KIP-94 Session Windows

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: *here*

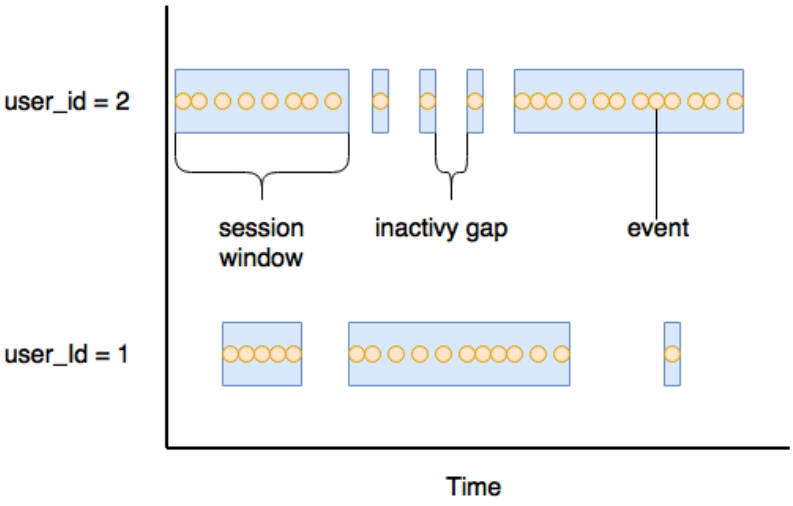
Released:

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

Motivation

Many stream processing applications need to maintain state across a period of time. Online services, for example, like to understand the behaviour of a typical user. In order to understand this behaviour, the interactions a user has with their website, or streaming service, are grouped together into sessions.

Sessions usually represent some period of user activity separated by a defined gap of inactivity. As events are processed they may create new sessions or be merged into existing sessions. Any sessions that are merged will usually extend the time window the session covers. Thus, session windows are dynamic in nature, i.e, they don't fit into fixed and aligned time windows, but rather are varying-sized time windows that can't be pre-computed and are driven by timestamps of the data.



Proposed Changes

We will provide a way for developers using the DSL to specify that they want an aggregation to be aggregated into *SessionWindows*. Three overloaded methods will be added to *KGroupedStream*:

```
KTable<Windowed<K>, V> reduce(final Reducer<V> reducer,
                              final SessionWindows sessionWindows,
                              final String storeName);


<T> KTable<Windowed<K>, T> aggregate(final Initializer<T> initializer,
                                     final Aggregator<? super K, ? super V, T> aggregator,
                                     final Merger<? super K, T> sessionMerger,
                                     final SessionWindows sessionWindows,
                                     final Serde<T> aggValueSerde,
                                     final String storeName);

KTable<Windowed<K>, Long> count(final SessionWindows sessionWindows, final String storeName);
```

A typical aggregation might look like:

```
stream.groupByKey().aggregate(initializer,
                              aggregator,
                              merger,
                              SessionWindows.with(FIVE_MINUTES)
                  .until(ONE_HOUR),
   aggregateValueSerde,
  “session-store”);
```

The above statement will aggregate the results into *SessionWindows* with an inactivity gap of five minutes. The Sessions will be retained for one hour from when they have closed, during which they will accept input from late-arriving records that would be considered as falling into a particular session.

In order to process *SessionWindows* we'll need to add a new Processor. This will be responsible for creating sessions, merging existing sessions into sessions with larger windows, and producing aggregates from a session's values.

On each incoming record the process method will:

1. Find any adjacent sessions that either start or end within the inactivity gap, i,e., where the end time of the session is > now - inactivity gap, or the start time is < now + inactivity gap.
2. Merge any existing sessions into a new larger session using the *SessionMerger* to merge the aggregates of the existing sessions.
3. Aggregate the value record being processed with the merged session.
4. Store the new merged session in the *SessionStore.*
5. Remove any merged sessions from the *SessionStore.*

We will leverage the work done in KIP-63 for forwarding the aggregated result of a session window. That is, the aggregates will be de-duplicated in the cache and only be forwarded downstream on *commit*, *flush*, or *evict*. This is inline with how all existing aggregations work in Kafka Streams.

