Samza SQL Window Operator design (SAMZA-552)

Introduction

To support stream SQL in Samza, we need to be able to materialize a view of the infinite stream in order to perform any relational algebra calculation. CQL paper lays the theoretical fundation for the window operator and there are various different implementation of windowing operations in StreamSQL, Esper, sqlstream, and Azure. There is a discussion in the literature regarding to the difference between time vs tuple based window. However, there is practically no window operator that is designed to work in a distributed stream process context with the following requirements:

- Support timely and deterministic window output with un-predictable message delivery latencies
- Support deterministic window output with node failure and message re-delivery

Use cases

The window operator generates a snapshot of the infinite stream marked by the start and end positions. Users will use the window output in:

- Window-based aggregation: e.g. count a certain type of events in the window
- Join: join the snapshot with another table

As most users only cares about the time when the message is generated, not the time it is received, we will use the term **event time** to refer to the timestamp when the message is generated in an application. This also implies that the producers will have to add timestamp to each and every message in time-based window.

Most window operator use cases can be categorized by the following:

- **Window size measurement**: Whether the window size is measured in event time or number of messages (i.e. tuples).
- **Window advancement**: This is referring to the pattern in which the window start point changes in a stream. Three types are defined: **tumbling window**, **hopping window**, and **sliding window**, as illustrated here.
- **Window length**: Whether the window length (measured in size) is fixed or dynamic.
- **Window creation**: Whether the window creation is static or on-demand.

Problems

There are a few issues to be addressed in a distributed environment:

1. Messages may be delayed or lost
2. Messages may be re-played
3. Window state may need to failover/recover when node crashes

4. Window output need to be reliably re-produced when recovered w/ messages re-delivered w/ arbitrary latencies

A Generic Model for Windowing

The talk from MillWheel and discussion in SAMZA-552 inspired the ideas:

1. The window operator should be able to update each window's output multiple times, controlled by window update policies
2. The window operator should keep the past window states s.t. re-computing the results for late arrivals are possible

Use Case Study

Now let's look at the use cases based on the generic model that supports the above requirements:

Case 1: windowed aggregation

When the window operator's output is used to perform an aggregation, such as "count" or "max", it is pretty simple. Consider the following sequence of messages in the stream Orders:

Orders:

1. m1: (offset=1, value=0, time=8:59:10) @ system wall clock time = 9:00:00
2. m2: (offset=2, value=5, time=9:00:01) @ system wall clock time = 9:00:59
3. m3: (offset=3, value=9, time=8:59:30) @ system wall clock time = 9:02:01

The first example is for a tuple-based window. If the tuple-based window has the window size 3 and is also configured to send update every 1 minute. Consider the aggregation operator is a "max", the "max" operator first receives window outputs {wnd=[1,3], values={m1, m2}} and compute a result {wnd=[1,3], value=5}. Later, m3 arrives and window operator sends another update {wnd=[1, 3], values={m3}} and the "max" operator will update the result to {wnd=[1,3], value=9}.

The second example is for a time-based window. If the time-based window has a window size of 1 minute and is also configured to send update every 1 minue. Using the same "max" operator as an example, the "max" operator first receives window outputs (wnd=[8:59:00,8:59:59], values={m1}} and (wnd=[9:00:00,9:00:59], values={m2}} and calculates two results (wnd=[8:59:00,8:59:59],value=0) and (wnd=(9:00:00,9:00:59], value=5). Later, when m3 arrives, the "max" operator receives window output {wnd=[8:59:00,8:59:59], values={m3}} and will re-compute the result {wnd=[8:59:00, 8:59:59], value=9}.

Case 2: windowed join

When the window operator's output is used to perform a join w/ another stream, it would require accessing each stream's past window states. Consider augmenting the sequence of messages above with an additional field id:

Orders:

1. m1: (offset=1, value=0, id=1, time=8:59:10) @ system wall clock time = 9:00:00
2. m2: (offset=2, value=5, id=3, time=9:00:00) @ system wall clock time = 9:00:59
3. m3: (offset=3, value=9, id=9, time=8:59:30) @ system wall clock time = 9:02:01

And another stream Shipments has the following sequence:

Shipments:

1. m1: (offset=1000, cost=0, id=3, time=9:00:10) @ system wall clock time = 9:00:00
2. m2: (offset=1001, cost=2, id=1, time=9:00:10) @ system wall clock time = 9:00:10
3. m3: (offset=1002, cost=3, id=1, time=9:01:10) @ system wall clock time = 9:01:01
4. m4: (offset=1003, cost=1, id=9, time=9:01:20) @ system wall clock time = 9:01:01

Now, the join is the following for a tuple-based window:

SELECT id, value, cost FROM Shipments JOIN Orders OVER (ROWS 3 PRECEDING) ON Orders.id = Shipments.id

