>() {` `  @Override` `  public String apply(Long leftValue, Double rightValue) {` `  return "left=" + leftValue + ", right=" + rightValue;` `  }` `  });` Detailed behavior: • The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`. • The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records. ◦ Input records with a `null` key are ignored and do not trigger the join. ◦ Input records with a `null` value are interpreted as *tombstones* for the corresponding key, which indicate the deletion of the key from the table. Tombstones do not trigger the join. When an input tombstone is received, then an output tombstone is forwarded directly to the join result KTable if required (i.e. only if the corresponding key actually exists already in the join result KTable). • For each input record on the left side that does not have any match on the right side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null)`; this explains the row with timestamp=3 in the table below, which lists `[A, null]` in the LEFT JOIN column. See the semantics overview at the bottom of this section for a detailed description.
Outer Join • (KTable, KTable) → KTable	Performs an OUTER JOIN of this table with another table. (details) **Data must be co-partitioned**: The input data for both sides must be co-partitioned. `KTable<String, Long> left = ...;` `KTable<String, Double> right = ...;` `// Java 8+ example, using lambda expressions`

Transformation	

```
KTable<String, String> joined = left.outerJoin(right,
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue /* ValueJoiner */
  );

// Java 7 example
KTable<String, String> joined = left.outerJoin(right,
    new ValueJoiner<Long, Double, String>() {
      @Override
      public String apply(Long leftValue, Double rightValue) {
        return "left=" + leftValue + ", right=" + rightValue;
      }
    });
```

Detailed behavior:

- The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`.
- The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records.
  - Input records with a `null` key are ignored and do not trigger the join.
  - **Input records with a `null` value are interpreted as *tombstones* for the corresponding key, which indicate the deletion of the key from the table. Tombstones do not**

    trigger the join. When an input tombstone is received, then an output tombstone is forwarded directly to the join result KTable if required (i.e. only if the corresponding key actually exists already in the join result KTable).

- For each input record on one side that does not have any match on the other side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null)` or `ValueJoiner#apply(null, rightRecord.value)`, respectively; this explains the rows with timestamp=3 and timestamp=7 in the table below, which list `[A, null]` and `[null, b]`, respectively, in the OUTER JOIN column.

See the semantics overview at the bottom of this section for a detailed description.

Semantics of table-table joins: The semantics of the various table-table join variants are explained below. To improve the readability of the table, you can assume that (1) all records have the same key (and thus the key in the table is omitted) and that (2) all records are processed in timestamp order. The columns INNER JOIN, LEFT JOIN, and OUTER JOIN denote what is passed as arguments to the user-supplied ValueJoiner for the `join`, `leftJoin`, and `outerJoin` methods, respectively, whenever a new input record is received on either side of the join. An empty table cell denotes that the `ValueJoiner` is not called at all.

Timestamp	Left (KTable)	Right (KTable)	(INNER) JOIN	LEFT JOIN	OUTER JOIN
1	null (tombstone)				
2		null (tombstone)			
3	A			[A, null]	[A, null]
4		a	[A, a]	[A, a]	[A, a]
5	B		[B, a]	[B, a]	[B, a]
6		b	[B, b]	[B, b]	[B, b]
7	null (tombstone)		null (tombstone)	null (tombstone)	[null, b]
8		null (tombstone)			null (tombstone)
9	C			[C, null]	[C, null]
10		c	[C, c]	[C, c]	[C, c]
11		null (tombstone)	null (tombstone)	[C, null]	[C, null]
12	null (tombstone)			null (tombstone)	null (tombstone)
13		null (tombstone)			
14		d			[null, d]
15	D		[D, d]	[D, d]	[D, d]

KStream-KTable Join

KStream-KTable joins are always *non-windowed* joins. They allow you to perform *table lookups* against a KTable (changelog stream) upon receiving a new record from the KStream (record stream). An example use case would be to enrich a stream of user activities (KStream) with the latest user profile information (KTable).

Join output records are effectively created as follows, leveraging the user-supplied `ValueJoiner`:

```
KeyValue<K, LV> leftRecord = ...;
KeyValue<K, RV> rightRecord = ...;
ValueJoiner<LV, RV, JV> joiner = ...;

KeyValue<K, JV> joinOutputRecord = KeyValue.pair(
    leftRecord.key, /* by definition, leftRecord.key == rightRecord.key */
    joiner.apply(leftRecord.value, rightRecord.value)
  );
```

Transformation	Description
Inner Join • (KStream,	Performs an INNER JOIN of this stream with the table, effectively doing a table lookup. (details) **Data must be co-partitioned**: The input data for both sides must be co-partitioned.

Transformation	Description
KTable) → KStream	**Causes data re-partitioning of the stream if and only if the stream was marked for re-partitioning.**
	Several variants of `join` exists, see the Javadocs for details. `KStream<String, Long> left = ...;` `KTable<String, Double> right = ...;` `// Java 8+ example, using lambda expressions` `KStream<String, String> joined = left.join(right,` `  (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue, /* ValueJoiner */` `  Joined.keySerde(Serdes.String()) /* key */` `  .withValueSerde(Serdes.Long()) /* left value */` `);` `// Java 7 example` `KStream<String, String> joined = left.join(right,` `  new ValueJoiner<Long, Double, String>() {` `  @Override` `  public String apply(Long leftValue, Double rightValue) {` `  return "left=" + leftValue + ", right=" + rightValue;` `  }` `  },` `  Joined.keySerde(Serdes.String()) /* key */` `  .withValueSerde(Serdes.Long()) /* left value */` `);` Detailed behavior: • The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`. • The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records. ◦ Only input records for the left side (stream) trigger the join. Input records for the right side (table) update only the internal right-side join state. ◦ Input records for the stream with a `null` key or a `null` value are ignored and do not trigger the join. ◦ Input records for the table with a `null` value are interpreted as *tombstones* for the corresponding key, which indicate the deletion of the key from the table. Tombstones do not trigger the join. See the semantics overview at the bottom of this section for a detailed description.
Left Join • (KStream, KTable) → KStream	Performs a LEFT JOIN of this stream with the table, effectively doing a table lookup. (details) **Data must be co-partitioned**: The input data for both sides must be co-partitioned. **Causes data re-partitioning of the stream if and only if the stream was marked for re-partitioning.** Several variants of `leftJoin` exists, see the Javadocs for details. `KStream<String, Long> left = ...;` `KTable<String, Double> right = ...;` `// Java 8+ example, using lambda expressions` `KStream<String, String> joined = left.leftJoin(right,` `  (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue, /* ValueJoiner */` `  Joined.keySerde(Serdes.String()) /* key */` `  .withValueSerde(Serdes.Long()) /* left value */` `);` `// Java 7 example` `KStream<String, String> joined = left.leftJoin(right,` `  new ValueJoiner<Long, Double, String>() {` `  @Override` `  public String apply(Long leftValue, Double rightValue) {` `  return "left=" + leftValue + ", right=" + rightValue;` `  }` `  },` `  Joined.keySerde(Serdes.String()) /* key */` `  .withValueSerde(Serdes.Long()) /* left value */` `);` Detailed behavior:

