Window Operator Design

This page summarize the design of the stateful window operator, related to SAMZA-552. This operator is primarily used for stream-stream join, in which streams are windowed (we do not support infinite window in Samza).

Class APIs

As described in this document, there are two classes extending the WindowOperator interface: AggregatedWindowOperator and FullStateWindowOperator. The difference between these two operators is that for AggregatedWindowOperator you cannot access to its windowed messages, whereas for FullStateWindowOperator you can. This feature of FullStateWindowOperator is necessary for operations such as joins, etc.

The classes are define as the following:

```
class OperatorSpec {

  List<StreamName> inputStreams;

  List<StreamName> outputStreams;

  ...
}

class WindowOperatorSpec extends OperatorSpec {

  enum SizeUnit { TIME_SEC, TIME_MS, TUPLE .. }

  enum Type { FIXED_WND, SESSION_WND .. }

  int windowSize;

  int stepSize;

  SizeUnit sizeUnit;

  Type type;

  RetentionPolicy retention;

  MessageStorageSpec msgStorageSpec;

  ...
}

interface Operator {
  // Initialize the operator
  void init();

  // Process a new input tuple for the operator
  void process(Tuple, MessageCollector, TaskCoordinator);
}

interface SimpleOperator extends Operator {
  // Get the operator specs
  OperatorSpec getSpec();
}

interface CompositeOperator extends Operator {
  // Add a new operator into this composite operator
  void addOperator(SimpleOperator)
}

class WindowOperator(WindowOperatorSpec) extends SimpleOperator {

  // Refresh the result when the window timer expires
  void refresh(Long, MessageCollector, TaskCoordinator);

  /* The following functions can be extended by users */

  // Called before processing the input tuple
  Tuple beforeProcess(Tuple, MessageCollector, TaskCoordinator) {}

  // Called before the result tuple is sent to the next task or operator
  Tuple beforeSend(Tuple, MessageCollector, TaskCoordinator) {}
}

class AggregatedWindowOperator extends WindowOperator {

  @Override
  protected void refresh(Long, MessageCollector, TaskCoordinator) {
    // For all output pending windows, check if
    //   1. it has scheduled output according to early emission policy
    //   2. its size has reached (time or tuple) according to size policy
    //   3. it has late arrived tuple according to late arrival policy
    // then update its output stream according to the aggregate function;

    // For generated output stream, call this.beforeSend() then call collector.send()
  }

  @Override
  protected void process(Tuple, MessageCollector, TaskCoordinator) {
    // 1. Call this.beforeProcess()
    // 2. Add an incoming message to all windows that includes it
    // 3. For those windows with new message add to output pending windows
    // 4. Call this.refresh()
  }
}

class FullStateWindowOperator extends WindowOperator {

  @Override
  protected void refresh(Long, MessageCollector, TaskCoordinator) {
    // For all output pending windows, check if
    //   1. it has scheduled output according to early emission policy
    //   2. its size has reached (time or tuple) according to size policy
    //   3. it has late arrived tuple according to late arrival policy
    // then update its output stream by just keeping its unflushed messages;

    // For generated output stream with unflushed messages, call this.beforeSend() then call collector.send()
  }

  @Override
  protected void process(Tuple, MessageCollector, TaskCoordinator) {
    // 1. Call this.beforeProcess()
    // 2. Add an incoming message to all windows that includes it
    // 3. For those windows with new message add to output pending windows
    // 4. Call this.refresh()
  }

  // Get messages based on the range and the filter fields
  public List<Tuple> getMessages(Range<T>, List<String>);

  // Get messages based on the range only
  public List<Tuple> getMessages(Range<T>);
}
```

Case Studies

Here are a list of example window operations and how they can be implemented.

Windowed Aggregation

The following aggregation on stream Orders:

```
SELECT STREAM product, AVG(price) AS avg_price

FROM Orders

OVER (ORDER BY time RANGE '10' min PRECEDING)
```

Can be implemented as follows:

```
AveragePriceTask extends StreamTask with InitableTask {

  void init() {
    // operator spec should include the aggregation function
    WindowOperator orderWindows = new AggregatedWindowOperator(OperatorSpec);
    orderWindows.init();
  }

  void process(Tuple tuple) {
    // if there are any new results generated, they will
    // be sent to output stream automatically
    orderWindows.process(tuple);
  }
}
```

Stream-Stream Join

First note that unbounded stream-stream joins are not supported, i.e. join predicates must include timestamps from both streams. For example, the following stream-stream join will be rejected at parsing time.

```
SELECT STREAM o.time as time, o.id as id, a.value, s.cost

FROM Orders as o

JOIN Shipments as s

ON o.id = s.id
```

The following join on stream Orders and Shipments aligns timestamp boundary from the two streams:

```
SELECT STREAM o.time as time, o.id as id, a.value, s.cost

FROM Orders as o

JOIN Shipments as s

ON o.id = s.id

AND s.time > o.time AND s.time < o.time + 5 MIN
```

To support this, Samza first will have a StreamStreamJoinOperator implementation:

```
StreamStreamJoinOperator extends SimpleOperator {

  @Override
  void process(Tuple tuple) {
    // 1. Find join streams other than tuple.getStreamName from spec.inputStreams
    // 2. For each of these other streams, find the join set via winOp.getMessages(range(0, 5), field(id,
EQUALS))
    // 3. For the given join set, call this.join()
  }

  OutputStream join(List<StreamName, List<Tuple>>) {
    // join the tuples from input streams, put the result into output stream and return.
  }
}
```

Then user's code would look like sth. like:

```
OrderShipmentsJoinTask extends StreamTask with InitableTask {
  void init() {
    // Note that we should connect these three operators via specs. I.e.:
    //   OrderSpec.outputStreams = {"order"}
    //   ShipmentSepc.outputStreams = {"shipment"}
    //   JoinSpec.inputStreams = {"order", "shipment"}
    WindowOperator orderWindows = new FullStateWindowOperator(OrderSpec);
    WindowOperator shipmentWindows = new FullStateWindowOperator(ShipmentSpec);
    StreamStreamJoinOperator osJoin = new StreamStreamJoinOperator(JoinSpec);
    CompositeOperator join = new CompositeOperator();

    orderWindows.init();
    shipmentWindows.init();
    osJoin.init();
    join.addOperator(orderWindows);
    join.addOperator(shipmentWindows);
    join.addOperator(osJoin);
    join.init();
  }

  void process(Tuple tuple) {
    // if there are any new results generated, they will
    // be sent to output stream automatically
    orderWindows.process(tuple);
  }
}
```

Discussion

Here are some more notes regarding the above APIs:

1. We can put the beforeSend / beforeProcess into a Callback class to get better code re-usage.