Assuming that the default windowing on a stream is (ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) and let's assume the tuple window for Orders is configured in the same way as in the aggregation case. Now, at 9:01:00, Orders send an output {wnd=[1, 3], values={m1,m2}} and Shipments has {wnd=(-INFINITE, 1001], values={m1, m2}}. Assuming the join operator always takes Orders output first, then Shipments, the following result would be processed: (id=3, value=5, cost=0) (id=1, value=0, cost=2)

By 09:02:00, Shipments have new messages arrived {m3, m4}. The join operator now tries to update the output w/ the Orders window, which has {wnd=[1, 3], {m1,m2}}. The following result is added to the join output: (id=1, value=0, cost=3)

At 9:03:00, Orders has one more message and yields additional output: {wnd=[1, 3], {m3}}. Now, the additional output is a join between past window output from Orders {wnd=[1, 3], values={m3}} with the unbounded window in Shipments {wnd=(-INFINITE, 1003], values={m1, m2, m3, m4}}. And that would generate the following additional output: (id=9, value=9, cost=1)

NOTE: the above result is only consistent if we always align the starting offsets of Orders and Shipments as (Orders.offset=Shipments.offset - 999), which we will record when initialize the job.

Now let's look at a time-based window:

SELECT id, value, cost FROM Shipments JOIN Orders OVER (ORDER BY time RANGE '2' min PRECEDING) ON Orders.id = Shipments.id

Assuming that the default windowing on a stream is (ORDER BY time RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) and let's assume that the time window for Orders has window size 2 minute and is configured to send update every 2 minute. Given the same sequence of event, at time 9:00:59, Shipments has the following window: {wnd=(-INFINITE,9:00:59], values={m1, m2}} and Orders has the window: {wnd=[8:59:00, 9:00:59], values={m1, m2}}, which yields the following result: (id=3, value=5, cost=0) (id=1, value=0, cost=2)

Now at 9:01:59, Shipments has the following window: {wnd=(-INFINITE, 9:01:59], values={m1, m2, m3, m4}}. Join with the Orders window {wnd=[8:59:00, 9:00:59], values={m1, m2}} yields an additional result. (id=1, value=0, cost=3)

At 9:02:59, Orders received m3 and updated the past window with additional output: {wnd=[8:59:00, 9:00:59], {m3}). Joining with the Shipments window (wnd=(-INFINITE, 9:01:59], values={m1, m2, m3, m4}}, one last result is generated: (id=9, value=9, cost=1)

NOTE: w/o the implicit time condition that Orders.time = Shipments.time, the alignment of Orders windows vs Shipments windows is not defined.

As a summary to the above examples, we decided to leave the logic of which Shipments windows need to join with which Orders windows to the join operator. As long as join operator determines the alignment between windows in different streams based on join conditions (e.g. Orders.time=Shipments.time), window operator will simply provide a method to access the window state in a specific stream for join.

Proposed Basic Windowing Model

From the above use cases, we propose the basic windowing model as the following:

1. The window operator keeps a large internal state (i.e. a materialized view of the stream) that covers all the past windows within a retention policy
    1. The retention policy should be both size/event time based to ensure a reasonable amount of history data is kept in the window state
2. The window operator will send one or multiple window outputs of the same window, based on different window update policies
    1. Early emission policy that allows the window operator to send **current window** outputs based on system wall clock before the window is considered full.
    2. Full size policy that allows the window operator to determine the window size is full based on messages received (e.g. number of messages, event time field in the messages, etc.), which will trigger a window update for the corresponding window.
    3. Late arrival policy that allows the window operator to send **past window** outputs based on late arrival messages.

In this model, we would define **current window** as the latest window we just opened, and all previous windows are **past windows**.

The above model ensures that:

1. Past windows are available in the window state for re-computing the window outputs
2. Each window may have multiple timely updates due to
    1. Early emission policy
    2. Full size policy
    3. Late arrival policy

Semantic Difference between Join and Aggregator Windows

There are several important semantic differences between join and aggregator windows:

1. Aggregator window needs to update all windows that include an incoming message s.t. all affected aggregation results are refreshed.
    1. I.e. a single message in a stream **MUST** be included in multiple sliding windows and should be counted toward all windows' aggregation results.
    2. The aggregation result is keyed per each window and most of the aggregation functions are incremental updatable (e.g. "count", "max") which does not require the full list of messages in the past. This would lead to important optimization to the data store used in window operators.
2. Join window should only send window output that include the incoming message for the first occurence.
    1. When a join occurs between stream A and B, with sliding windows, we could have multiple windows that include a single message in stream A. However, join of the same message from stream A to messages in stream B should yield the same result. I.e. when join a single message from stream A to stream B, only the window in stream B is used to bound the join range. Different windows in stream A associated with the same message should not create multiple join results. Hence, the window operator shoould only send the first occurence of a message in stream A in the sliding windows to avoid re-computation.

This semantic difference will lead to the difference in implementation of the window algorithms described later.