Transformation	Description
	The join is *key-based*, i.e. with the join predicate `leftRecord.key == rightRecord.key`. • The join will be triggered under the conditions listed below whenever new input is received. When it is triggered, the user-supplied `ValueJoiner` will be called to produce join output records. ◦ Only input records for the left side (stream) trigger the join. Input records for the right side (table) update only the internal right-side join state. ◦ Input records for the stream with a `null` key or a `null` value are ignored and do not trigger the join. ◦ **Input records for the table with a `null` value are interpreted as *tombstones* for the corresponding key, which indicate the deletion of the key from the table.** Tombstones do not trigger the join. • For each input record on the left side that does not have any match on the right side, the `ValueJoiner` will be called with `ValueJoiner#apply(leftRecord.value, null)`; this explains the row with timestamp=3 in the table below, which lists `[A, null]` in the LEFT JOIN column. See the semantics overview at the bottom of this section for a detailed description.

Semantics of stream-table joins: The semantics of the various stream-table join variants are explained below. To improve the readability of the table we assume that (1) all records have the same key (and thus we omit the key in the table) and that (2) all records are processed in timestamp order. The columns INNER JOIN and LEFT JOIN denote what is passed as arguments to the user-supplied ValueJoiner for the `join` and `leftJoin` methods, respectively, whenever a new input record is received on either side of the join. An empty table cell denotes that the `ValueJoiner` is not called at all.

Timestamp	Left (KStream)	Right (KTable)	(INNER) JOIN	LEFT JOIN
1	null			
2		null (tombstone)		
3	A			[A, null]
4		a		
5	B		[B, a]	[B, a]
6		b		
7	null			
8		null (tombstone)		
9	C			[C, null]
10		c		
11		null		
12	null			
13		null		
14		d		
15	D		[D, d]	[D, d]

KStream-GlobalKTable Join

KStream-GlobalKTable joins are always *non-windowed* joins. They allow you to perform *table lookups* against a GlobalKTable (entire changelog stream) upon receiving a new record from the KStream (record stream). An example use case would be "star queries" or "star joins", where you would enrich a stream of user activities (KStream) with the latest user profile information (GlobalKTable) and further context information (further GlobalKTables).

At a high-level, KStream-GlobalKTable joins are very similar to KStream-KTable joins. However, global tables provide you with much more flexibility at the some expense when compared to partitioned tables:

- They do not require data co-partitioning.
- They allow for efficient "star joins"; i.e., joining a large-scale "facts" stream against "dimension" tables
- They allow for joining against foreign keys; i.e., you can lookup data in the table not just by the keys of records in the stream, but also by data in the record values.
- They make many use cases feasible where you must work on heavily skewed data and thus suffer from hot partitions.
- They are often more efficient than their partitioned KTable counterpart when you need to perform multiple joins in succession.

Join output records are effectively created as follows, leveraging the user-supplied `ValueJoiner`:

```
KeyValue<K, LV> leftRecord = ...;
KeyValue<K, RV> rightRecord = ...;
ValueJoiner<LV, RV, JV> joiner = ...;

KeyValue<K, JV> joinOutputRecord = KeyValue.pair(
    leftRecord.key, /* by definition, leftRecord.key == rightRecord.key */
    joiner.apply(leftRecord.value, rightRecord.value)
  );
```

Transformation	Description
Inner Join • (KStream, GlobalKTable) → KStream	Performs an INNER JOIN of this stream with the global table, effectively doing a table lookup. (detail The `GlobalKTable` is fully bootstrapped upon (re)start of a `KafkaStreams` instance, which means the all the data in the underlying topic that is available at the time of the startup. The actual data process bootstrapping has completed. **Causes data re-partitioning of the stream if and only if the stream was marked for re-partitioning.** Detailed behavior: • The join is indirectly *key-based*, i.e. with the join predicate `KeyValueMapper#apply(leftRecord.key, leftRecord.value) == rightRecord.key`. • The join will be triggered under the conditions listed below whenever new input is received. When supplied `ValueJoiner` will be called to produce join output records. ◦ Only input records for the left side (stream) trigger the join. Input records for the right side (tal

```
KStream<String, Long> left = ...;
GlobalKTable<Integer, Double> right = ...;

// Java 8+ example, using lambda expressions
KStream<String, String> joined = left.join(right,
    (leftKey, leftValue) -> leftKey.length(), /* derive a (potentially) new key by which to looku
    (leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue /* ValueJoiner */
  );

// Java 7 example
KStream<String, String> joined = left.join(right,
    new KeyValueMapper<String, Long, Integer>() { /* derive a (potentially) new key by which to

      @Override
      public Integer apply(String key, Long value) {
        return key.length();
      }
    },
    new ValueJoiner<Long, Double, String>() {
      @Override
      public String apply(Long leftValue, Double rightValue) {
        return "left=" + leftValue + ", right=" + rightValue;
      }
    });
```

Transformation	Description
	right-side join state. • Input records for the stream with a `null` key or a `null` value are ignored and do not trigger t • **Input records for the table with a `null` value are interpreted as *tombstones*, which indicate the deletion o Tombstones do not trigger the** join.
Left Join • (KStream, GlobalKTable) → KStream	Performs a LEFT JOIN of this stream with the global table, effectively doing a table lookup. (details) The `GlobalKTable` is fully bootstrapped upon (re)start of a `KafkaStreams` instance, which means the all the data in the underlying topic that is available at the time of the startup. The actual data process bootstrapping has completed. **Causes data re-partitioning of the stream if and only if the stream was marked for re-partitioning.** `KStream<String, Long> left = ...;` `GlobalKTable<Integer, Double> right = ...;` `// Java 8+ example, using lambda expressions` `KStream<String, String> joined = left.leftJoin(right,` `(leftKey, leftValue) -> leftKey.length(), /* derive a (potentially) new key by which to looku` `(leftValue, rightValue) -> "left=" + leftValue + ", right=" + rightValue /* ValueJoiner */` `);` `// Java 7 example` `KStream<String, String> joined = left.leftJoin(right,` `new KeyValueMapper<String, Long, Integer>() { /* derive a (potentially) new key by which to` `@Override` `public Integer apply(String key, Long value) {` `return key.length();` `}` `},` `new ValueJoiner<Long, Double, String>() {` `@Override` `public String apply(Long leftValue, Double rightValue) {` `return "left=" + leftValue + ", right=" + rightValue;` `}` `});` Detailed behavior: • The join is indirectly *key-based*, i.e. with the join predicate `KeyValueMapper#apply(leftRecord.key, leftRecord.value) == rightRecord.key`. • The join will be triggered under the conditions listed below whenever new input is received. When supplied `ValueJoiner` will be called to produce join output records. ◦ Only input records for the left side (stream) trigger the join. Input records for the right side (tal right-side join state. ◦ Input records for the stream with a `null` key or a `null` value are ignored and do not trigger t ◦ Input records for the table with a `null` value are interpreted as *tombstones*, which indicate th from the table. Tombstones do not trigger the join. • For each input record on the left side that does not have any match on the right side, the `ValueJo` `ValueJoiner#apply(leftRecord.value, null)`.