Late arriving data

Late arriving data is mostly treated the same as non-late arriving data, i.e., it can create a new session or be merged into an existing one. The only difference is that if the data has arrived after the retention period, defined by *SessionWindows.until(..)*, a new session will be created and aggregated, but it will not be persisted to the store.

SessionWindows

We propose to add a new class *SessionWindows*. *SessionWindows* will be able to be used with new overloaded operations on *KGroupedStream*, i.e, *aggregate(…), count(..), reduce(…)*. A *SessionWindows* will have a defined gap, that represents the period of inactivity. It will also provide a method, *until(...)*, to specify how long the data is retained for, i.e., to allow for late arriving data.

SessionStore

We propose to add a new type of *StateStore*, *SessionStore*. A *SessionStore*, is a segmented store, similar to a *WindowStore*, but the segments are indexed by session end time. We index by end time so that we can expire (remove) the *Segments* containing sessions where session endTime < stream-time - retention-period.

The records in the *SessionStore* will be stored by a *Windowed* key. The *Windowed* key is a composite of the record key and a *TimeWindow*. The start and end times of the *TimeWindow* are driven by the data. If the Session only has a single value then start == end. The segment a Session is stored in is determined by *TimeWindow.end*. Fetch requests against the *SessionStore* use both the *TimeWindow.start* and *TimeWindow.end* to find sessions to merge.

Each *Segment* is for a particular interval of time. To work out which *Segment* a session belongs in we simply divide *TimeWindow.end* by the segment interval. The segment interval is calculated as *Math.max(retentionPeriod / (numSegments - 1), MIN_SEGMENT_INTERVAL).*

For example, given a segment interval of 1000:

TimeWindow End	Segment Index
0	0
500	0
1000	1
2000	2

Put

As session aggregates arrive, i.e., on put, the implementation of *SessionStore* will:

1. use *TimeWindow.end* to get an existing segment or create a new *Segment* to store the aggregate in. A new *Segment* will only be created if the *TimeWindow.end* is within the retention period.
2. If the *Segment* is non-null, we add the aggregate to the *Segment*.
3. If the *Segment* was null, this signals that the record is late and has arrived after the retention period. This record is not added to the store.

FindSessionsToMerge

When *SessionStore.findSessionsToMerge(...)* is called we find all the aggregates for the record key where *TimeWindow.end* >= *earliestEndTime* && *TimeWindow.start* <= *latestStartTime*. In order to do this:

1. Find the *Segments* to search by getting all *Segments* starting from *earliestEndTime*
2. Define the range query as:
    a. from = (record-key, end=earliestEndTime, start=0)
    b. to = (record-key, end=Long.MAX_VALUE, start=latestStartTime)

For example, if for an arbitrary record key we had the following sessions in the store:

Session Start	Session End
0	99
101	200
201	300
301	400

When we query the store with earliestEndTime = 150, latestStartTime = 300 we would retrieve sessions starting at 101 and 201.

Public Interfaces

SessionWindows

```
public class SessionWindows {

    /**
     * Create a new SessionWindows with the specified inactivityGap
     *
     * @param inactivityGap
     * @return
     */
    public static SessionWindows with(final long inactivityGap)


    /**
     * Set the window maintain duration in milliseconds of streams time.
     * This retention time is a guaranteed <i>lower bound</i> for how long
     * a window will be maintained.
     *
     * @return  itself
     */
    public SessionWindows until(long durationMs)

    /**
     * @return the inactivityGap
     */
        public long inactivityGap()

   /**
    * @return the minimum amount of time a window will be maintained for.
    */
    public long maintainMs();

}
```

Merger

```
/**
 * The interface for merging aggregate values with the given key
 *
 * @param <K>   key type
 * @param <T>   aggregate value type
 */
public interface Merger<K, T> {
   /**
    * Compute a new aggregate from the key and two aggregates
    *
    * @param aggKey    the key of the record
    * @param aggOne    the first aggregate
    * @param aggTwo    the second aggregate
    * @return          the new aggregate value
    */
   T apply(K aggKey, T aggOne, T aggTwo);
}
```