Basic Data Structure in Window Operators

There are the following data structure used by each window operator:

1. Message Store: the window internal state provides the data set to re-compute the window output. It keeps messages in all windows within retention. Note this message store keeps an exact-once materialized view of the incoming messaging stream.
2. Window Metadata: this marks advancement of windows. Each window metadata is a key-value pair. The key and value are different for tuple and time based windows
    1. For tuple-based window, the key is the start offset and the value is the start offset and the last offset of the window.
    2. For time-based window, the key is the start time and the value is the start and last offset of the window and the start and end timestamp of the window
3. Window outputs: it keeps a list of tuples as the output of the certain window. The content of a window output varies between aggregator and join windows:
    1. For aggregator windows, the window output is a list of tuples that are keyed by each window, and the value is the updated aggregation result
    2. For join windows, the window output is just a list of messages that are newly added to the window

NOTE: The initial window metadata for each stream also can be used by join to identify the alignment between the tuple-based windows in different streams, as mentioned in the tuple-window join example in case 2.

Order of updates to the message store and the window metadata:

1. For each message, save to the message store and update window metadata (i.e. update the last offset).
2. When a window output is sent out, flush the message store and window metadata, checkpoint

the current offset.
3. When a new window is opened, flush the message store and the window metadata, checkpoint the current offset.
4. When the message store is purged due to retention policy (i.e. the start offset of the first window may be updated), flush the message store and the window metadata, and checkpoint the current offset.

The above sequence of state update will guarantee the following:

1. When message store is flushed and available, the current window output is always recoverable from the message store and messages can be replayed from the latest checkpoint.
2. When the message store is lost, we can always replay the messages from the start offset in earliest window metadata to restore the window state.
    1. To shorten the time of recovery in this case, a lazy recovery may be used. I.e. the window operator can start functioning before fully recovered the message store and correct the results as more messages are recovered. Detailed discussion is postponed.

Time-based Window

In a time-based window, the following are the issues:

1. Messages may arrive out-of-order in terms of event time
2. Latencies between messages may be arbitrarily long

Hence, we would have three different types of messages in a time-based window:

1. **in-time message**: the message's event time is in the range of [wnd.start_time, wnd.end_time] of the **current window**
2. **past message**: the messages event time is strictly less than the **current window**'s start time
3. **future message**: the message event time is strictly greater than the **current window**'s end time

An illustration of Message Store and Window Metadata for time-based window is below:

tumbling window with out-of-order messages

```
+--------------------------------------------------------------+
|                                                              |
|                          past_n.last_offset                  |
|                               |                              |
|      current window           |     |                        |
|              |  past   window1|     |                        |
|              |     .....    | |past window_n    |            |
|   ---+---+---+---+......+---+---+---+---+---+---+            |
|      | - | / | L |      |   |   | F |   |   |   |            |
|   ---+---+---+---+......+---+---+---+---+---+---+            |
|          |<---  Message          Store     ---> |            |
|            |                      |           |              |
|            |                      |      past_n.start_offset |
|            |                past1.start_offset               |
|            |                      .                          |
|            |                      .                          |
|            |                      .                          |
|      cur.last_offset        cur.start_offset                 |
|                                                              |
+--------------------------------------------------------------+
```

hopping window and **sliding window** w/ out-of-order messages

```
+--------------------------------------------------------------+
|                                                              |
|                                                              |
|                                                              |
|                  (past window output)                        |
|                           |                                  |
|                     current window      |                    |
|                           | |past   window1 |                |
|                           | |     ....      |                |
|                           | |   |  past   window_n  |        |
|   ---+---+---+---+..+---+---+---+---+---+---+---+---+        |
|      | / | F | / |  | - | L |   |   | F |   |   |   |        |
|   ---+---+---+---+..+---+---+---+---+---+---+---+---+        |
|      |<---   Message          Store   |   |   ----> |        |
|        |                          |   |   |       |          |
|        |              past_n.last_offset  past_n.start_offset|
|        |                              |   |                  |
|        |                            past_1.start_offset      |
|        |                              |                      |
|   cur.last_offset               cur.start_offset             |
|                                                              |
+--------------------------------------------------------------+

  +---+
  | / | insertion tuples in the current window output
  +---+

  +---+
  | F | future messages in window state
  +---+

  +---+
  | L | past messages as late arrivals
  +---+

  +---+
  | - | past messages that are out-of-retention policy (discarded)
  +---+
```

Basic Tuple-based Window

In tuple-based window, the window metadata is simpler and only uses start_offset as the window key.