Semantics of stream-table joins: The join semantics are identical toKStream-KTable joins. The only difference is that, for KStream-GlobalKTable joins, the left input record is first "mapped" with a user-supplied `KeyValueMapper` into the table's keyspace prior to the table lookup.

Windowing

Windowing lets you control how to group records that have the same key for stateful operations such asaggregations or joins into so-called windows. Windows are tracked per record key.

Note

A related operation is grouping, which groups all records that have the same key to ensure that data is properly partitioned ("keyed") for subsequent operations. Once grouped, windowing allows you to further sub-group the records of a key.

For example, in join operations, a windowing state store is used to store all the records received so far within the defined window boundary. In aggregating operations, a windowing state store is used to store the latest aggregation results per window. Old records in the state store are purged after the specified window retention period. Kafka Streams guarantees to keep a window for at least this specified time; the default value is one day and can be changed via `Materialized#withRetention()`.

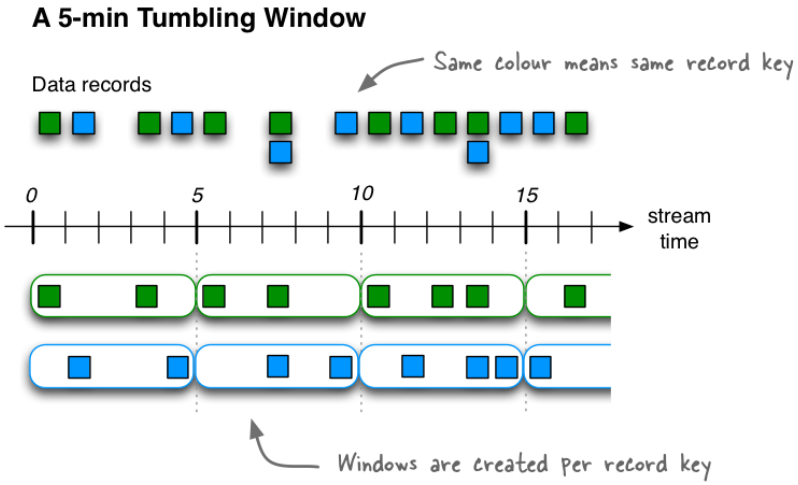
The DSL supports the following types of windows:

Window name	Behavior	Short description
Tumbling time window	Time-based	Fixed-size, non-overlapping, gap-less windows
Hopping time window	Time-based	Fixed-size, overlapping windows
Sliding time window	Time-based	Fixed-size, overlapping windows that work on differences between record timestamps
Session window	Session-based	Dynamically-sized, non-overlapping, data-driven windows

An example of implementing a custom time window is provided at the end of this section.

Tumbling time windows

Tumbling time windows are a special case of hopping time windows and, like the latter, are windows based on time intervals. They model fixed-size, non-overlapping, gap-less windows. A tumbling window is defined by a single property: the window's *size*. A tumbling window is a hopping window whose window size is equal to its advance interval. Since tumbling windows never overlap, a data record will belong to one and only one window.



*This diagram shows windowing a stream of data records with tumbling windows. Windows do not overlap because, by definition, the advance interval is identical to the window size. In this diagram the time numbers represent minutes; e.g. t=5 means "at the five-minute mark". In reality, the unit of time in Kafka Streams is milliseconds, which means the time numbers would need to be multiplied with 60 * 1,000 to convert from minutes to milliseconds (e.g. t=5 would become t=300,000).*

Tumbling time windows are *aligned to the epoch*, with the lower interval bound being inclusive and the upper bound being exclusive. "Aligned to the epoch" means that the first window starts at timestamp zero. For example, tumbling windows with a size of 5000ms have predictable window

boundaries `[0;5000),[5000;10000),...` --- and **not** `[1000;6000),[6000;11000),...` or even something "random" like `[1452;6452),[6452;11452),...`.

The following code defines a tumbling window with a size of 5 minutes:

```
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

// A tumbling time window with a size of 5 minutes (and, by definition, an implicit
// advance interval of 5 minutes).
Duration windowSizeMs = Duration.ofMinutes(5);
TimeWindows.of(windowSizeMs);

// The above is equivalent to the following code:
TimeWindows.of(windowSizeMs).advanceBy(windowSizeMs);
```

Counting example using tumbling windows:

```
// Key (String) is user ID, value (Avro record) is the page view event for that user.
// Such a data stream is often called a "clickstream".
KStream<String, GenericRecord> pageViews = ...;

// Count page views per window, per user, with tumbling windows of size 5 minutes
KTable<Windowed<String>, Long> windowedPageViewCounts = pageViews
    .groupByKey(Grouped.with(Serdes.String(), genericAvroSerde))
    .windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
    .count();
```

