Kafka Stream Usage Patterns

This page collects common usage pattern on how to use Kafka Streams Processor API or DSL.

Feel free to add your own code snippets and/or improve existing examples.

- How to compute an (windowed) average?
- How to compute windowed aggregations over successively increasing timed windows?
- How to aggregate data from all currently active sessions?
- How to manage the size of state stores using tombstones?
- How to purge data from KTables based on age

How to compute an (windowed) average?

Kafka Streams DSL aggregation natively support so-called "incremental" or "cumulative" aggregation functions (cf, https://en.wikipedia.org/wiki/Associative_property – also called "distributive functions" by Jim Gray in "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals") like count, sum, min, max. However, average function (a so-called "algebraic" function, cf. "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals") cannot be computed like this, but it can be composed of distributive functions, namely count and sum. Thus, it can be implemented in a two step approach:

Non-Windowed Average

```
class Tuple2<T1, T2> {
  public T1 value1;
  public T2 value2;

  Tuple2(T1 v1, T2 v2) {
    value1 = v1;
    value2 = v2;
  }
}

final KStreamBuilder builder = new KStreamBuilder();

// first step: compute count and sum in a single aggregation step and emit 2-tuples <count,sum> as aggregation
result values
final KTable<String,Tuple2<Long,Long>> countAndSum = builder.stream("someInputTopic")
                                                            .groupByKey()
                                                            .aggregate(
                                                                new Initializer<Tuple2<Long, Long>>() {
                                                                    @Override
                                                                    public Tuple2<Long, Long> apply() {
                                                                        return new Tuple2<>(0L, 0L);
                                                                    }
                                                                },
                                                                new Aggregator<String, String, Tuple2<Long,
Long>>() {
                                                                    @Override
                                                                    public Tuple2<Long, Long> apply(final
String key, final Long value, final Tuple2<Long, Long> aggregate) {
                                                                            ++aggregate.value1;
                                                                            aggregate.value2 += value;
                                                                            return aggregate;
                                                                    }
                                                                },
                                                                new Tuple2Serde()); // omitted for brevity

// second step: compute average for each 2-tuple
final KTable<String,Double> average = countAndSum.mapValues(
                                                    new ValueMapper<Tuple2<Long, Long>, Double>() {
                                                        @Override
                                                        public Double apply(Tuple2<Long, Long> value) {
                                                            return value.value2 / (double) value.value1;
                                                        }
                                                    });
```

This pattern can also be applied to compute a windowed average or to compose other algebraic functions.

How to compute windowed aggregations over successively increasing timed windows?

Status: *Draft*

```
KTable<Windowed<Key>, Value> oneMinuteWindowed = // where Key and Value stand for your actual key and value
types

        yourKStream

        .groupByKey()

        .reduce(/*your adder*/, TimeWindows.of(60*1000, 60*1000), "store1m");
                //where your adder can be as simple as (val, agg) -> agg + val
                //for primitive types or as complex as you need

KTable<Windowed<Key>, Value> fiveMinuteWindowed =
```

```
        oneMinuteWindowed
        .groupBy( (windowedKey, value) ->
                new KeyValue<>(
                        new Windowed<>(
                                windowedKey.key(),
                                new Window<>(
                                        windowedKey.window().start() /1000/60/5 *1000*60*5,
                                        windowedKey.window().start() /1000/60/5 *1000*60*5 + 1000*60*5
                            // the above rounds time down to a timestamp divisible by 5 minutes
                                )
                        ),
                        value
                ),
                /* your key serde */,
                /* your value serde */
        )
    .reduce(/*your adder*/, /*your subtractor*/, "store5m");
        // where your subtractor can be as simple as (val, agg) -> agg - val for primitive types
        // or as complex as you need,
        // just make sure you get the parameter order right, subtraction is not commutative!

KTable<Windowed<Key>, Value> fifteenMinuteWindowed =

        fiveMinuteWindowed

    .groupBy( (windowedKey, value) ->
                new KeyValue<>(
                        new Windowed<>(
                                windowedKey.key(),
                                new Window<>(
                                        windowedKey.window().start() /1000/60/15 *1000*60*15,
                                        windowedKey.window().start() /1000/60/15 *1000*60*15 + 1000*60*15
                            // the above rounds time down to a timestamp divisible by 15 minutes
                                )
                        ),
                        value
                ),
                /* your key serde */,
                /* your value serde */
        )
    .reduce(/*your adder*/, /*your subtractor*/, "store15m");

KTable<Windowed<Key>, Value> sixtyMinuteWindowed =

        fifteeenMinuteWindowed

    .groupBy( (windowedKey, value) ->
                new KeyValue<>(
                        new Windowed<>(
                                windowedKey.key(),
                                new Window<>(
                                        windowedKey.window().start() /1000/60/60 *1000*60*60,
                                        windowedKey.window().start() /1000/60/60 *1000*60*60 + 1000*60*60
                                        // the above rounds time down to a timestamp divisible by 60 minutes
                                )
                        ),
                        value
                ),
                /* your key serde */,
                /* your value serde */
        )

        .reduce(/*your adder*/, /*your subtractor*/, "store60m");
```