Further optimization is possible in tuple-based window since we can actually determine the end of the window and avoid any late arrival updates. We will save it for another discussion, since most of our important use cases are time-based windows.

tumbling window

```
+-----------------------------------------------------------+
|        cur.last_offset       past.last_offset             |
|            |                        |                     |
|       current window |    ....    | |past window  |       |
|       ---+---+---+---+....+---+---+---+---+---+---+       |
|          | / | / | / |    |   |   |   |   |   |   |       |
|       ---+---+---+---+....+---+---+---+---+---+---+       |
|          |         |                            | |       |
|          |<---   Message         Store       ---> |       |
|                    |                            |         |
|             cur.start_offset            past.start_offset |
|                                                           |
+-----------------------------------------------------------+
```

hopping window and **sliding window**

```
+-----------------------------------------------------------+
|                                                           |
|                                                           |
|                                 past_n.last_offset        |
|                 current window |     |                    |
|                  |   past window   | |                    |
|                       .....    |   past window_n   |      |
|       ---+---+---+---+.....+---+---+---+---+---+---+      |
|          | / | / |   |     |   |   |   |   |   |   |      |
|       ---+---+---+---+.....+---+---+---+---+---+---+      |
|          |                   |                   | |      |
|          |<---   Message     |    Store       ---> |      |
|                              |                   |        |
|                  cur.start_offset      past_n.start_offset|
|                                                           |
+-----------------------------------------------------------+


  +---+
  | / | insertion tuples in the window output
  +---+
```

NOTE: the last message offset in the flushed window state will not be later than the last message offset in the window output, any offset persisted in the window metadata will not be later than the last message flushed into the window state.

Window Metadata

We choose to use a KV-store with changelog to store window metadata due to the following reason:

1. With windows of dynamic length (i.e. session windows), we cannot deduce the start_time or start_offset of a window from the earliest window metadata unless we replay the whole stream.

2. Each window metadata for a fixed time window include start_offset and start_time. It is easy to deduce start_time of a later window, but hard to reason about the start_offset unless we replay the whole stream.
3. For optimization of windowed aggregator that do not need to keep individual messages in a window (e.g. "count", "max", and "min"), KV-store for Window Metadata can be used to store the result of windowed aggregation as well, completely remove the need for Message Store.

Comparative, using a changelog attached KV-store to store all past window metadata will only pay the cost to scan the changelog when recover, which should be much smaller in size when the number of windows in retention is much smaller than the number of messages.

Hence, window metadata store should be implemented as:

1. All past window metadata are accessible via a window key
2. The value of the window metadata should include
    1. start_offset and last_offset
    2. start_time if the window is time-based window
    3. end_time if the window is dynamic length time window
    4. the aggregated value of the window (optional)
3. All window keys should be sorted in the order of expiration as per retention policy
    1. For time-based window, window expiration is always based on event time. Hence the window key should be start_time
    2. For tuple-based window, window expiration is always based on tuple/offset. Hence the window key should be start_offset and here we will have to assume that the offsets are comparable according to the message arrival orders.

NOTE: We will discuss the case when offsets are not sorted according to message arrival orders in another discussion.

Message Store

A KV-store for Message Store is not needed if the windowed aggregation result is incrementally updatable. Hence, it should only be used when:

1. The aggregator on the window requires the full state of the window. e.g. "median"
2. The join on the window requires the full state of the window

With the above use case in mind, message store needs satisfy the following requirements:

R.1. All past window messages in retention are accessible via a range query R.2. A fast way to access the messages within the past windows R.3. A fast way to drop "expired" messages according to retention policy R.4. The mapping from Kafka messages to the Message Store should be exact-once, i.e. replay and duplicated messages are not stored multiple times in the message store.

Here are the list of factors to be considered in designing the key of the message store:

1. Time vs tuple based window
2. Aggregate vs Join
3. Join condition w/ equal fields
4. Join condition w/ time range on time-based window

5. Window Size

In general, we can use the following matrix to determine the **message_key**:

Time or Tuple	Aggregate or Join	Join w/ equals	Join w/ time range	Window Size	Message Key
Tuple	Aggregate	N/A	N/A	N/A	<offset>
Tuple	Join	No	N/A	N/A	<offset>
Tuple	Join	Yes	N/A	Small	<offset>
Tuple	Join	Yes	N/A	Large	<equal_flds>_<offset>
Time	Aggregate	N/A	N/A	N/A	<timestamp>_<offset>
Time	Join	No	No	N/A	<timestamp>_<offset>
Time	Join	No	Yes	N/A	<timestamp>_<offset>
Time	Join	Yes	No	Small/Large	<equal_flds>_<timestamp>_<offset>
Time	Join	Yes	Yes	Small	<timestamp>_<equal_flds>_<offset>
Time	Join	Yes	Yes	Large	<equal_flds>_<timestamp>_<offset>

The default choice of **message_key** for tuple window is <offset> and for time window is <timestamp>_<offset>

The reason of the above choices are:

