Kstream – Statefull DSL

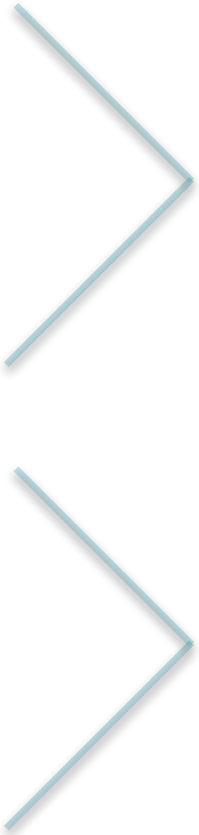
Stream Processing *Hard Parts*

- *Ordering* ✓
- *Partitioning & Scalability* ✓
- *Fault tolerance*
- *State Management*
- *Time, Window & Out-of-order Data*
- *Re-processing*

Stateful Transformations

States in Stream Processing

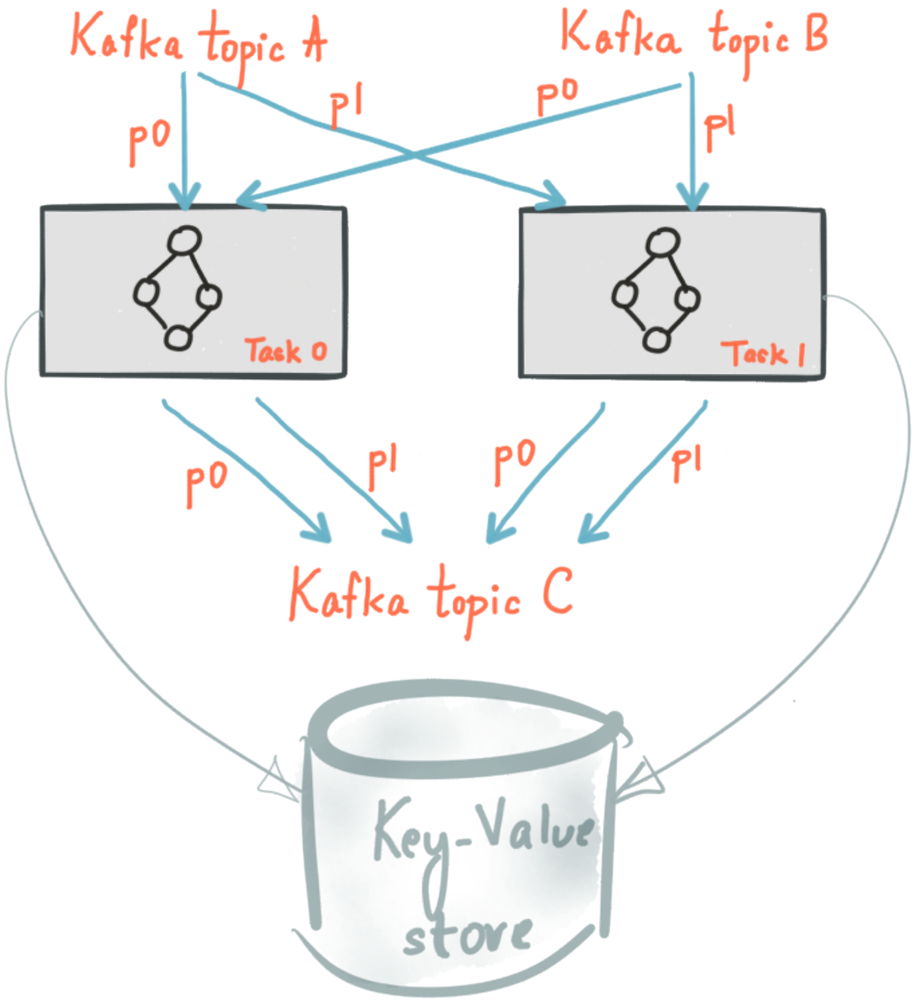
- ***filter***
- ***map***
- ***join***
- ***aggregate***



