Windowing

Windowing lets you control how to group records that have the same key for stateful operations such as aggregations 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 `Windows#until()` and `SessionWindows#until()`.

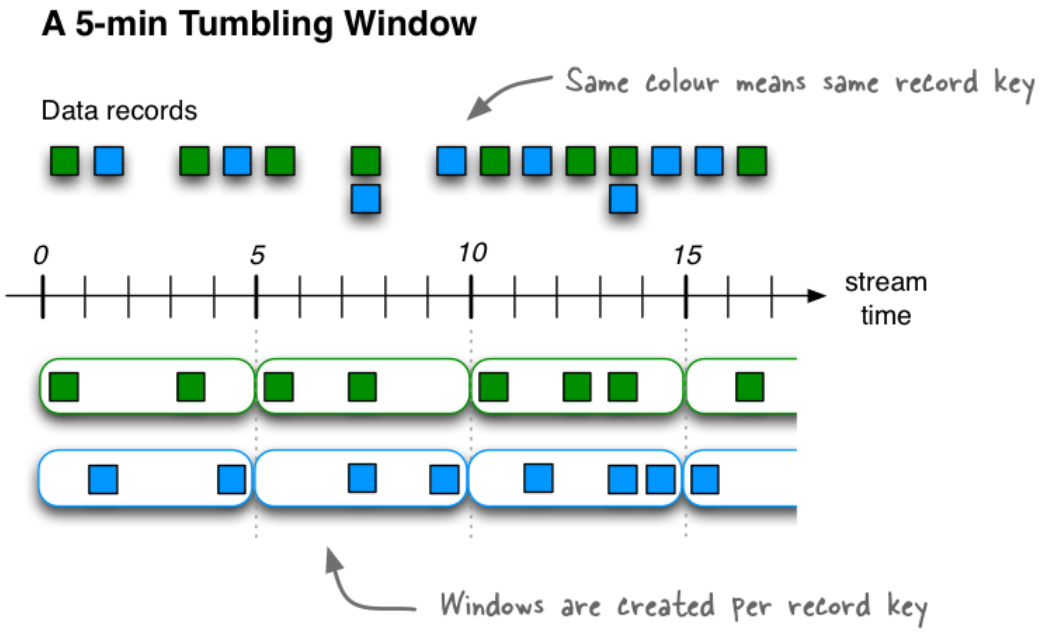
The DSL supports the following types of windows:

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

Window name	Behavior	Short description
Session window	Sess base	Dynamically-sized, non-overlapping, data-driven windows

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.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.TimeWindows;

// A tumbling time window with a size of 5 minutes (a
// advance interval of 5 minutes).
long windowSizeMs = TimeUnit.MINUTES.toMillis(5); // 5
TimeWindows.of(windowSizeMs);

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

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 windows.