TODO: to mitigate infinite state store growth (until re-balance rebuilds it from the changelog) you can implement the Windowed key serde to store the timestamp(s) before the actual record key and periodically do a ranged query on each of the state stores to find and delete all data older than *x* (using punctuate() inside a Processor). TBC...

How to aggregate data from all currently active sessions?

Status: *Draft*
Intent:

- transaction sums per customer session (simple using session-windowed aggregation)
- global transaction sums for all ***currently active*** customer sessions

```
builder
  .stream(/*key serde*/, /*transaction serde*/, "transaciton-topic")

  .groupByKey(/*key serde*/, /*transaction serde*/)

  .aggregate(
    () -> /*empty aggregate*/,
    aggregator(),
    merger(),
    SessionWindows.with(SESSION_TIMEOUT_MS).until(SESSION_TIMEOUT_MS*2),
    /* aggregate serde */,
    txPerCustomerSumStore() // this store can be queried for per customer session data  )

  .toStream()

  .filter(((key, value) -> value != null)) // tombstones only come when a session is merged into a bigger
session, so ignore them

 // the below map/groupByKey/reduce operations are to only propagate updates to the latest session per customer
to downstream

  .map((windowedCustomerId, agg) -> // this moves timestamp from the windowed key into the value
                                    // so that we can group by customerId only and reduce to the later value
    new KeyValue<>(
      windowedCustomerId.key(), // just customerId
      new WindowedAggsImpl( // this is just like a tuple2 but with nicely named accessors: timestamp() and
aggs()
        windowedCustomerId.window().end(),
        agg
      )
    )
  )
  .groupByKey( /*key serde*/, /*windowed aggs serde*/ ) // key is just customerId
  .reduce( // take later session value and ignore any older - downstream only cares about current sessions
    (val, agg) -> val.timestamp() > agg.timestamp() ? val : agg,
    TimeWindows.of(SESSION_TIMEOUT_MS).advanceBy(SESSION_TIMOUT_DELAY_TOLERANCE_MS),
    "latest-session-windowed"
  )

  .groupBy((windowedCustomerId, timeAndAggs) -> // calculate totals with maximum granularity, which is per-
partition
    new KeyValue<>(
      new Windowed<>(
        windowedCustomerId.key().hashCode() % PARTITION_COUNT_FOR_TOTALS,  // KIP-159 would come in handy here,
to access partition number instead
        windowedCustomerId.window() // will use this in the interactive queries to pick the oldest not-yet-
expired window
      ),
      timeAndAggs.aggs()
    ),
    new SessionKeySerde<>(Serdes.Integer()),
    /* aggregate serde */
  )

  .reduce(
    (val, agg) -> agg.add(val),
    (val, agg) -> agg.subtract(val),
    txTotalsStore() // this store can be queried to get totals per partition for all active sessions
  );

builder.globalTable(
  new SessionKeySerde<>(Serdes.Integer()),
  /* aggregate serde */,
  changelogTopicForStore(TRANSACTION_TOTALS), "totals");
// this global table puts per partition totals on every node, so that they can be easily summed for global
totals, picking the oldest not-yet-expired window
```

```
TODO: put in StreamParitioners (with KTable.through variants added in KAFKA-5045) to avoid re-partitioning where
I know it's unnecessary.
```

The idea behind the % PARTITION_COUNT_FOR_TOTALS bit is that I want to first do summation with max parallelism and minimize the work needed downstream. So I calculate a per-partition sum first to limit the updates that the totals topic will receive and the summing work done by the interactive queries on the global store.