Hopping time windows

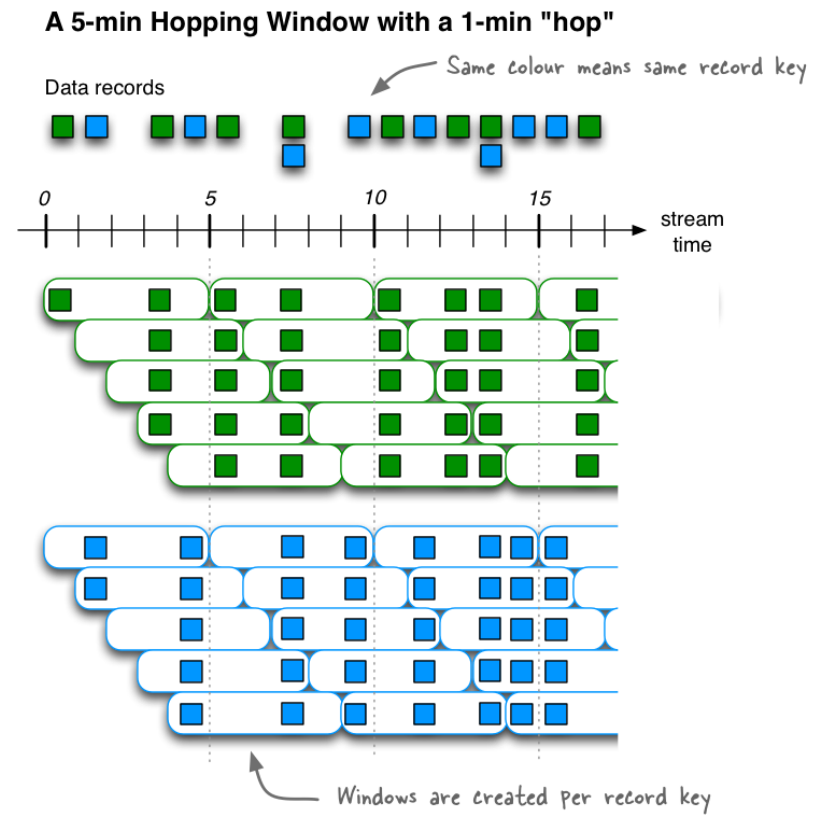
Hopping time windows are windows based on time intervals. They model fixed-sized, (possibly) overlapping windows. A hopping window is defined by two properties: the window's *size* and its *advance interval* (aka "hop"). The advance interval specifies by how much a window moves forward relative to the previous one. For example, you can configure a hopping window with a size 5 minutes and an advance interval of 1 minute. Since hopping windows can overlap -- and in general they do -- a data record may belong to more than one such window.

Note

Hopping windows vs. sliding windows: Hopping windows are sometimes called "sliding windows" in other stream processing tools. Kafka Streams follows the terminology in academic literature, where the semantics of sliding windows are different to those of hopping windows.

The following code defines a hopping window with a size of 5 minutes and an advance interval of 1 minute:

```
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

// A hopping time window with a size of 5 minutes and an advance interval of 1 minute.
// The window's name -- the string parameter -- is used to e.g. name the backing state store.
Duration windowSizeMs = Duration.ofMinutes(5);
Duration advanceMs =    Duration.ofMinutes(1);
TimeWindows.of(windowSizeMs).advanceBy(advanceMs);
```



*This diagram shows windowing a stream of data records with hopping windows. In this diagram the time numbers represent minutes; e.g. t=5 means "at the five-minute mark". In reality, the unit of time in Kafka Streams is milliseconds, which means the time numbers would need to be multiplied with 60 * 1,000 to convert from minutes to milliseconds (e.g. t=5 would become t=300,000).*

Hopping time windows are *aligned to the epoch*, with the lower interval bound being inclusive and the upper bound being exclusive. "Aligned to the epoch" means that the first window starts at timestamp zero. For example, hopping windows with a size of 5000ms and an advance interval ("hop") of 3000ms have predictable window boundaries `[0;5000),[3000;8000),...` — and **not** `[1000;6000),[4000;9000),...` or even something "random" like `[1452;6452),[4452;9452),...`.

Counting example using hopping windows:

```
// Key (String) is user ID, value (Avro record) is the page view event for that user.
// Such a data stream is often called a "clickstream".
KStream<String, GenericRecord> pageViews = ...;

// Count page views per window, per user, with hopping windows of size 5 minutes that advance every 1 minute
KTable<Windowed<String>, Long> windowedPageViewCounts = pageViews
    .groupByKey(Grouped.with(Serdes.String(), genericAvroSerde))
    .windowedBy(TimeWindows.of(Duration.ofMinutes(5).advanceBy(Duration.ofMinutes(1))))
    .count()
```

Unlike non-windowed aggregates that we have seen previously, windowed aggregates return a *windowed KTable* whose keys type is `Windowed<K>`. This is to differentiate aggregate values with the same key from different windows. The corresponding window instance and the embedded key can be retrieved as `Windowed#window()` and `Windowed#key()`, respectively.

Sliding time windows

Sliding windows are actually quite different from hopping and tumbling windows. In Kafka Streams, sliding windows are used only for join operations, and can be specified through the `JoinWindows` class.

A sliding window models a fixed-size window that slides continuously over the time axis; here, two data records are said to be included in the same window if (in the case of symmetric windows) the difference of their timestamps is within the window size. Thus, sliding windows are not aligned to the epoch, but to the data record timestamps. In contrast to hopping and tumbling windows, the lower and upper window time interval bounds of sliding windows are *both inclusive*.

Session Windows

Session windows are used to aggregate key-based events into so-called*sessions*, the process of which is referred to as*sessionization*. Sessions represent a **period of activity** separated by a defined**gap of inactivity** (or "idleness"). Any events processed that fall within the inactivity gap of any existing sessions are merged into the existing sessions. If an event falls outside of the session gap, then a new session will be created.

Session windows are different from the other window types in that:

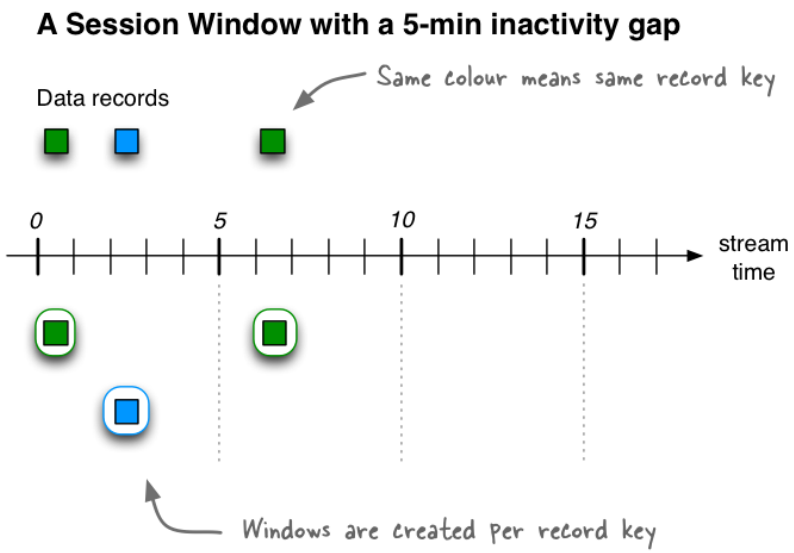
- all windows are tracked independently across keys -- e.g. windows of different keys typically have different start and end times
- their window sizes sizes vary -- even windows for the same key typically have different sizes

The prime area of application for session windows is**user behavior analysis**. Session-based analyses can range from simple metrics (e.g. count of user visits on a news website or social platform) to more complex metrics (e.g. customer conversion funnel and event flows).