1. For all time-based window, the **windowKey** is the start_time of the window. Since all time-based window needs to access the messages according to the time range, all **message_key** starts with <timestamp>.
   1. Range query to access all messages in a window, the range scan starts from <wnd.start_time>_<wnd.start_offset> (inclusively) to <wnd.end_time>_<wnd.last_offset> (inclusively)
   2. Range query to expire messages according to retention policy. When the retention policy determines that we need to expire old messages, in time-based window, we always retires the messages earlier than an timestamp. Hence, the range scan starts from the earliest window metadata's <start_time>_<start_offset> (inclusively) to <retention.new_starttime>_MINOFFSET (exclusively).
   3. Access messages within a window via an iterator according to the full key order (i.e. <timestamp>_<offset>).
2. For all tuple-based window, the **windowKey** is the start_offset of the window. The **message_key** is used as the following
   1. Range query to access all messages in a window from wnd.start_offset (inclusive) to wnd.last_offset (inclusive)
   2. Range query to expire messages according to retention policy. When the retention policy determines that we need to expire old messages, in tuple-based window, we always retires the messages earlier than an offset. Hence, the range scan starts from the earliest window metadata's start_offset (inclusively) to retention.new_offset (inclusively).
   3. Access messages within a window via an iterator according to offset.

The optimization in the presence of equal join conditions:

Case 1 Time windowed join w/ equal conditions and no time conditions. In this case, for each join, the potential number of past windows included in the join is unbounded. Hence, the **message_key** has the <equal_flds> first.

1. Range query to access all messages in a window w/ equal conditions starts from <equal_flds>_<wnd.start_time>_<wnd.start_offset> (inclusive) to <equal_flds>_<wnd.end_time>_<wnd.last_offset> (inclusive)
2. Range query to access all messages in all windows: prefix scan with <equal_flds>_ as the prefix
3. Expiration of messages according to retention policy would need to use lazy cleanup: for all keys w/ expired event time in a prefix scan <equal_flds>_, delete

Case 2 Time windowed join w/ equal conditions, time range conditions, and a large window size. In this case, for each join, iterating through a large window size is not optimal and we apply the same optimization as in Case 1 above.

Case 3 Tuple windowed join w/ equal conditions and large window size. In this case, for each join, iterating through a large window size is not optimal. Hence, the **message_key** has the <equal_flds> first.

1. Range query to access all messages in a window w/ equal conditions starts from <equal_flds>_<wnd.start_offset> (inclusive) to <equal_flds>_<wnd.last_offset> (inclusive)
2. Expiration of messages according to retention policy would need to use lazy cleanup: for all keys w/ expired offsets in a prefix scan <equal_flds>_, delete

Session Window

The case of session window is often illustrated by the following use case:

1. There is a session key to identify such a window, or windows if we allow multiple session windows to be opened over time
2. A session window is opened when a specific session key occurs the first time in the stream
3. A session window will have a session termination policy to determine the end of the window

We will refer to the window with dynamic session length as **session window** later.

Hence, the extension to a session window is:

1. Each session window will have a window metadata that has the following key
    1. For tuple-based window, <session_key>_<start_offset>
    2. For time-based window, <session_key>_<start_time>
2. Each session window has a session termination policy that is based on incoming messages w/o the corresponding <session_key>
    1. For tuple-based window, it is a max number of messages w/o the corresponding <session_key> in the session window
    2. For time-based window, it is a max event time difference between the last and new message w/ the same <session_key> in the session window

Similarly, Message Store would need to have the <session_key> prefix to differentiate messages belong to different session windows as well, for aggregates and join w/ condition on <session_key>s. The performance impact is that the expiration of old messages now has to be done per

<session_key>.

Note: In a join which does not include <session_key> in join condition, it does not make sense to have dynamic session windows either, since join from one stream now needs to look into each and every dynamic sessions from the other stream, which is equivalent to a join w/o the session windows based on <session_key>.

Class Hierarchy for Different Types of Windows

We define the following class hierarchy for different types of windows, due to their implementation choices:

```
+-------------------------------------------------------------------+
|                                                                   |
|                 +----------------------+                          |
|                 |   Window Operator    |                          |
|                 +----------------------+                          |
|                  /            |     \                             |
|    +----------------------+   |   +----------------------+        |
|    |  Windowed Aggregator |   |   |   Join Window Op     |        |
|    +----------------------+   |   +----------------------+        |
|                               |                                   |
|                 +----------------------+                          |
|                 |  Session Window Op   |                          |
|                 +----------------------+                          |
|                  /                   \                            |
|    +----------------------+       +-------------------------+     |
|    |  Session  Aggregator |       |  Session Join Window Op |     |
|    +----------------------+       +-------------------------+     |
+-------------------------------------------------------------------+
```

Block Diagram of Window Operator Design

The following are the window operator design choices:

Windowed Aggregator

This is to illustrate the overall design of a window operator that has an aggregator function integrated.

