KIP-450: Sliding Window Aggregations in the DSL

Status

Current state: *Accepted*

Discussion thread: *here*

Vote Thread: *here*

JIRA: *KAFKA-5636*

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Currently, Kafka Streams implements session windows, tumbling windows, and hopping windows as windowed aggregation methods. While hopping windows with a small advance time are similar to a sliding window, this implementation's performance is poor because it results in many overlapping and often redundant windows that require expensive calculations. As sliding windows do calculations for each *distinct* window, the sliding window implementation would be a more efficient way to do these types of aggregations. Additionally, sliding windows are inclusive on both the start and end time, creating a different set of windows than hopping windows, which are only inclusive on the start time.

This KIP proposes adding sliding windows to give users an efficient way to do sliding aggregation.

Public Interfaces

Add *kafka.streams.kstream.SlidingWindows*

```
public final class SlidingWindows {

         /**
     * Return a window definition with the window size based on the given maximum time difference between
records in the same window
          * and given window grace period
     * Records that come after set grace period will be ignored
     *
     * @param timeDifference the max time difference between two records in a window
     * @param grace the grace period to admit out-of-order events to a window
     * @return a new window definition
     * @throws IllegalArgumentException if the specified time difference or grace is zero or negative or can't
be represented as {@code long milliseconds}
     */
        public static SlidingWindows withTimeDifferenceAndGrace(final Duration timeDifference, final Duration
grace) throws IllegalArgumentException{}
```

Add additional *windowedBy* method to *KGroupedStream.java*

```
/**
     * Create a new {@link TimeWindowedKStream} instance that can be used to perform sliding windowed
aggregations.
     * @param windows the specification of the aggregation {@link SlidingWindows}
     * @return an instance of {@link TimeWindowedKStream}
     */
    TimeWindowedKStream<K, V> windowedBy(final SlidingWindows windows);
```

Add additional *windowedBy* method to *CogroupedKStream.java*

```
/**
* Create a new {@link TimeWindowedCogroupedKStream} instance that can be used to perform sliding windowed
* aggregations.
*
* @param windows the specification of the aggregation {@link SlidingWindows}
* @return an instance of {@link TimeWindowedCogroupedKStream}
*/
TimeWindowedCogroupedKStream<K, VOut> windowedBy(final SlidingWindows windows);
```

Proposed Changes

Creating a new finals class SlidingWindows, to add aggregation flexibility and features for users. Sliding windows will have an inclusive start and end point and each window will be unique, meaning that each distinct set of records will only appear in one window. This is in contrast with hopping windows, which can mimic sliding windows with an advance of 1ms, but result in many overlapping windows, as shown below. This causes expensive aggregation calculations for each window, whereas for the sliding windows, aggregations are only done for each unique window, cutting down significantly on the amount of work required.

Figure 1: Windows with SlidingWindow Implementation: time difference 10ms, 7 windows for 4 records.