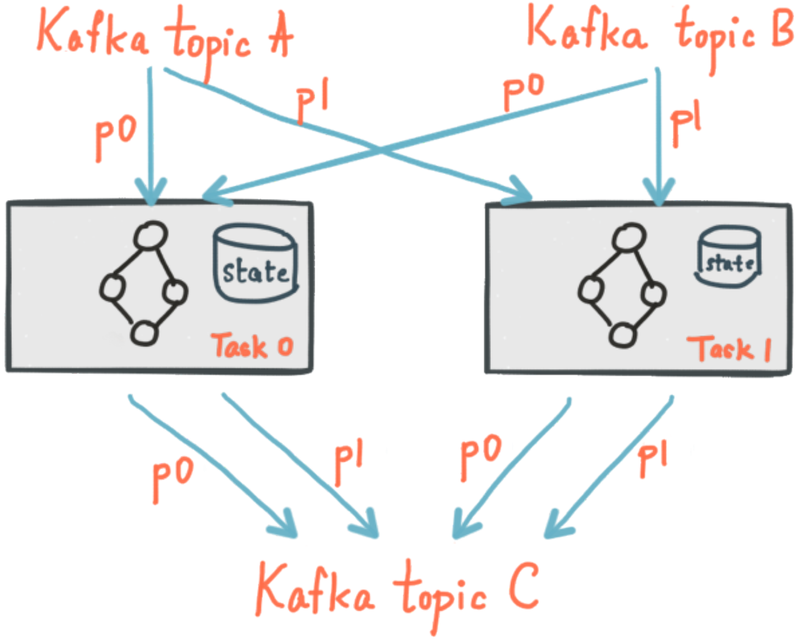
Stateless

Stateful

Remote State

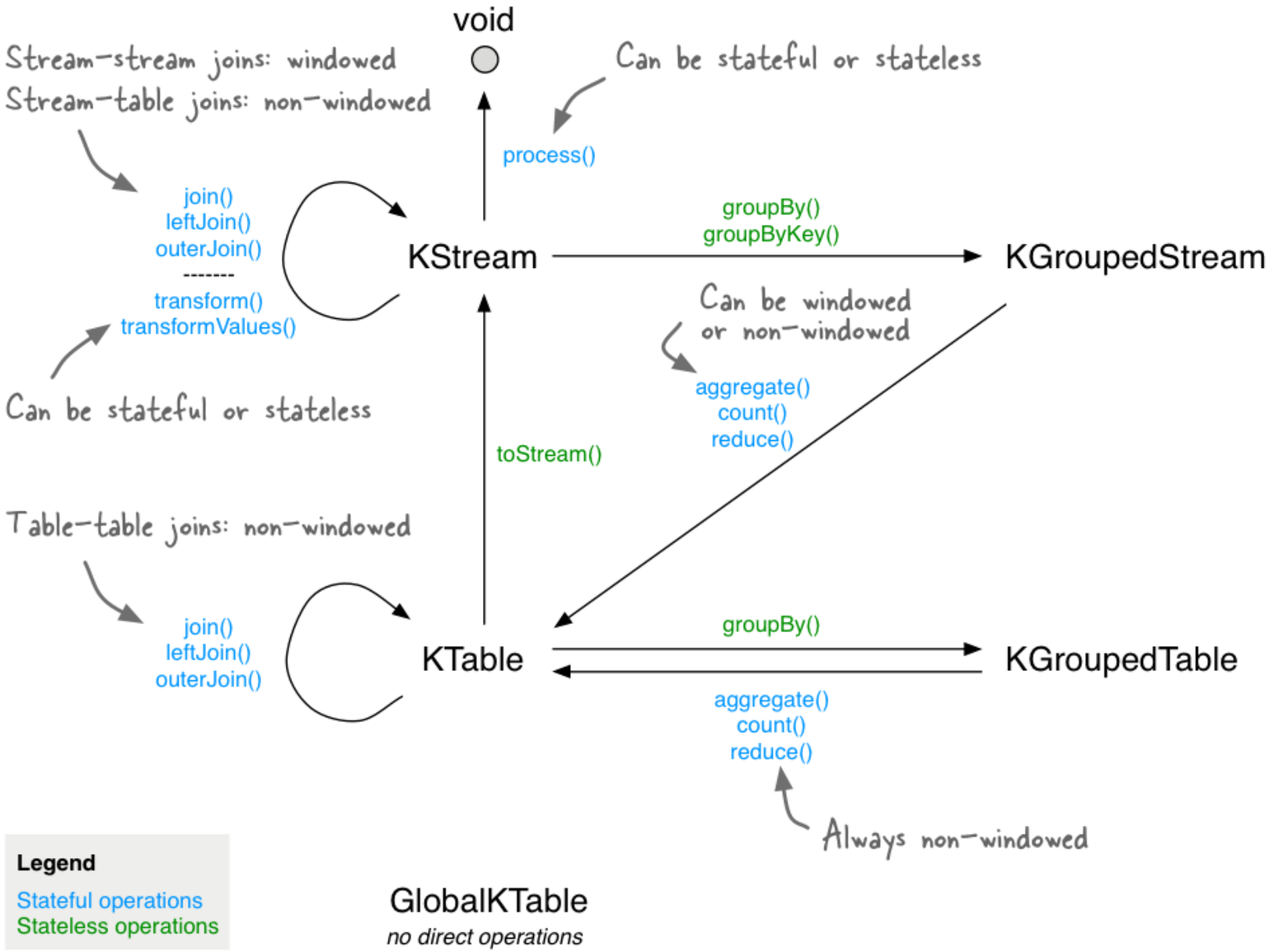


Local State



- faster
- better isolation
- flexible

Stateful transformations in the DSL



A stateful application: the WordCount algorithm

```
// 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();
```

Rolling aggregation

Aggregates the values of (non-windowed) records by the grouped key. Aggregating is a generalization of reduce and allows, for example, the aggregate value to have a different type than the input values.

```
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 */
);
// 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 */
);
```

Windowed aggregation

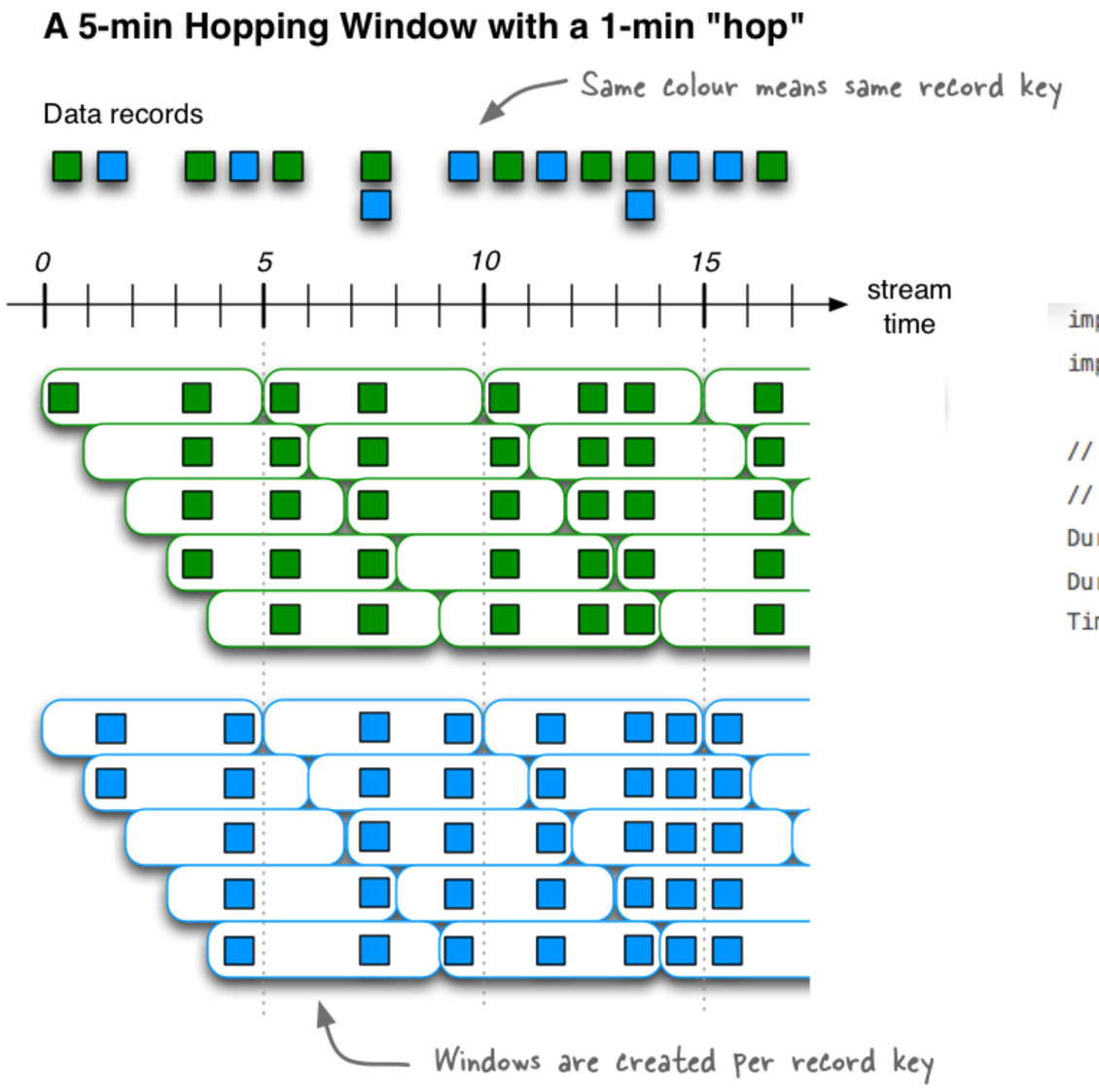
Aggregates the values of records, per window, by the grouped key. Aggregating is a generalization of reduce and allows, for example, the aggregate value to have a different type than the input values.

```
import java.time.Duration;

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

// 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)))
    .aggregate(
      () -> 0L, /* initializer */
      (aggKey, newValue, aggValue) -> aggValue + newValue, /* adder */
      Materialized.<String, Long, WindowStore<Bytes, byte[]>>as("time-windowed-aggregated-stream-store") /* state store name
        .withValueSerde(Serdes.Long())); /* serde for aggregate value */

// Aggregating with time-based windowing (here: with 5-minute tumbling windows)
// (note: the required "adder" aggregator is specified in the prior `cogroup()` call already)
KTable<Windowed<String>, Long> timeWindowedAggregatedStream = cogroupedStream.windowedBy(TimeWindows.of(Duration.ofMinutes(5)))
    .aggregate(
      () -> 0L, /* initializer */
      Materialized.<String, Long, WindowStore<Bytes, byte[]>>as("time-windowed-aggregated-stream-store") /* state store name
        .withValueSerde(Serdes.Long())); /* serde for aggregate value */
```