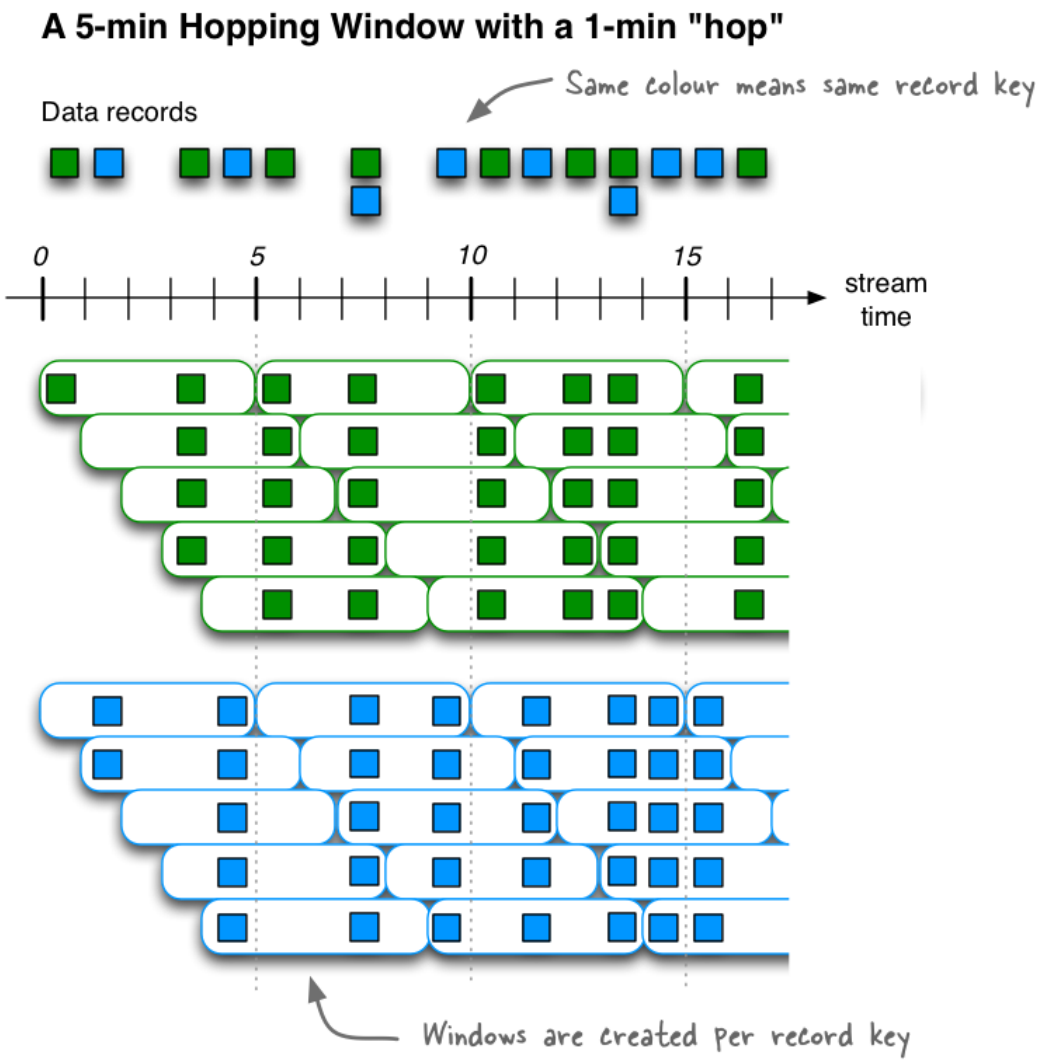
Note

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.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.TimeWindows;

// A hopping time window with a size of 5 minutes and
// The window's name -- the string parameter -- is us
long windowSizeMs = TimeUnit.MINUTES.toMillis(5); // 5
```

```
long advanceMs =    TimeUnit.MINUTES.toMillis(1); // 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),...`.

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:

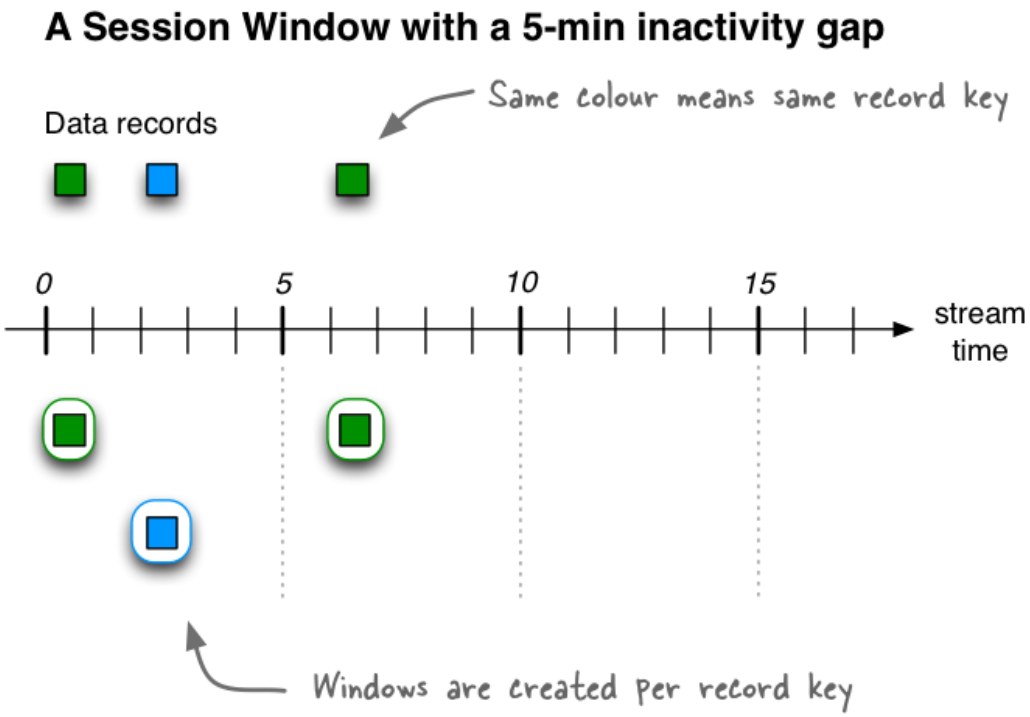
- all windows are tracked independently across keys – e.g. windows of different keys typically have different start and end times
- their window sizes sizes vary – even windows for the same key typically have different sizes

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:

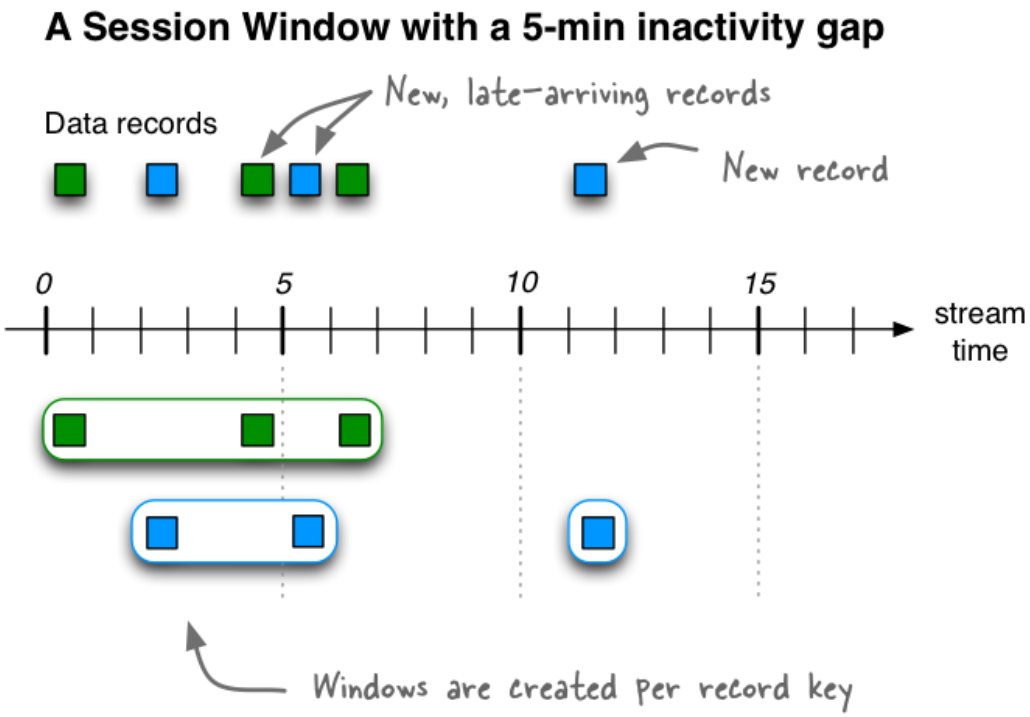
```
import java.util.concurrent.TimeUnit;
import org.apache.kafka.streams.kstream.SessionWindows

// A session window with an inactivity gap of 5 minut
SessionWindows.with(TimeUnit.MINUTES.toMillis(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.