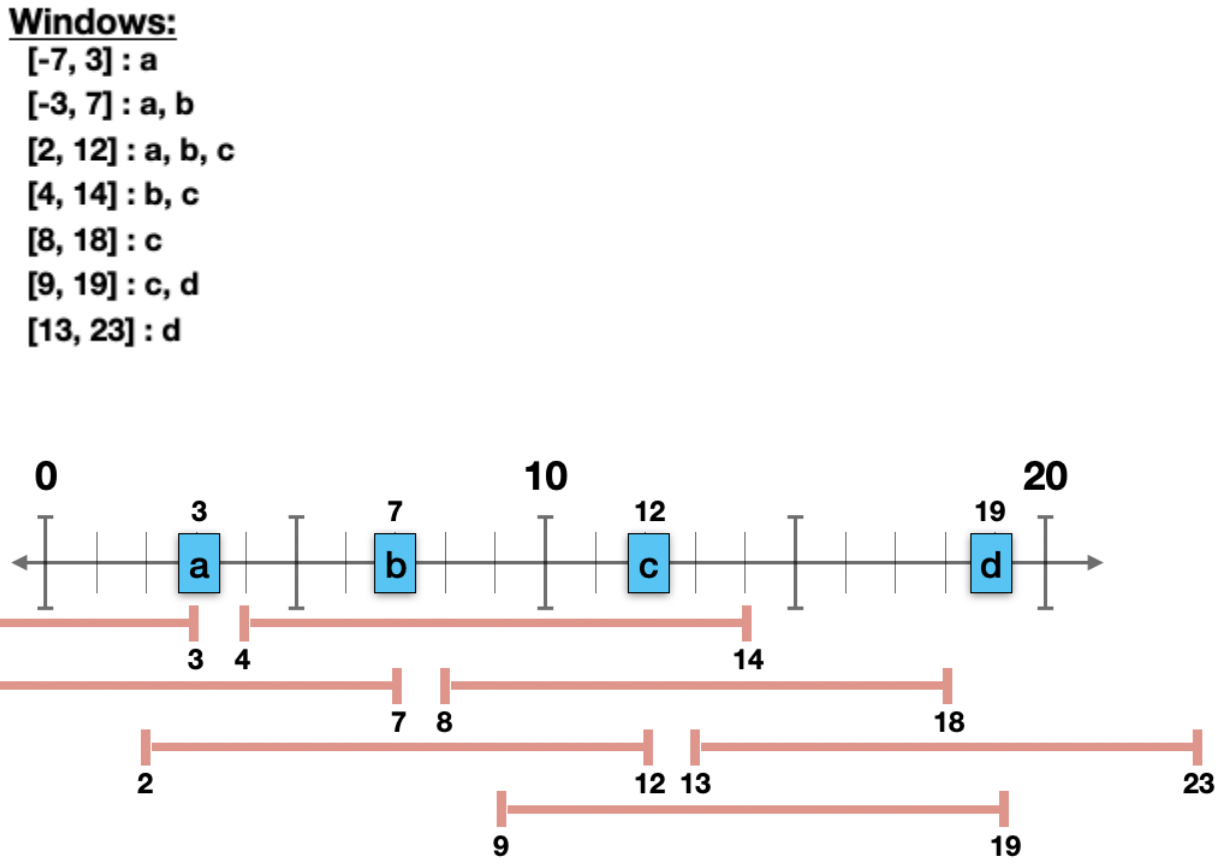
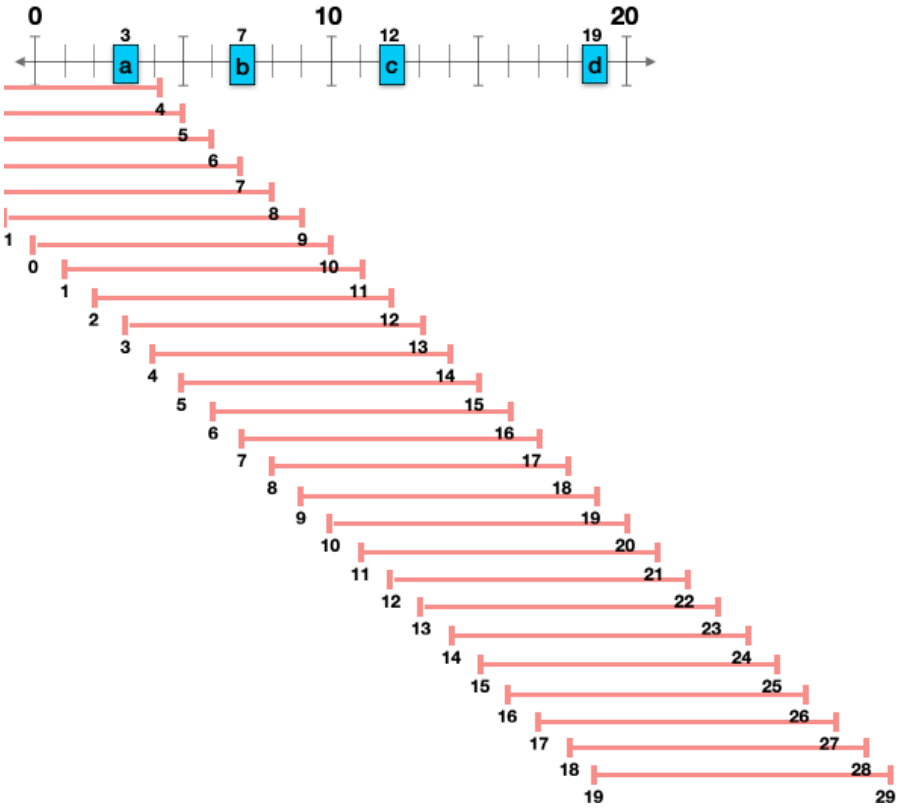


Figure 2: Windows with HoppingWindow with a 1ms advance: time difference 10ms, 26 windows for 4 records.