The following code defines a session window with an inactivity gap of 5 minutes:

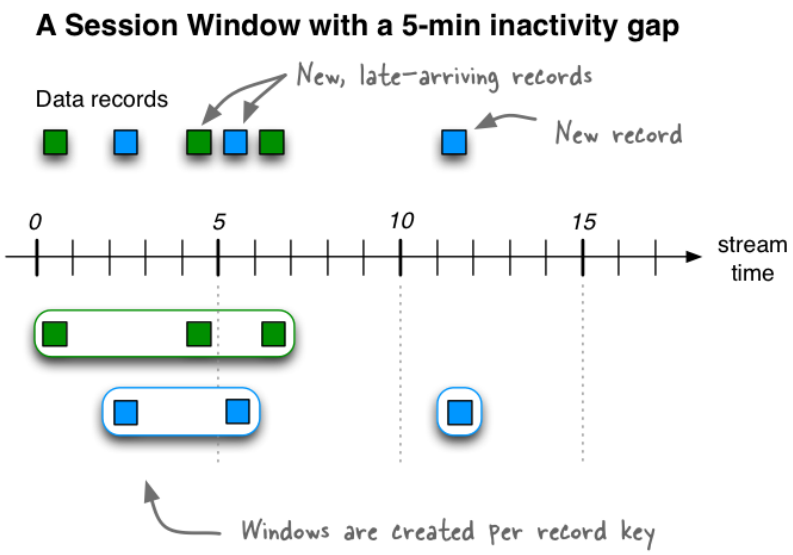
```
import java.time.Duration;
import org.apache.kafka.streams.kstream.SessionWindows;

// A session window with an inactivity gap of 5 minutes.
SessionWindows.with(Duration.ofMinutes(5));
```

Given the previous session window example, here's what would happen on an input stream of six records. When the first three records arrive (upper part of in the diagram below), we'd have three sessions (see lower part) after having processed those records: two for the green record key, with one session starting and ending at the 0-minute mark (only due to the illustration it looks as if the session goes from 0 to 1), and another starting and ending at the 6-minute mark; and one session for the blue record key, starting and ending at the 2-minute mark.



*Detected sessions after having received three input records: two records for the green record key at t=0 and t=6, and one record for the blue record key at t=2. In this diagram the time numbers represent minutes; e.g. t=5 means "at the five-minute mark". In reality, the unit of time in Kafka Streams is milliseconds, which means the time numbers would need to be multiplied with 60 * 1,000 to convert from minutes to milliseconds (e.g. t=5 would become t=300,000).*

If we then receive three additional records (including two late-arriving records), what would happen is that the two existing sessions for the green record key will be merged into a single session starting at time 0 and ending at time 6, consisting of a total of three records. The existing session for the blue record key will be extended to end at time 5, consisting of a total of two records. And, finally, there will be a new session for the blue key starting and ending at time 11.



Detected sessions after having received six input records. Note the two late-arriving data records at t=4 (green) and t=5 (blue), which lead to a merge of sessions and an extension of a session, respectively.

Counting example using session windows: Let's say we want to analyze reader behavior on a news website such as the New York Times, given a session definition of "As long as a person views (clicks on) another page at least once every 5 minutes (= inactivity gap), we consider this to be a single visit and thus a single, contiguous reading session of that person." What we want to compute off of this stream of input data is the number of page views per session.

```
// Key (String) is user ID, value (Avro record) is the page view event for that user.
// Such a data stream is often called a "clickstream".
KStream<String, GenericRecord> pageViews = ...;

// Count page views per session, per user, with session windows that have an inactivity gap of 5 minutes
KTable<Windowed<String>, Long> sessionizedPageViewCounts = pageViews
    .groupByKey(Grouped.with(Serdes.String(), genericAvroSerde))
    .windowedBy(SessionWindows.with(Duration.ofMinutes(5)))
    .count();
```

Window Final Results

In Kafka Streams, windowed computations update their results continuously. As new data arrives for a window, freshly computed results are emitted downstream. For many applications, this is ideal, since fresh results are always available, and Kafka Streams is designed to make programming continuous computations seamless. However, some applications need to take action **only** on the final result of a windowed computation. Common examples of this are sending alerts or delivering results to a system that doesn't support updates.

Suppose that you have an hourly windowed count of events per user. If you want to send an alert when a user has *less than* three events in an hour, you have a real challange. All users would match this condition at first, until they accrue enough events, so you can't simply send an alert when someone matches the condition; you have to wait until you know you won't see any more events for a particular window, and *then* send the alert.

Kafka Streams offers a clean way to define this logic: after defining your windowed computation, you can `suppress` the intermediate results, emitting the final count for each user when the window is **closed**.

For example:

```
KGroupedStream<UserId, Event> grouped = ...;
grouped
    .windowedBy(TimeWindows.of(Duration.ofHours(1)).grace(Duration.ofMinutes(10)))
    .count()
    .suppress(Suppressed.untilWindowCloses(unbounded()))
    .filter((windowedUserId, count) -> count < 3)
    .toStream()
    .foreach((windowedUserId, count) -> sendAlert(windowedUserId.window(), windowedUserId.key(), count));
```

The key parts of this program are:

`grace(Duration.ofMinutes(10))`

This allows us to bound the lateness of events the window will accept. For example, the 09:00 to 10:00 window will accept late-arriving records until 10:10, at which point, the window is **closed**.

`.suppress(Suppressed.untilWindowCloses(...))`

This configures the suppression operator to emit nothing for a window until it closes, and then emit the final result. For example, if use `U` gets 10 events between 09:00 and 10:10, the `filter` downstream of the suppression will get no events for the windowed ke `@09:00-10:00` until 10:10, and then it will get exactly one event with the value `10`. This is the final result of the windowed count.

`unbounded()`

This configures the buffer used for storing events until their windows close. Production code is able to put a cap on the amount of memory to use for the buffer, but this simple example creates a buffer with no upper bound.

One thing to note is that suppression is just like any other Kafka Streams operator, so you can build a topology with two branches emerging from the `count`, one suppressed, and one not, or even multiple differently configured suppressions. This allows you to apply suppressions where they are needed and otherwise rely on the default continuous update behavior.

For more detailed information, see the JavaDoc on the `Suppressed` config object and KIP-328.

Example: Custom Time Window

In addition to using the windows implementations provided with the Kafka Streams client library, you can extend the ava Windows abstract class to create custom time windows to suit your use cases.

To view a custom implementation of a daily window starting every day at 6pm, refer to streams/window example.

The example also shows a potential problem in dealing with time zones that have daylight savings time.