Windowing

Window name	Behavior	Short description
Hopping time window	Time-based	Fixed-size, overlapping windows
Tumbling time window	Time-based	Fixed-size, non-overlapping, gap-less 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

Hopping time window

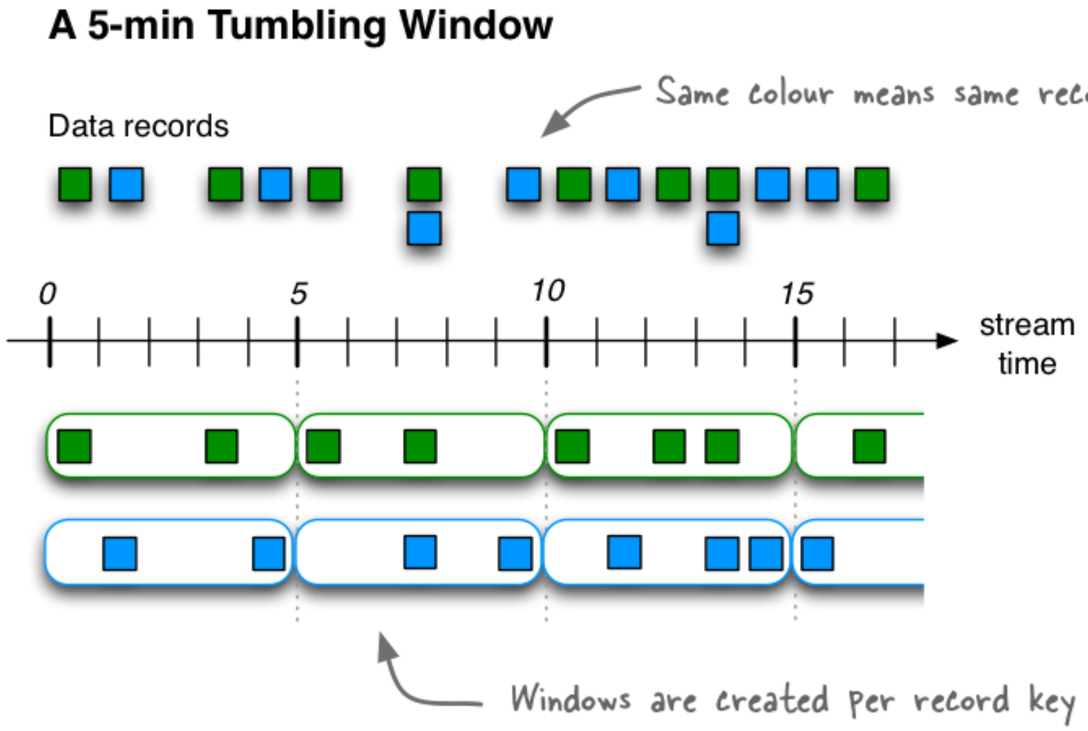



```
import java.time.Duration;
import org.apache.kafka.streams.kstream.TimeWindows;

// 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);
```