Usage

Typical Use

Sliding window aggregation to create windows of twenty seconds with a grace period of 30 seconds.

(**Notes**: grace will be required in SlidingWindows instead of implementing the 24 hour default seen in TimeWindows and SessionWindows. It will also be required to use a Timestamped Window Store, as calculating the sliding windows relies the extra information stored in a Timestamped Window Store. A different type of store will throw an exception):

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

Using Suppress to Limit Results

Send an alert if the window has fewer than 4 records, using suppression to limit all but the final results of the window. Original example here, suppression details here.

```
stream
        .groupByKey()
        .WindowedBy(SlidingWindows.withTimeDifferenceAndGrace(ofSeconds(20), ofSeconds(30))
        .count(Materialized.as("count-metric"))
        .suppress(Suppressed.untilWindowClose(BufferConfig.unbounded()))
        .filter(_ < 4)
        .toStream()
```

Querying IQ

When users want to query results from IQ, they need to know the start time of the window to get results. For most users it will be target time - timeDifference, since windows are defined backwards in SlidingWindows. For example, if there's an incident at 9:15am and they have a time difference of 10 minutes, they're looking for a window with the start time of 9:05. If they don't have the exact time, they can use a range query and traverse through the results to get the one they're looking for. After KIP-617, users will be able to traverse backwards through their range query, making accessing the latest window simpler.

```
// Define the processing topology (here: WordCount)
KGroupedStream<String, String> groupedByWord = textLines
  .flatMapValues(value -> Arrays.asList(value.toLowerCase().split("\\W+")))
  .groupBy((key, word) -> word, Grouped.with(stringSerde, stringSerde));

// Create a window state store named "CountsWindowStore" that contains the word counts for every minute, grace
period of 2 minutes
groupedByWord.windowedBy(SlidingWindows.TimeDifference(Duration.ofMinutes(1), Duration.ofMinutes(2))
  .count(Materialized.<String, Long, WindowStore<Bytes, byte[]>as("CountsWindowStore"));

// Get the window store named "CountsWindowStore"
ReadOnlyWindowStore<String, Long> windowStore =
    streams.store("CountsWindowStore", QueryableStoreTypes.windowStore());

//Find the most recent window using currently available queries

// Fetch values for the key "world" for all of the windows available in this application instance.
Instant timeFrom = Instant.ofEpochMilli(0); // beginning of time = oldest available
Instant timeTo = Instant.now(); // now (in processing-time)
WindowStoreIterator<Long> iterator = windowStore.fetch("world", timeFrom, timeTo);
while (iterator.hasNext()) {
  KeyValue<Long, Long> next = iterator.next();
  long windowTimestamp = next.key;
}
 System.out.println("Count of 'world' in the last window @ " + windowTimestamp + " is " + next.value);

// close the iterator to release resources
iterator.close()

//Find the most recent window using KIP-617's reverse iterator

Instant timeFrom = Instant.ofEpochMilli(0); // beginning of time = oldest available
Instant timeTo = Instant.now(); // now (in processing-time)
WindowStoreIterator<Long> iterator = windowStore.backwardFetch("world", timeFrom, timeTo);
KeyValue<Long, Long> next = iterator.next();
long windowTimestamp = next.key;
System.out.println("Count of 'world' in the last window @ " + windowTimestamp + " is " + next.value);
iterator.close()
```

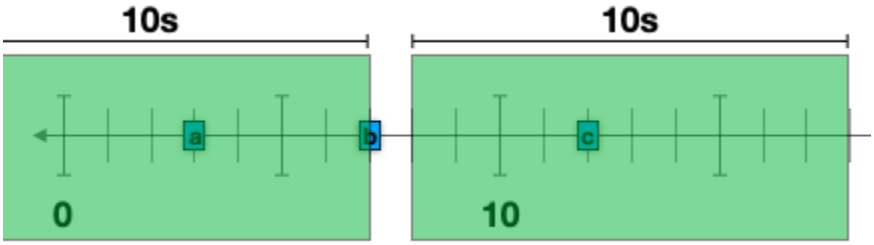
Processing Windows

To process a new record, there are three major steps.