SessionStore

```
/**
* Interface for storing the aggregated values of sessions
* @param <K>   type of the record keys
* @param <AGG> type of the aggregated values
*/
public interface SessionStore<K, AGG> extends StateStore, ReadOnlySessionStore<K, AGG> {

    /**
     * Find any aggregated session values with the matching key and where the
     * session’s end time is >= earliestSessionEndTime, i.e, the oldest session to
     * merge with, and the session’s start time is <= latestSessionStartTime, i.e,
     * the newest session to merge with.
     */
   KeyValueIterator<Windowed<K>, AGG> findSessionsToMerge(final K key, final long earliestSessionEndTime, final
long latestSessionStartTime);

   /**
    * Remove the aggregated value for the session with the matching session key
    */
   void remove(final Windowed<K> sessionKey);

   /**
    * Write the aggregated result for the given session key
    */
   void put(final Windowed<K> sessionKey, AGG aggregate);
}
```

ReadOnlySessionStore

This is primarily provided for InteractiveQueries

```
/**
 * A session store that only supports read operations.
 * Implementations should be thread-safe as concurrent reads and writes
 * are expected.
 *
 * @param <K> the key type
 * @param <V> the value type
 */
@InterfaceStability.Unstable
public interface ReadOnlySessionStore<K, AGG> {

    /**
     * Retrieve all aggregated sessions for the provided key
     * @param    key record key to find aggregated session values for
     * @return   KeyValueIterator containing all session aggregates for the provided key.
     */
    KeyValueIterator<Windowed<K>, AGG> fetch(final K key);
}
```

QueryableStoreTypes

Additional Method

```
/**
 * A {@link QueryableStoreType} that accepts {@link ReadOnlySessionStore}
 * @param <K>   key type of the store
 * @param <V>   value type of the store
 * @return  {@link SessionStoreType}
 */
public static <K, V> QueryableStoreType<ReadOnlySessionStore<K, V>> sessionStore()
```

KGroupedStream

Additional methods

```
KTable<Windowed<K>, V> reduce(final Reducer<V> reducer,
                              final SessionWindows sessionWindows,
                              final String storeName);

KTable<Windowed<K>, V> reduce(final Reducer<V> reducer,
                              final SessionWindows sessionWindows,
                              final StateStoreSupplier<SessionStore> storeSupplier);


<T> KTable<Windowed<K>, T> aggregate(final Initializer<T> initializer,
                                     final Aggregator<? super K, ? super V, T> aggregator,
                                     final Merger<? super K, T> sessionMerger,
                                     final SessionWindows sessionWindows,
                                     final Serde<T> aggValueSerde,
                                     final String storeName);

<T> KTable<Windowed<K>, T> aggregate(final Initializer<T> initializer,
                                     final Aggregator<? super K, ? super V, T> aggregator,
                                     final Merger<? super K, T> sessionMerger,
                                     final SessionWindows sessionWindows,
                                     final Serde<T> aggValueSerde,
                                     final StateStoreSupplier<SessionStore> storeSupplier);

KTable<Windowed<K>, Long> count(final SessionWindows sessionWindows, final String storeName);

KTable<Windowed<K>, Long> count(final SessionWindows sessionWindows, final StateStoreSupplier<SessionStore>
storeSupplier);
```

Compatibility, Deprecation, and Migration Plan

- None required as we are introducing new APIs only

Test Plan

The majority of this feature can be tested with unit tests, however we will write integration and/or system tests to cover the end-to-end scenarios.

Rejected Alternatives

- Using the KeyValueStore to store the sessions. It would result in excessive IOs as we'd need to store a List per sessionId. So every lookup for a given sessionId would need to fetch and deserialize the list. Also, punctuate would need to retrieve every session every time it runs in order to determine those that need to be aggregated and forwarded.
- Keeping the list of values in the store. Whilst this would result in us being able to use the existing Windows based API methods on KGroupedStrea , it could result in excessive IO operations and possibly excessive storage use.
- Using punctuate to perform the aggregations and forward downstream. This would result in potentially fewer records being forwarded downstream as they would only be forwarded when a session is 'closed'. However, it doesn't align with the current approach in Kafka Streams, i.e., to use the cache for de-duplication and forward on commit, flush, or eviction (KIP-63). Further, punctuate currently depends on stream time and is will not be called in the absence of records.