Stream - Window

```
stream_in
    .filter((key, value) -> value.startsWith("pulse"))
    .leftJoin(streamKtableDev, (pulse, device)
            -> device.substring(0, device.lastIndexOf('@')).replace('@', '_')
            + pulse.substring(pulse.indexOf('@')))
    .map((k, v) -> new KeyValue<>(v.substring(0, v.indexOf('@')),
            v.substring(v.indexOf('@') + 1)))
    .groupByKey()
    .windowedBy(TimeWindows.of(TimeUnit.MINUTES.toMillis(1)))
    .count()
    .toStream((k, v) -> String.format("%s - %s", k.key(),
            Date.from(Instant.ofEpochMilli(k.window().start()))))
    .mapValues(v -> "" + v)
    .to("stream_out");
```

Tumbling time windows




```
// A tumbling time window with a size of 5 minutes (and, by definition, an implicit
// advance interval of 5 minutes). Note the explicit grace period, as the current
// default value is 24 hours, which may be larger than needed for smaller windows.
Duration windowSizeMs = Duration.ofMinutes(5);
Duration gracePeriodMs = Duration.ofMinutes(1);
TimeWindows.of(windowSizeMs).grace(gracePeriodMs);

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

Sliding time windows

In Kafka Streams, sliding windows are used only for join operations, and can be specified through the JoinWindows class.

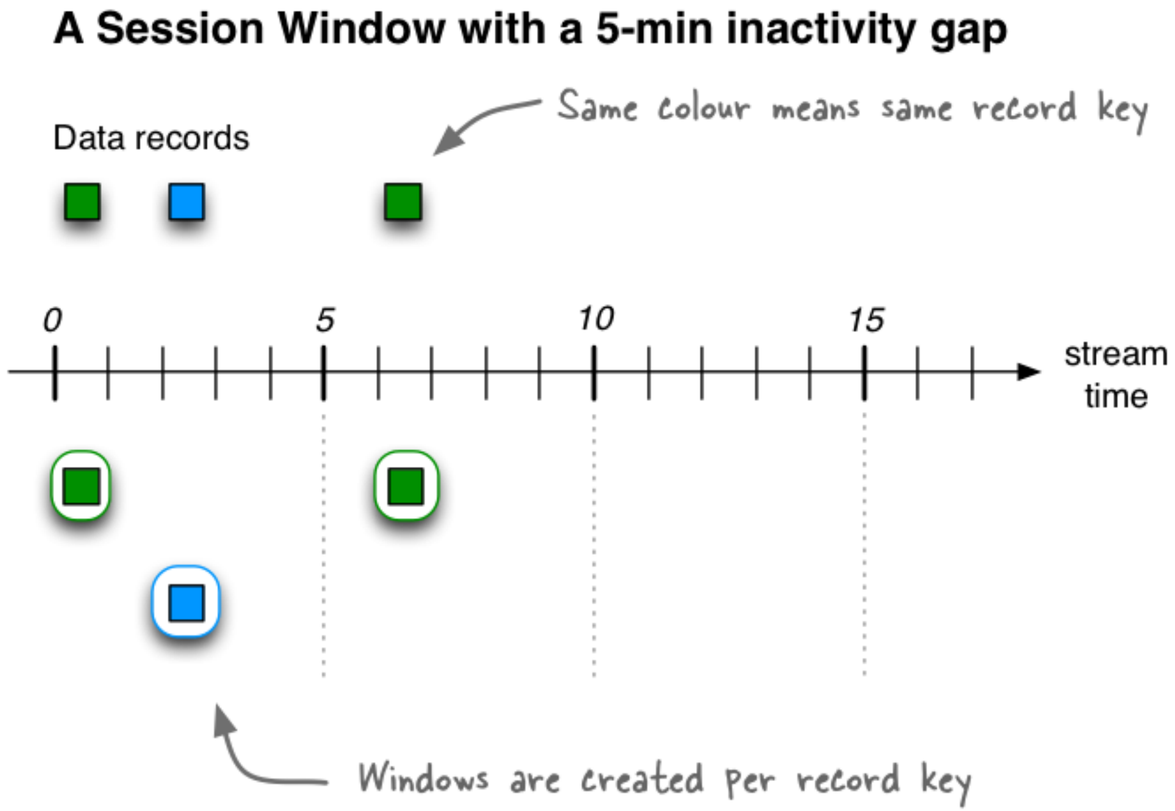
sliding windows are not aligned to the epoch, but to the data record timestamps.

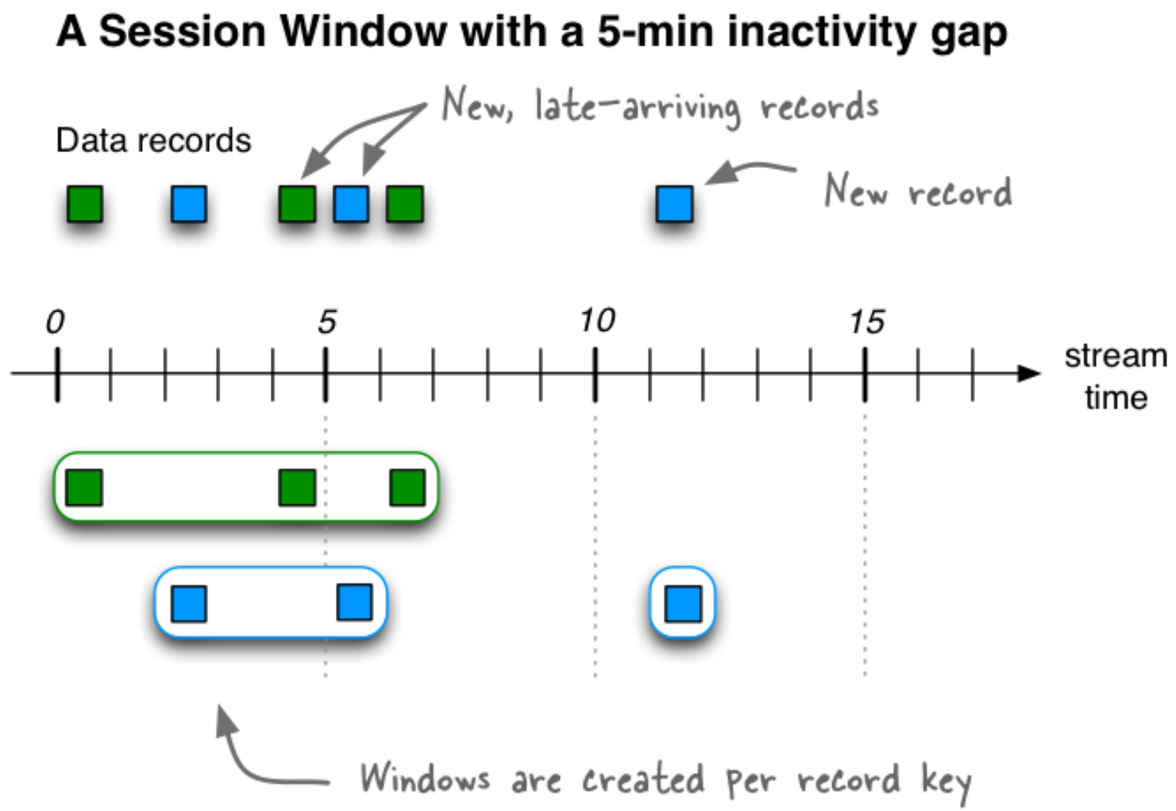
the lower and upper window time interval bounds of sliding windows are *both inclusive*.