The first design illustrate the implementation of a windowed aggregator that does not need the full list of messages in the past window. In this design, there is no need to have a Message Store to hold all past messages for later computation. The aggregated result is stored together w/ the Window Metadata.

```
+---------------------------------------------------------------------------------+
|                                                                                 |
|                +--------------------------------------------------+             |
|                |          Windowed Aggregator                     |             |
|   +---+        |              +--------------------------+         |             |
|   | m | ---> addMessage()     |        Window Config     |         |             |
|   +---+        |       |      +------+-----+------+------+         |             |
|                |       |      |Early |Late | Full |Reten-|         |             |
|                |       |      |Emiss.|Arriv| Size |tion  |         |             |
|              timeout()\|/     +------+-----+------+------+         |   To        |
|                |      +------+           /|\                       | Next Op     |
|                |      | Aggr |<-+-------- | ----------------------->[([sessKey,]|
|    To          |      | Func |  |         |                        |  wndKey,    |
|   User         |      +------+ \|/        |                        |  result),   |
|[([sessKey,]<-getResult()       getWindow([sessKey,]wndKey)         | ...]        |
|  wndKey,       |  /|\          putWindow([sessKey,]wndKey,val)     |             |
|  result),      |   |            delWindow([sessKey,]wndKey)        |             |
| ...]           |   |                      |                        |             |
|                |   |                     \|/                       |             |
|                |   |            +-----------------------+          |             |
|                |   |            |  Window Meta Store    |          |             |
|                |   +----------- |                       |          |             |
|              flush() ---------> |                       |          |             |
|                |                +-----------------------+          |             |
|                |                                                   |             |
|                +---------------------------------------------------+             |
|                                                                                  |
|                                                                                  |
+----------------------------------------------------------------------------------+
```

The second option is a more complicated full-state windowed aggregator that requires the list of messages in the past window:

```
+----------------------------------------------------------------------------------+
|                                                                                  |
|               +----------------------------------------------------+             |
|               |      Full State Windowed Aggregator                |             |
|  +---+        |                +--------------------------+        |             |
|  | m | ---> addMessage()       |       Window Config      |        |             |
|  +---+        |        |       +------+-----+------+------+        |             |
|               |        |       |Early |Late | Full |Reten-|        |             |
|               |        |       |Emiss.|Arriv| Size |tion  |        |             |
|             timeout() \|/      +------+-----+------+------+        |   To        |
|               |      +------+           /|\                        | Next Op     |
|               |      | Aggr |----------- | ------------------------->[([sessKey,]|
|    To         |      | Func |            |                         |  wndKey,    |
|   User        |      +------+            |                         |  result),   |
|[([sessKey,]<-getResult()  /|\   delWindow([sessKey,]wndKey)        | ...]        |
|  wndKey,      |   /|\     +-->putWindow([sessKey,]wndKey,val)      |             |
|  result),     |    +----- | -  getWindow([sessKey,]wndKey)         |             |
| ...]          |           |              |                         |             |
|               |           |             \|/                        |             |
|               |           |    +-----------------------+           |             |
|             flush() --+------> |                       |           |             |
|               |       |   |    |  Window Meta Store    |           |             |
|               |       |   |    +-----------------------+           |             |
|               |       |   |                                        |             |
|               |       |   |                                        |             |
|               |      \|/ \|/                                       |             |
|               |   getMessages([sessKey,]startKey,endKey)          |             |
|               |   addMessage([sessKey,]msg)                       |             |
|               |     retire([sessKey,]startKey,endKey)             |             |
|               |                       /|\                         |             |
|               |                        |                          |             |
|               |                       \|/                         |             |
|               |            +--------------------------+           |             |
|               |            |    Message Store         |           |             |
|               |            |                          |           |             |
|               |            +--------------------------+           |             |
|               |                                                   |             |
|               +---------------------------------------------------+             |
|                                                                                 |
+---------------------------------------------------------------------------------+
```

Join Window Operator

This is to illustrate the overall design of a window operator that will provide both delta updates and full-state access to windows.

```
+---------------------------------------------------------------------------------+
|                                                                                 |
|                +---------------------------------------------------+            |
|                |      Full State Join Window Op                    |            |
|   +---+         |                 +--------------------------+      |            |
|   | m | ---> addMessage()<-+     |       Window Config      |      |            |
|   +---+         |     /|\   |     +------+-----+------+------+      |            |
|                |      |    |     |Early |Late | Full |Reten-|      |            |
|                |      |    |     |Emiss.|Arriv| Size |tion  |      |            |
|              timeout()|    |     +------+-----+------+------+      |            |
|                |      |    |             /|\                       |            |
|    To          |      |    |              |                        |            |
|   User         |      |    |              |                        |            |
|[([sessKey,]<-getResult()   |   getWindow([sessKey,]wndKey)         |            |
|  msg),         |   /|\|    +-->putWindow([sessKey,]wndKey,val)     |            |
| ...]           |    | |         delWindow([sessKey,]wndKey)        |            |
|                |    | |                   |                        |            |
|                |    | |                  \|/                       |            |
|                |    | |         +----------------------+           |            |
|              flush() ----+----> |                      |           |            |
|                |    | |  |      |  Window Meta Store   |           |            |
|                |    | |  |      +----------------------+           |            |
|                |    | |  |                |                        |            |
|  Windows       |    | |  |               \|/                       |            |
|[([sessKey,]<-getWindows([sessKey,]startKey,endKey)                 |            |
|  wndKey,       |    | |  |                                         |   To       |
|  wndMeta),     |    |\|/\|/                                        | Next Op    |
| ...]           |getMessages([sessKey,]stKey,endKey[,joinKeys])     |[([sessKey,]|
|                | addMessage([sessKey,]msgKey,[joinKeys,]msg)  ----->  msg),     |
|                |    retire([sessKey,]stKey,endKey,[joinKeys,])     | ...]       |
|                |      |                /|\                         |            |
|                |      |                 |                          |            |
|                |      |                \|/                         |            |
|                |      |     +--------------------------+           |            |
|                |      |     |    Message Store         |           |            |
|                |      |     |                           |           |            |
|                |      |     +---------------------------+           |            |
|  Msgs          |     \|/                                            |            |
|[([sessKey,]<-getMessages([sessKey,]startKey,endKey[,joinKeys])     |            |
|  msg),         |                                                    |            |
| ...]           |                                                    |            |
|                |                                                    |            |
|                +----------------------------------------------------+            |
|                                                                                  |
+----------------------------------------------------------------------------------+
```