Applying processors and transformers (Processor API integration)

Beyond the aforementioned stateless and stateful transformations, you may also leverage the Processor API from the DSL. There are a number of scenarios where this may be helpful:

- **Customization:** You need to implement special, customized logic that is not or not yet available in the DSL.
- **Combining ease-of-use with full flexibility where it's needed:** Even though you generally prefer to use the expressiveness of the DSL, there are certain steps in your processing that require more flexibility and tinkering than the DSL provides. For example, only the Processor API provides access to a record's metadata such as its topic, partition, and offset information. However, you don't want to switch completely to the Processor API just because of that.
- **Migrating from other tools:** You are migrating from other stream processing technologies that provide an imperative API, and migrating some of your legacy code to the Processor API was faster and/or easier than to migrate completely to the DSL right away.

Transformation	Description
Process • KStream -> void	**Terminal operation.** Applies a `Processor` to each record. `process()` allows you to leverage the Processor API from the DSL. (details) This is essentially equivalent to adding the `Processor` via `Topology#addProcessor()` to your processor topology. An example is available in the javadocs.
Transform • KStream -> KStream	Applies a `Transformer` to each record. `transform()` allows you to leverage the Processor API from the DSL. (details) Each input record is transformed into zero, one, or more output records (similar to the stateless `flatMap`). The `Transformer` must return `null` for zero output. You can modify the record's key and value, including their types. **Marks the stream for data re-partitioning:** Applying a grouping or a join after `transform` will result in re-partitioning of the records. If possible use `transformValues` instead, which will not cause data re-partitioning. `transform` is essentially equivalent to adding the `Transformer` via `Topology#addProcessor()` to your processor topology. An example is available in the javadocs. Also, a full end-to-end demo is available at MixAndMatchLambdaIntegrationTest.
Transform (values only) • KStream -> KStream	Applies a `ValueTransformer` to each record, while retaining the key of the original record. `transformValues()` allows you to leverage the Processor API from the DSL. (details) Each input record is transformed into exactly one output record (zero output records or multiple output records are not possible). The `ValueTransformer` may return `null` as the new value for a record. `transformValues` is preferable to `transform` because it will not cause data re-partitioning. It is also possible to get read-only access to the input record key if you use `ValueTransformerWithKey` (provided via `ValueTransformerWithKeySupplier`) instead. `transformValues` is essentially equivalent to adding the `ValueTransformer` via `Topology#addProcessor()` to your processor topology. An example is available in the javadocs.

The following example shows how to leverage, via the `KStream#process()` method, a custom `Processor` that sends an email notification whenever a page view count reaches a predefined threshold.

First, we need to implement a custom stream processor, `PopularPageEmailAlert`, that implements the `Processor` interface:

```
// A processor that sends an alert message about a popular page to a configurable email address
public class PopularPageEmailAlert implements Processor<PageId, Long> {

  private final String emailAddress;
  private ProcessorContext context;

  public PopularPageEmailAlert(String emailAddress) {
    this.emailAddress = emailAddress;
  }

  @Override
  public void init(ProcessorContext context) {
    this.context = context;

    // Here you would perform any additional initializations such as setting up an email client.
  }

  @Override
  void process(PageId pageId, Long count) {
    // Here you would format and send the alert email.
    //
    // In this specific example, you would be able to include information about the page's ID and its view count
    // (because the class implements `Processor<PageId, Long>`).
  }

  @Override
  void close() {
    // Any code for clean up would go here.  This processor instance will not be used again after this call.
  }

}
```

Tip

Even though we do not demonstrate it in this example, a stream processor can access any available state stores by calling `ProcessorContext#getStateStore()`. Only such state stores are available that (1) have been named in the corresponding `KStream#process()` method call (note that this is a different method thar `Processor#process()`), plus (2) all global stores. Note that global stores do not need to be attached explicitly; however, they only allow for read-only access.

Then we can leverage the `PopularPageEmailAlert` processor in the DSL via `KStream#process`.

In Java 8+, using lambda expressions:

```
KStream<String, GenericRecord> pageViews = ...;

// Send an email notification when the view count of a page reaches one thousand.
pageViews.groupByKey()
         .count()
         .filter((PageId pageId, Long viewCount) -> viewCount == 1000)
         // PopularPageEmailAlert is your custom processor that implements the
         // `Processor` interface, see above.
         .process(() -> new PopularPageEmailAlert("alerts@yourcompany.com"));
```

In Java 7:

```
// Send an email notification when the view count of a page reaches one thousand.
pageViews.groupByKey().
         .count()
         .filter(
            new Predicate<PageId, Long>() {
              public boolean test(PageId pageId, Long viewCount) {
                return viewCount == 1000;
              }
            })
         .process(
           new ProcessorSupplier<PageId, Long>() {
             public Processor<PageId, Long> get() {
               // PopularPageEmailAlert is your custom processor that implements
               // the `Processor` interface, see above.
               return new PopularPageEmailAlert("alerts@yourcompany.com");
             }
           });
```

Controlling KTable emit rate

A KTable is logically a continuously updated table. These updates make their way to downstream operators whenever new data is available, ensuring that the whole computation is as fresh as possible. Most programs describe a series of logical transformations, and the update rate is not a factor in the program behavior.

In these cases, the rate of update is a performance concern, which is best addressed directly via the relevant configurations.

However, for some applications, the rate of update itself is an important semantic property.

Rather than achieving this as a side-effect of the record caches, you can directly impose a rate limit via the `KTable#suppress` operator.

For example:

```
KGroupedTable<String, String> groupedTable = ...;
groupedTable
    .count()
    .suppress(untilTimeLimit(Duration.ofMinutes(5), maxBytes(1_000_000L).emitEarlyWhenFull()))
    .toStream();
```

This configuration ensures that, downstream of `suppress`, each key is updated no more than once every 5 minutes (in stream time, not wall-clock time).

Note that the latest state for each key has to be buffered in memory for that 5-minute period. You have the option to control the maximum amount of memory to use for this buffer (in this case, 1MB). There is also an option to impose a limit in terms of number of records or to leave both limits unspecified.

Additionally, it is possible to choose what happens if the buffer fills up. This example takes a relaxed approach and just emits the oldest records before their 5-minute time limit to bring the buffer back down to size. Alternatively, you can choose to stop processing and shut the application down. This may seem extreme, but it gives you a guarantee that the 5-minute time limit will be absolutely enforced. After the application shuts down, you could allocate more memory for the buffer and resume processing. Emitting early is preferable for most applications.

For more detailed information, see the JavaDoc on the `Suppressed` config object and KIP-328.