```
stream
    .groupByKey()
    .windowedBy(SlidingWindows.withTimeDifferenceAndGrace(ofSeconds(20), ofSeconds(30))
    .toStream()
```

Session Windows

```
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));
```





Windowed aggregation

```
// Aggregating with session-based windowing (here: with an inactivity gap of 5 minutes)
KTable<Windowed<String>, Long> sessionizedAggregatedStream = groupedStream.windowedBy(SessionWindows.with(Duration.ofMinutes(5)).
    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-store") /* state store name
*/
        .withValueSerde(Serdes.Long())); /* serde for aggregate value */

// Aggregating with session-based windowing (here: with an inactivity gap of 5 minutes)
// (note: the required "adder" aggregator is specified in the prior `cogroup()` call already)
KTable<Windowed<String>, Long> sessionizedAggregatedStream = cogroupedStream.windowedBy(SessionWindows.with(Duration.ofMinutes(5)).
    aggregate(
        () -> 0L, /* initializer */
        (aggKey, leftAggValue, rightAggValue) -> leftAggValue + rightAggValue, /* session merger */
            Materialized.<String, Long, SessionStore<Bytes, byte[]>>as("sessionized-aggregated-stream-store") /* state store name
*/
        .withValueSerde(Serdes.Long())); /* serde for aggregate value */
```

Windowed aggregation.

```
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 gaps)
KTable<Windowed<String>, Long> aggregatedStream = groupedStream.windowedBy(
    SessionWindows.with(Duration.ofMinutes(5))) /* session window */
    .count();
```

Windowed aggregation

Combines the values of records, per window, by the grouped key. The current record value is combined with the last reduced value, and a new reduced value is returned.

```
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 */
  );
```

Rolling aggregation

Counts the number of records by the grouped key.

```
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();
```

Rolling aggregation.

Combines the values of (non-windowed) records by the grouped key. The current record value is combined with the last reduced value, and a new reduced value is returned.

```
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 */);
```

Example of semantics for stream aggregations

```
// 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 */
```

Example of semantics for stream aggregations

	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