In the above full-state join window operator, there are two additional interface methods to allow users to access windows and the messages:

1. getWindows: this is to allow the users to get the list of windows' metedata that contains the range specified in the parameters.
2. getMessages: this is to allow the users to get the list of messages within the range specified in the parameters.

Configuration of A Windowed Aggregator

The following configuration is required for a windowed aggregator:

1. Aggregator function
2. Window Size Measurement Unit
3. Window Size
4. Window Advance Step Size
5. Window Retention Size
6. Window Retention Time

The following configuration is optional for a windowed aggregator:

1. Early emission policy: Default is disabled. We only want one aggregated result when window size is full.
2. Full size policy: Default to the first message that exceeds the end boundary of the corresponding window and is greater than the window's start_offset (i.e. the end boundary would be end_time for time-based window)
3. Late arrival policy: Default to batching the updates for 10 seconds before send out the updated aggregation result

Basic Algorithm for Windowed Aggregators

The windowed aggregator implements the following APIs:

1. addMessage: A message is received
2. timeout: A timer is triggered
3. getResult: For standalone usage that allows the user to directly get the window output if it is ready
4. flush: For standalone usage that allows the user to explicitly call flush to save the window metadata and message store to disk

Adding Message to Windowed Aggregator

For operators that are executed in an automatic execution context, the windowed aggregator will send the updated aggregation result to the downstream operator automatically, while in a standalone context, the windowed aggregator will keep a copy of the updated aggregation result until the user explicitly retrieves the result and flush the state.

Automatic addMessage procedure is the following:

1. Check whether the message selector is disabled for this window operator
    1. If yes, log a warning and throws exception
2. Check whether the message is out-of-retention
    1. If yes, send the message to error topic and return
3. Check to see whether the message is a replay that we have already sent result for
    1. If yes, return as NOOP
    2. Otherwise, continue
4. Check to see whether the incoming message belongs to a new window
    1. If yes, Call openNextWindow() and repeat
    2. Otherwise, continue
5. Add incoming message to all windows that includes it and add those windows to pending updates list
6. For all windows in pending updates list, do the following
    1. updateResult() to refresh the aggregation results
    2. Check to see whether we need to emit the corresponding window outputs
        1. If the window is the current window, check with early emission policy to see whether we need to flush aggregated result
            1. If yes, add this window to pending flush list and continue
        2. For any window, check the full size policy to see whether we need to send out the window output
            1. If yes, add this window to pending flush list and continue
        3. For a past window, check with late arrival policy to see whether we need to send out past window outputs
            1. If yes, add this window to pending flush list and continue
7. Send out pending windows updates
    1. For each window in pending flush list, send the updates to the next operator
        1. If window outputs can't be accepted, disable the message selector from delivering messages to this window operator and return
    2. Finally, flush the Window State Store and Window Mark
        1. If retention limit is triggered, purge the out-of-retention windows/messages from Message Store and Window Meta Store before flushing

Helper function openNextWindow()

1. If there is any new messages in **current window**, add **current window** to pending updates list
2. Create the next window and add all **in-time** messages to the next window
3. set next window to **current window**, return

Note: The above function will disable the message delivery if the downstream operator refuses to take the window outputs. The re-enabling of message delivery is done in the timeout function below.