How to manage the size of state stores using tombstones?

An application that uses aggregations can make better use of it's resources by removing records it no longer needs from its state stores. Kafka Streams makes this possible through the usage of tombstone records, which are records that contain a non-null key, and a null value. When Kafka Streams sees a tombstone record, it deletes the corresponding key from the state store, thus freeing up space.

The usage of tombstone records will become apparent in the example below, but it's important to note that any record with a null key will be internally dropped, and will not be seen by your aggregation. Therefore, it is necessary that your aggregation include the logic for recognizing when a record can be dropped from the state store, and by returning null when this condition is met.

Consider the following example. An airline wants to track the various stages of a customer's flight. For this example, a customer can be in one of 4 stages: **booked, boarded, landed,** and **post-flight survey completed**. Once the customer has completed the post-flight survey, the airline no longer needs to track the customer. Until then, the airline would like to know what stage the customer is in, and perform various aggregations on the customer's data. This can be accomplished using the following topology.

```
// customer flight statuses
final String BOOKED = "booked";
final String BOARDED = "boarded";
final String LANDED = "landed";
final String COMPLETED_FLIGHT_SURVEY = "survey";

// topics
final String SOURCE_TOPIC = "someInputTopic";
final String STORE_NAME = "someStateStore";

// topology
KStreamBuilder builder = new KStreamBuilder();
KStream<String, String> stream = builder.stream(SOURCE_TOPIC);

KTable<String, String> customerFlightStatus = stream
    .groupByKey()
    .reduce(
        new Reducer<String>() {
            @Override
            public String apply(String value1, String value2) {
                if (value2.equals(COMPLETED_FLIGHT_SURVEY)) {
                    // we no longer need to keep track of this customer since
                    // they completed the flight survey. Create a tombstone
                    return null;
                }
                // keeping this simple for brevity
                return value2;
            }
       }, STORE_NAME);
```

Returning null in the **Reducer,** but only when the customer has completed their post-flight survey, allows us to perform aggregations *until* we no longer are interested in tracking this key. A tombstone is created by returning null, and the record is deleted from the state store immediately. We can verify this with the following tests:

```
// Simple test
final String customerFlightNumber = "customer123_dl90210";
File stateDir = createStateDir(); // implementation omitted
KStreamTestDriver driver = new KStreamTestDriver(builder, stateDir, Serdes.String(), Serdes.String());
driver.setTime(0L);

// get a reference to the state store
KeyValueStore<String, String> store = (KeyValueStore<String, String>) driver.context().getStateStore
(STORE_NAME);

// the customer has booked their flight
driver.process(SOURCE_TOPIC, customerFlightNumber, BOOKED);
store.flush();
assertEquals(BOOKED, store.get(customerFlightNumber));

// the customer has boarded their flight
driver.process(SOURCE_TOPIC, customerFlightNumber, BOARDED);
store.flush();
assertEquals(BOARDED, store.get(customerFlightNumber));

// the customer's flight has landed
driver.process(SOURCE_TOPIC, customerFlightNumber, LANDED);
store.flush();
assertEquals(LANDED, store.get(customerFlightNumber));

// the customer has filled out the post-flight survey, so we no longer need to track them
// in the state store. make sure the key was deleted
driver.process(SOURCE_TOPIC, customerFlightNumber, COMPLETED_FLIGHT_SURVEY);
store.flush();
assertEquals(null, store.get(customerFlightNumber));
```

Finally, it's important to note how tombstones are forwarded downstream. Whether or a not a tombstone is visible to additional sub-topologies depends on which abstraction (e.g. KTable or KStream) a sub-topology uses for streaming it's input data. The following code snippets highlight these differences. Tomb stones are visible in record streams, and it is common to filter them out before performing additional transformations (see below). However, in changelog streams, the tombstones are forwarded directly, and not visible when using operators like *filter, mapValues,* etc.

```
// tombstones are visible in a KStream and must be filtered out
KStream<String, String> subtopologyStream = customerFlightStatus
        .toStream()
        .filter((key, value) -> {
            if (value == null) {
                // tombstone! skip this
                return false;
            }
            // other filtering conditions...
            return true;
        });

// tombstone forwarding is different in KTables. The filter below is not evaluated for a tombstone
KTable<String, String> subtopologyKtable = customerFlightStatus
        .filter((key, value) -> {
            // the tombstone never makes it here. no need to check for null

                        // other filtering conditions...
            return true;
        });
```

How to purge data from KTables based on age

I'm not certain I would recommend this in general, but I've been asked to recommend a pattern for effectively implementing a TTL in a KTable. In principle, this could be done straightforwardly with a custom state store, but it raises questions about the integrity of the data provenance and the soundness of the application.