```
// 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 */
```

Example of semantics for table aggregations

	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**)

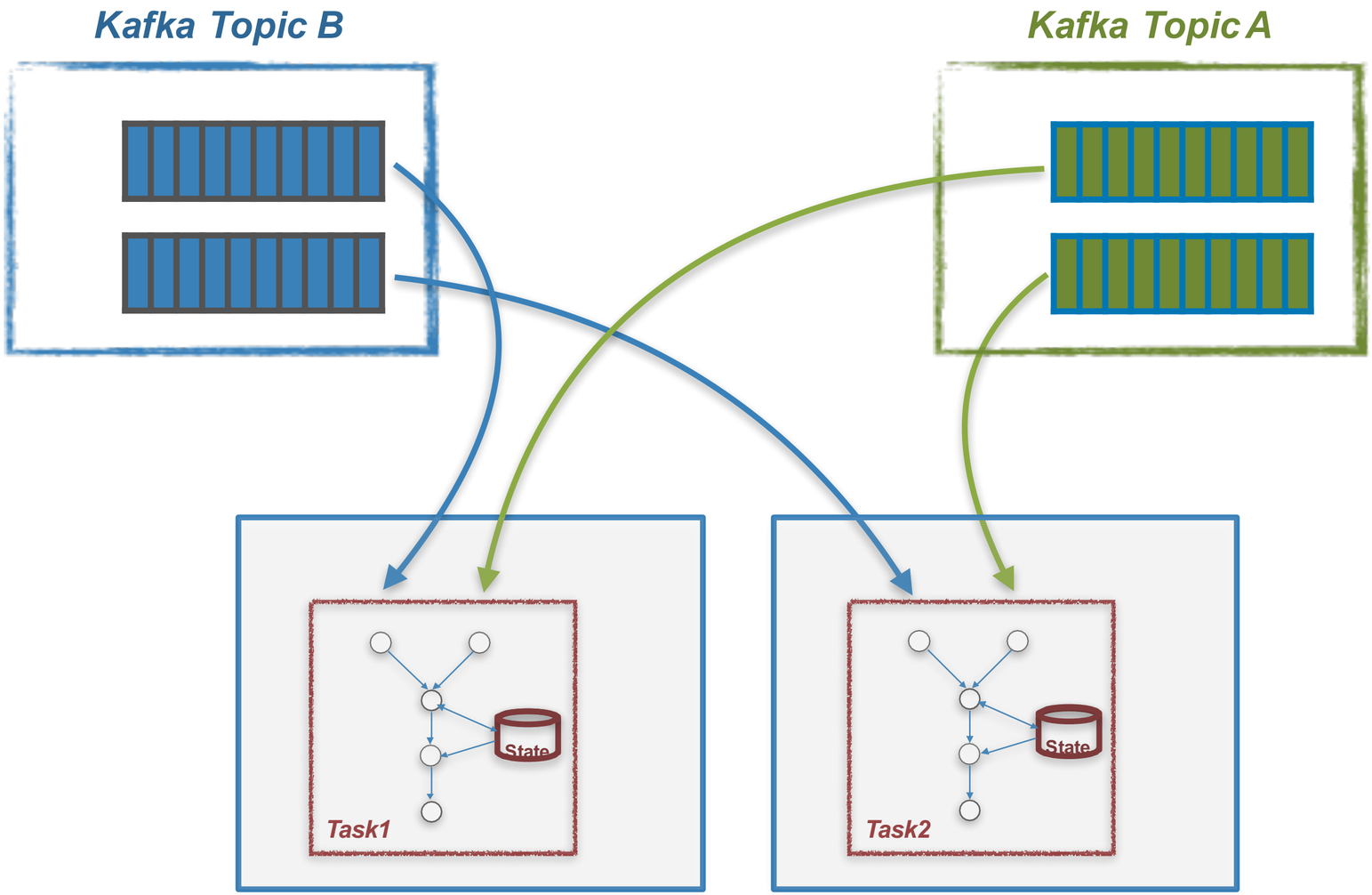
Join DSL

Join operands	Type	(INNER) JOIN	LEFT JOIN	OUTER JOIN
KStream-to-KStream	Windowed	Supported	Supported	Supported
KTable-to-KTable	Non-windowed	Supported	Supported	Supported
KTable-to-KTable Foreign-Key Join	Non-windowed	Supported	Supported	Not Supported
KStream-to-KTable	Non-windowed	Supported	Supported	Not Supported
KStream-to-GlobalKTable	Non-windowed	Supported	Supported	Not Supported
KTable-to-GlobalKTable	N/A	Not Supported	Not Supported	Not Supported

Join DSL

```
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 */
  );
```

States in Stream Processing



It's all about *Time*

- *Event-time* *(when an event is created)*
- *Processing-time* *(when an event is processed)*

Out-of-Order

Event-time	*1*	*2*	*3*	*4*	*5*	*6*	*7*
Processing-time	*1999*	*2002*	*2005*	*1997*	*1980*	*1983*	*2015*

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

```
public long extract(ConsumerRecord<Object, Object> record) {
    return record.timestamp();
}
```

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

processing-time

```
public long extract(ConsumerRecord<Object, Object> record) {
    return record.timestamp();
}
```

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

processing-time

```
public long extract(ConsumerRecord<Object, Object> record) {
    return record.timestamp();
}
```

event-time

Timestamp Extractor

```
public long extract(ConsumerRecord<Object, Object> record) {
    return System.currentTimeMillis();
}
```

processing-time

```
public long extract(ConsumerRecord<Object, Object> record) {
    return ((JsonNode) record.value()).get("timestamp").longValue();
}
```

event-time

Window Final Results

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

Applying processors and transformers

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

Email: Inbox (125)

```
// 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 further down below.
         .process(() -> new PopularPageEmailAlert("alerts@yourcompany.com"));
```

```
// 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 a
  }

  @Override
  void process(PageId pageId, Long count) {
    // Here you would format and send the alert email.
  }

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

Lab

Stream – DSL and Windows

DSL - join a stream and a table together