1. Create the two new windows defined by this record (assuming these windows do not yet exist)
    a. Aggregate existing records that fall within these windows with the current record's value (or note that these windows are empty and ignore them)
2. Find existing windows that the new record falls within
    a. Add the new record's value to these windows' aggregations

With Sliding Windows, a new record creates two new windows: one that ends at the record's timestamp, and one that starts 1ms after the record's timestamp. The first window will contain the new record, while the second will not. This is shown with record **b** in figure 3. These windows were chosen to ensure that for all the records, every combination of possible records given the time difference will be created. This allows for easy aggregation when creating new windows, as any aggregation needed in a new window will have already been calculated in an existing window, explanation here. For a new window that overlaps with existing records (record **a** in figure 3), these records will be aggregated and added to the new window's value.

Figure 3: New windows of time difference 10ms defined for **b**



Any already created windows that the new record falls within will be updated to have an aggregation that includes the new records value. This requires doing a scan through the WindowStore to find these windows, similar to the SessionWindow scan required for aggregating SessionWindows, and updating the values.

Grace Period

Instead of creating the 24 hour default grace period seen with TimeWindows and SessionWindows, SlidingWindows will require the grace period to be set by the user. The default grace period of 24 hours is often too long and was chosen when retention time and grace period were the same. Grace period will be an additional parameter in the #of method to make sure users know that it is required: *.windowedBy(SlidingWindows.withTimeDifferenceAndGrace (twentySeconds, fiftySeconds).* If grace period isn't properly initialized, an error will be thrown through the *withTimeDifferenceAndGrace* method.

Out-of-order Records

Records that come out of order will be processed the same way as in-order records, as long as they fall within the grace period. Two new windows will be created by the out-of-order record, one ending at that record's timestamp, and one starting at 1ms past that record's timestamp. These windows will be updated with existing records that fall into them by using aggregations previously calculated. If either of these windows are empty, or empty aside from the current record's value, this will be taken into account. Additionally, any existing windows that the new record falls within will be updated with a new aggregation that includes this out-of-order record.

Emitting Results

As with hopping windows, sliding windows will emit partial results as windows are updated, so any window that gets a new aggregate or an updated aggregate will emit. This feature can be suppressed through suppression, which allows users to ignore all results except for the final one, only emitting after the grace period has passed. In the future, suppress can be updated to emit results for windows that are still within the grace period but whose start and end points are both earlier than stream time.

Compatibility, Deprecation, and Migration Plan

Implementing a new feature, should be compatible with all existing features.

Rejected Alternatives

Other Types of Sliding Windows:

- Creating a new window with the record's timestamp at the beginning, and a second window with 1ms before the record's timestamp at the end. This version of defining 2 new windows is less intuitive and is less compatible with looking at a time segment back in time.
- Having either/both ends not inclusive. Traditionally, time windows are inclusive, and as hopping windows and tumbling windows are exclusive on the endpoint, this is a distinguishing feature. Changing exclusivity is just a matter of adding 1ms.
- Having one window anchored and ending at the current time. By setting the retention time equal to the time difference + grace period, this can be achieved, and allowing only this type of sliding window limits the historical views and therefore flexibility for use cases.

Additional Features:

- Allow users to specify granularity of window definition (something other than 1ms). This is an option for the future, but allowing a user to specify 1 second or 1 minute granularity results in windows that are very similar to tumbling windows, straying from the purpose of sliding windows.
- Include a 'subtraction' feature as a complement to aggregation. While this might be a nice to have feature in the future, it's out of scope for this KIP and isn't necessary to make sliding windows work well.

Other Implementation Options:

- Making SlidingWindows an option within TimeWindow. This would add more requirements for users and blur the lines between hopping and sliding windows, while potentially making optimization harder.
- Making SlidingWindows extend Windows<Window> or Windows<TimeWindow>. To keep the code clean and allow for future-proofing this was not chosen.
- Setting the default grace period to 0ms. This was deemed to be too confusing for users who are used to a 24 hour default and would now have to remember which type of windows have a certain grace period default.