Writing streams back to Kafka

Any streams and tables may be (continuously) written back to a Kafka topic. As we will describe in more detail below, the output data might be re-partitioned on its way to Kafka, depending on the situation.

Writing to Kafka	Description
To • KStream -> void	**Terminal operation.** Write the records to Kafka topic(s). (KStream details) When to provide serdes explicitly: • If you do not specify SerDes explicitly, the default SerDes from the configuration are used. • You **must specify SerDes explicitly** via the `Produced` class if the key and/or value types of the `KStream` do not match the configured default SerDes. • See Data Types and Serialization for information about configuring default SerDes, available SerDes, and implementing your own custom SerDes. A variant of `to` exists that enables you to specify how the data is produced by using a `Produced` instance to specify, for example, a `StreamPartitioner` that gives you control over how output records are distributed across the partitions of the output topic. Another variant of `to` enables you to dynamically choose which topic to send to for each

Writing to Kafka | Description	
	Another variant of `to` enables you to dynamically choose which topic to send to for each record via a `TopicNameExtractor` instance. `KStream<String, Long> stream = ...;` `KTable<String, Long> table = ...;` `// Write the stream to the output topic, using the configured default key` `// and value serdes of your `StreamsConfig`.` `stream.to("my-stream-output-topic");` `// Write the stream to the output topic, using explicit key and value serdes,` `// (thus overriding the defaults of your `StreamsConfig`).` `stream.to("my-stream-output-topic", Produced.with(Serdes.String(), Serdes.Long());` `// Write the stream to the output topics, the topic name is dynamically deteremined for` `// each record; also using explicit stream partitioner to determine which partition` `// of the topic to send to` `stream.to(` `  (key, value, recordContext) ->  // topicNameExtractor` `    if (myPattern.matcher(key).matches) {` `      "special-stream-output-topic"` `    } else {` `      "normal-stream-output-topic"` `    },` `  Produced.streamPartitioner(` `    (topic, key, value, numPartitions) ->` `      if (topic.equals("special-stream-output-topic")) {` `        specialHash(key, value, numPartitions)` `      } else {` `        md5Hash(key, value, numPartitions)` `      }` `  )` `);` **Causes data re-partitioning if any of the following conditions is true:** 1. If the output topic has a different number of partitions than the stream/table. 2. If the `KStream` was marked for re-partitioning. 3. If you provide a custom `StreamPartitioner` to explicitly control how to distribute the output records across the partitions of the output topic. 4. If the key of an output record is `null`.
Through • KStream -> KStream • KTable -> KTable	Write the records to a Kafka topic and create a new stream/table from that topic. Essentially a shorthand for `KStream#to()` followed by `StreamsBuilder#stream()`, same for tables. (KStream details) When to provide SerDes explicitly: • If you do not specify SerDes explicitly, the default SerDes from the configuration are used. • You **must specify SerDes explicitly** if the key and/or value types of the `KStream` or `KTable` do not match the configured default SerDes. • See Data Types and Serialization for information about configuring default SerDes, available SerDes, and implementing your own custom SerDes. A variant of `through` exists that enables you to specify how the data is produced by using a `Produced` instance to specify, for example, a `StreamPartitioner` that gives you control over how output records are distributed across the partitions of the output topic. `StreamsBuilder builder = ...;` `KStream<String, Long> stream = ...;` `KTable<String, Long> table = ...;` `// Variant 1: Imagine that your application needs to continue reading and processing` `// the records after they have been written to a topic via ``to()``. Here, one option` `// is to write to an output topic, then read from the same topic by constructing a` `// new stream from it, and then begin processing it (here: via `map`, for example).` `stream.to("my-stream-output-topic");` `KStream<String, Long> newStream = builder.stream("my-stream-output-topic").map(...);`  `// Variant 2 (better): Since the above is a common pattern, the DSL provides the` `// convenience method ``through`` that is equivalent to the code above.` `// Note that you may need to specify key and value serdes explicitly, which is` `// not shown in this simple example.` `KStream<String, Long> newStream = stream.through("user-clicks-topic").map(...);`

Writing to Kafka | Description	
	Causes data re-partitioning if any of the following conditions is true: 1. If the output topic has a different number of partitions than the stream/table. 2. If the `KStream` was marked for re-partitioning. 3. If you provide a custom `StreamPartitioner` to explicitly control how to distribute the output records across the partitions of the output topic. 4. If the key of an output record is `null`.

Note

When you want to write to systems other than Kafka: Besides writing the data back to Kafka, you can also apply a custom processor as a stream sink at the end of the processing to, for example, write to external databases. First, doing so is not a recommended pattern -- we strongly suggest to use the Kafka Connect API instead. However, if you do use such a sink processor, please be aware that it is now your responsibility to guarantee message delivery semantics when talking to such external systems (e.g., to retry on delivery failure or to prevent message duplication).

Kafka Streams DSL for Scala

Kafka Streams provides a Scala wrapper for the Java API to provide:

1. Better type inference in Scala.
2. Less boilerplate in application code.
3. The usual builder-style composition that developers get with the original Java API.
4. Implicit serializers and de-serializers leading to better abstraction and less verbosity.
5. Better type safety during compile time.

All functionality provided by Kafka Streams DSL for Scala are under the root package name o `org.apache.kafka.streams.scala`.

Many of the public facing types from the Java API are wrapped. The following Scala abstractions are available to the user:

- `org.apache.kafka.streams.scala.StreamsBuilder`
- `org.apache.kafka.streams.scala.kstream.KStream`
- `org.apache.kafka.streams.scala.kstream.KTable`
- `org.apache.kafka.streams.scala.kstream.KGroupedStream`
- `org.apache.kafka.streams.scala.kstream.KGroupedTable`
- `org.apache.kafka.streams.scala.kstream.SessionWindowedKStream`
- `org.apache.kafka.streams.scala.kstream.TimeWindowedKStream`

The library also has several utility abstractions and modules that the user needs to use for proper semantics.

