KIP-450: Sliding Window Aggregations in the DSL

Status

Current state: *Under Discussion*

Discussion thread: *here*

JIRA: *here*

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

Motivation

The DSL currently supports windowed aggregations for only two types of time-based window: hopping and tumbling. A third kind of window is defined, but only used in join operations: sliding windows. Users needing sliding window semantics can approximate them with hopping windows where the advance time is 1, but this workaround only artificially resembles the sliding window model; aggregates will be output for every defined hopping window, of which there will likely be a large number (specifically, the size of the window in milliseconds). The semantics for out-of-order data also differ, as sliding windows are inclusive at both ends (ie start and end time bounds)

Public Interfaces

Rather than adapt the existing TimeWindows interface (which provides semantics for tumbling and hopping windows), I propose to add a separate SlidingWindows class. This will resemble a stripped-down version of the TimeWindows class, and have the following public API:

```
public final class SlidingWindows extends Windows<TimeWindow> {

        public static SlidingWindows of(final Duration size);

        public SlidingWindows grace(final Duration afterWindowEnd);

        @Override
        public Map<Long, TimeWindow> windowsFor(final long timestamp);

        @Override
    public long gracePeriodMs();

        @Override
    public long size();

        @Override
    public boolean equals(final Object o);

        @Override
    public int hashCode();

        @Override
        public String toString();
}
```

This would effectively be used in the same manner as TimeWindows. For example, to do a counting aggregation with sliding windows, you would have something like

```
final KTable<Windowed<String>, Long> counts = source
            .groupBy((key, value) -> value)
            .windowedBy(SlidingWindows.of(Duration.ofSeconds(5)))
            .count();
```

Proposed Changes

The semantics of the sliding window based aggregations are as follows:

- Only one window is defined for each distinct set of records falling within a single window.
- Since the windows represent records that fall within "windowSize" of each other, both time bounds are inclusive
- The window is defined "forwards" in time, such that window start time < window end time
- A record is output for a window when it can no longer be updated, ie when its endTime < streamTime - gracePeriod
- Each distinct timestamp seen in input effectively triggers one output. In other words, each new record results in exactly one output UNLESS we have already seen a record with the same timestamp (in which case there would be no new distinct set of records, and thus no new window)

- Out of order data that still falls within the grace period will be kept (but will never trigger any output as it does not advance streamtime)
- Out of order data that arrives outside the grace period is dropped
- Grace period will default to 0

* Note that in processing a new record, an arbitrary number of outputs may be triggered by stream time advancing and any number of older windows leaving the grace period. But semantically, 1 (distinct) record 1 output

Implementation

We can leverage on the existing window stores with a slightly different processing path. As with other types of windowed aggregations, the underlying store will be used to hold the aggregate of each defined window. Rather than putting a new window into the store every X ms (as is done for TimeWindows), a new window will be inserted into the store only upon seeing a record (with new timestamp). This enforces "one window per distinct set of records."

Upon advancing streamtime, a range query can be used to collect all the windows that will be permanently closed (endTime < new streamTime - gracePeriod) by this new streamTime and their final aggregation result will at this time be forwarded. The new window will then be inserted into the store, and a range query used to find and update any existing open windows that include this new record.

If a record arrives that does not advance streamTime, nothing will be forwarded; all we must do in this case is update the existing windows that contain it.

Compatibility, Deprecation, and Migration Plan

N/A

Rejected Alternatives

Operations & Semantics

In considering the semantics we have some flexibility in how/when to output the results of an aggregations. For example, rather than outputting only the final result after the window has left the grace period we might have wanted to send a result as soon as it closed, and then send further updates as any out of order data arrived. However realistically out of order data occurs often enough that it makes sense to not output a result right away, and rather wait for potential updates for some amount of time. Naturally the grace period would be a sensible choice for this time to wait so as not to flood the downstream with more updates than results. Outputting only the final result, rather than some potential result and a chance of some updates, is likely to be the more straightforward to deal with even if you may not see output immediately but until the grace period has passed.

API

Rather than implementing a new Windows class we could have added a boolean method/flag to TimeWindows, signaling whether the window is sliding or not (tumbling/hopping). However there might be some confusion there if, say, a user set "slidingWindow = true" but then also specified an advance time for their TimeWindows. We could of course just throw an exception in such cases but it would unnecessarily clutter the TimeWindows class when we could instead have a minimal class that clearly describes what it does (sliding windows).

Really, here we should just choose whichever option is most discoverable for users.

Implementation