Standalone addMessage procedure is similar:

1. Run Step 2-6 in Automatic Window Operator addMessage()

Timeout w/o Incoming Message to Window

Automatic timeout procedure:

1. Check whether there is pending flush list for window outputs
    1. If yes, try sending the window outputs from where it stopped before (i.e. Step 7 in addMessage())
        1. If not completed, return and wait for the next round.
        2. Otherwise, re-enable the message selector to delivery messages to this window and continue
2. For all windows w/ pending outputs, repeat Step 6.2 in addMessage()
3. Run Step 6 in addMessage()

Standalone Windowed Aggregator APIs

The standalone window operator will implement two additional interface functions:

1. getResult: The programmer explicitly calls getResult to get the current window outputs
2. flush: Allow the programmer to explicitly flush the Window State Store and Window Mark

The following will be the basic implementation.

getResult from a Window

1. Copy and return all window outputs

flush

1. Clear all window outputs
2. Run Step 7.2 in Automatic Window Operator to flush the Message Store and Window Meta Store

Configuration of A Join Window Operator

The following configuration is required for a join window operator:

1. Window Size Measurement Unit
2. Window Size
3. Window Advance Step Size
4. Window Retention Size
5. Window Retention Time

The following configuration is optional for a join window operator:

1. setKeyedMessageStore(joinKeys): Default is disabled. Setting this means to use a hash-key on the joinKeys as the prefix to **message_key**
2. Early emission policy: Default is set to emit for each incoming message to get join result as soon as possible.
    1. Periodic configuration: setOntimeUpdatePeriod(10) set the window to send out window

    output every 10 sec, if there is any pending updates
3. Full size policy: Default to the first message that exceeds the end boundary of the corresponding window (i.e. the end boundary would be end_time for time-based window)
4. Late arrival policy: Default to send updates for each late arrivals
    1. Periodic configuration: setLateUpdatePeriod(10) set the window to send out window output every 10 sec for late arrivals

Basic Algorithm for Join Window Operators

The join window operator implements the following APIs:

1. addMessage: A message is received
2. timeout: A timer is triggered
3. getResult: For standalone usage that allows the user to directly get the window outputs if it is ready
4. flush: For standalone usage that allows the user to explicitly call flush to save the window metadata and message store to disk
5. getWindows: Get all windows that covers the intented join scope
6. getMessages: Get all messages that are in a windowed range

Adding Message to Join Window Operator

Automatic addMessage procedure is the following:

1. Run Windowed Aggregator addMessage Step 1
2. Run Windowed Aggregator addMessage Step 2-4
3. Add incoming message to the first window that includes it
4. Run Windowed Aggregator addMessage Step 6
5. Run Windowed Aggregator addMessage Step 7

Helper function openNextWindow() has a modification:

1. If there is any new messages in **current window**, add **current window** to pending updates list
2. Create the next window and set next window to **current window**, return

Standalone addMessage procedure is similar:

1. Run Step 2-5 in Join Window Operator addMessage()

Timeout w/o Incoming Message to Window

Automatic timeout procedure:

1. Same steps as in Windowed Aggregator timeout()

Get Windows

The join window operator allows the user to query all the windows that covers a range of keys via the following API:

1. getWindows(startTime, endTime): returns a sorted list of windows that covers the whole range

startTime to endTime
2. getWindows(sessKey, startTime, endTime): return a sorted list of session windows that overlaps w/ the range startTime to endTime

Get Messages

The join window operator allows the user to query all the messages within a range via the following APIs:

For time-based windows:

1. getMessages(startTime, endTime): returns a list of messages that falls into the specified time ranges: <startTime> to <endTime>
2. getMessages(startTime, endTime, joinKeys): returns a list of messages that falls into the specified time range: <join_flds>_<startTime> to <join_flds>_<endTime>
3. getMessages(sessKey, startTime, endTime): returns a list of messages that falls into the specified time ranges: <sessKey>_<startTime> to <sessKey>_<endTime>
4. getMessages(sessKey, startTime, endTime, joinKeys): returns a list of messages that falls into the specified time range: <sessKey>_<join_flds>_<startTime> to <sessKey>_<join_flds>_<endTime>

For tuple-based windows:

1. getMessages(startOffset, endOffset): returns a list of messages that falls into the specified offset ranges: <startOffset> to <endOffset>
2. getMessages(startOffset, endOffset, joinKeys): returns a list of messages that falls into the specified offset range: <join_flds>_<startOffset> to <join_flds>_<endOffset>
3. getMessages(sessKey, startOffset, endOffset): returns a list of messages that falls into the specified offset ranges: <sessKey>_<startOffset> to <sessKey>_<endOffset>
4. getMessages(sessKey, startOffset, endOffset, joinKeys): returns a list of messages that falls into the specified offset range: <sessKey>_<join_flds>_<startOffset> to <sessKey>_<join_flds>_<endOffset>

Standalone Join Window Aggregator APIs

The standalone window operator will implement two additional interface functions:

1. getResult: The programmer explicitly calls getResult to get the current window outputs
2. flush: Allow the programmer to explicitly flush the Window State Store and Window Mark

The following will be the basic implementation.

getResult from a Window

1. Copy and return all window outputs

flush

1. Clear all window outputs
2. Run Step 7 in join window operator addMessage() to flush the Message Store and Window Meta Store