- `org.apache.kafka.streams.scala.ImplicitConversions`: Class that brings into scope the implicit conversions between the Scala and Java classes.
- `org.apache.kafka.streams.scala.Serdes`: Class that contains core SerDes that can be imported as implicits and a helper to create custom SerDes. (see Implicit SerDes)

The library is cross-built with Scala 2.11 and 2.12. To reference the library compiled against Scala 2.11 add the following in your maver `pom.xml`:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams-scala_2.11</artifactId>
  <version>2.1.0</version>
</dependency>
```

To use the library compiled against Scala 2.12 replace the `artifactId` with `kafka-streams-scala_2.12`.

When using SBT then you can reference the correct library using the following:

```
libraryDependencies += "org.apache.kafka" %% "kafka-streams-scala" % "2.1.0"
```

Notes:

- There is an upstream dependency that causes trouble in SBT builds. This issue is present in `2.1.0` but is fixed in subsequent major, minor, and bugfix releases.

  If you must use an affected version, add an explicit dependency on the problematic library as a workaround:

  `2.1.0`:

  ```
  libraryDependencies += "javax.ws.rs" % "javax.ws.rs-api" % "2.1.1" artifacts(Artifact("javax.ws.rs-api", "jar", "jar"
  ))
  ```

  `(any later release)`:

  No workaround needed

Sample Usage

The library works by wrapping the original Java abstractions of Kafka Streams within a Scala wrapper object. All the Scala abstractions are named identically as the corresponding Java abstraction, but they reside in a different package of the library. For example, the Scala class `org.apache.kafka.streams.scala.StreamsBuilder` is a wrapper around `org.apache.kafka.streams.StreamsBuilder`, `org.apache.kafka.streams.scala.kstream.KStream` is a wrapper around `org.apache.kafka.streams.kstream.KStream`, and so on.

The net result is that the following code is structured just like using the Java API, but with fewer type annotations compared to using the Java API directly from Scala. The difference in type annotation usage is more obvious when given an example.

Here's an example of the classic WordCount program that uses the Scala `StreamsBuilder` that builds an instance of `KStream` which is a wrapper around Java `KStream`. Then we convert to a table and get a `KTable`, which, again is a wrapper around Java `KTable`.

```
import java.time.Duration
import java.util.Properties

import org.apache.kafka.streams.kstream.Materialized
import org.apache.kafka.streams.scala.ImplicitConversions._
import org.apache.kafka.streams.scala._
import org.apache.kafka.streams.scala.kstream._
import org.apache.kafka.streams.{KafkaStreams, StreamsConfig}

object WordCountApplication extends App {
  import Serdes._

  val props: Properties = {
    val p = new Properties()
    p.put(StreamsConfig.APPLICATION_ID_CONFIG, "wordcount-application")
    p.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "kafka-broker1:9092")
    p
  }

  val builder: StreamsBuilder = new StreamsBuilder
  val textLines: KStream[String, String] = builder.stream[String, String]("TextLinesTopic")
  val wordCounts: KTable[String, Long] = textLines
    .flatMapValues(textLine => textLine.toLowerCase.split("\\W+"))
    .groupBy((_, word) => word)
    .count(Materialized.as("counts-store"))
  wordCounts.toStream.to("WordsWithCountsTopic")

  val streams: KafkaStreams = new KafkaStreams(builder.build(), props)
  streams.start()

  sys.ShutdownHookThread {
     streams.close(Duration.ofSeconds(10))
  }
}
```

In the above code snippet, we don't have to provide any SerDes `Grouped`, `Produced`, `Consumed` or `Joined` explicitly. They will also not be dependent on any SerDes specified in the config. **In fact all SerDes specified in the config will be ignored by the Scala APIs** All SerDes and `Grouped`, `Produced`, `Consumed` or `Joined` will be handled through implicit SerDes as discussed later in the Implicit SerDes section. The complete independence from configuration based SerDes is what makes this library completely typesafe. Any missing instances of SerDes, `Grouped`, `Produced`, `Consumed` or `Joined` will be flagged as a compile time error.

Implicit SerDes

The library uses the power of Scala implicit parameters to avoid repetitively having to specify SerDes throughout the topology. As a user you can provide implicit SerDes or implicit values of `Grouped`, `Produced`, `Consumed` or `Joined` once and make your code less verbose.

The library also bundles all implicit SerDes of the commonly used types in `org.apache.kafka.streams.scala.Serdes`. Importing this class's members removes the need to specify serdes for any standard data type.

Here's an example:

```
// Serdes brings into scope pre-defined implicit SerDes
// that will set up all Grouped, Produced, Consumed and Joined instances.
// So all APIs below that accept Grouped, Produced, Consumed or Joined will
// get these instances automatically

import Serdes._
import org.apache.kafka.streams.scala.Serdes._
import org.apache.kafka.streams.scala.ImplicitConvertions._

val builder = new StreamsBuilder()

val userClicksStream: KStream[String, Long] = builder.stream(userClicksTopic)

val userRegionsTable: KTable[String, String] = builder.table(userRegionsTopic)

// The following code fragment does not have a single instance of Grouped,
// Produced, Consumed or Joined supplied explicitly.
// All of them are taken care of by the implicit SerDes imported by Serdes
val clicksPerRegion: KTable[String, Long] =
  userClicksStream
    .leftJoin(userRegionsTable)((clicks, region) => (if (region == null) "UNKNOWN" else region, clicks))
    .map((_, regionWithClicks) => regionWithClicks)
    .groupByKey
    .reduce(_ + _)

clicksPerRegion.toStream.to(outputTopic)
```

Quite a few things are going on in the above code snippet that may warrant a few lines of elaboration:

1. The code snippet does not depend on any config defined SerDes. In fact any SerDes defined as part of the config will be ignored.
2. All SerDes are picked up from the implicits in scope. Anc `import Serdes._` brings all necessary SerDes in scope.
3. Any needed SerDe not provided by the imported implicits would be a compile-time error.
4. The code is very tidy and focused on the actual transformation.

User-Defined SerDes

When the core SerDes are not enough and we need to define custom SerDes, the usage is exactly the same as above. Just define the implicit SerDes and start building the stream transformation. Here's an example with `AvroSerde`:

```
// domain object as a case class
case class UserClicks(clicks: Long)

// An implicit Serde implementation for the values we want to
// serialize as avro
implicit val userClicksSerde: Serde[UserClicks] = new AvroSerde

// Primitive SerDes
import Serdes._

// And then business as usual ..

val userClicksStream: KStream[String, UserClicks] = builder.stream(userClicksTopic)

val userRegionsTable: KTable[String, String] = builder.table(userRegionsTopic)

// Compute the total per region by summing the individual click counts per region.
val clicksPerRegion: KTable[String, Long] =
 userClicksStream

   // Join the stream against the table.
   .leftJoin(userRegionsTable)((clicks, region) => (if (region == null) "UNKNOWN" else region, clicks.clicks))

   // Change the stream from <user> -> <region, clicks> to <region> -> <clicks>
   .map((_, regionWithClicks) => regionWithClicks)

   // Compute the total per region by summing the individual click counts per region.
   .groupByKey
   .reduce(_ + _)

// Write the (continuously updating) results to the output topic.
clicksPerRegion.toStream.to(outputTopic)
```

A complete example of user-defined SerDes can be found in a test class within the library.

Please report any inaccuracies on this page or suggest an edit.

Last updated on Sep 10, 2019.