If the data in question is truly not needed anymore after 24 hours (or whatever other criteria), I think a better approach is to emit tombstones into the topic that populates the KTable in question. This circumvents a lot of tricky distributed systems questions.

Even better than this would be to purge the data from the source system and let those deletes naturally propagate into the topic and then into the KTable, but there are some special cases in which this isn't practical.

For example, I have seen one application that populates a keyed topic from a daily feed rather than a database's changelog. The feed only contains records that exist, records that have been deleted from the prior feed are simply not mentioned. Thus, there's no opportunity for the ingest to emit tombstones into the topic. One approach would be to effectively diff the current and prior feeds to identify records that have been deleted. But depending on the size and complexity of the feed, this might not be so simple.

In contrast, we can separate the concern of purging old data into a the following Streams application, intended to be run independently for each topic that needs purging. It simply watches the topic for records that have not been updated in a configured threshold of time and purges them from the topic by writing a tombstone back to it. Thus, the ingest job can just naively reflect the latest feed into the topic and all consumers can just consume the topic naively as well, and "forgotten" records will be purged from the topic by this job.

A refinement on this approach would be to use some identifying characteristic of the feed itself as a "generation" number and then tombstoning records that are not of the current generation, rather than using the record timestamp and age as the determining factor.

So, here you go: the comments should explain everything:

```
package io.confluent.example;

import org.apache.kafka.common.header.Headers;
import org.apache.kafka.common.serialization.Deserializer;
import org.apache.kafka.common.serialization.Serde;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.common.serialization.Serializer;
import org.apache.kafka.streams.KeyValue;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.TopologyTestDriver;
import org.apache.kafka.streams.kstream.Consumed;
import org.apache.kafka.streams.kstream.Produced;
import org.apache.kafka.streams.kstream.Transformer;
import org.apache.kafka.streams.kstream.TransformerSupplier;
import org.apache.kafka.streams.processor.ProcessorContext;
import org.apache.kafka.streams.processor.PunctuationType;
import org.apache.kafka.streams.state.KeyValueIterator;
import org.apache.kafka.streams.state.KeyValueStore;
import org.apache.kafka.streams.state.StoreBuilder;
import org.apache.kafka.streams.state.Stores;
import org.apache.kafka.streams.test.ConsumerRecordFactory;

import java.time.Duration;
import java.util.Map;
import java.util.Properties;

/**
 * This is one approach to purging old records from a data set.
 * Instead of implementing a TTL in a state store directly, it
 * consumes from an input topic and builds a table representing
 * the last non-tombstone update for each key.
 *
 * Twice a day, it scans the state store looking for records that are
 * over a day old. When it finds one, it emits a tombstone back to the
 * topic. These tombstones will be consumed later on, cleaning up that
 * record from this state store, as well as from any other application
 * building tables from the same topic.
 *
 * Thus, this application can be run as a co-process alongside other
 * Streams applications (and other Consumers in general), enforcing
 * a data retention policy, etc.
 */
public class PurgeApp {
    private static final String TOPIC = "input";
    private static final Duration SCAN_FREQUENCY = Duration.ofHours(12);
    private static final Duration MAX_AGE = Duration.ofDays(1);
```

```
    private static final String STATE_STORE_NAME = "purge-worker-store";

    /**
     * Not technically necessary, but I wanted to document that the
     * transformer only emits null values, so I've made the return type Void.
     * This means that we need a serde for it as well, which is trivial to implement,
     * since Void values can only be null.
     */
    public static class VoidSerde implements Serde<Void>, Serializer<Void>, Deserializer<Void> {

        @Override
        public Void deserialize(final String topic, final byte[] data) {
            if (data != null) {
                throw new IllegalArgumentException();
            } else {
                return null;
            }
        }

        @Override
        public Void deserialize(final String topic, final Headers headers, final byte[] data) {
            return deserialize(topic, data);
        }

        @Override
        public void configure(final Map<String, ?> configs, final boolean isKey) {}

        @Override
        public byte[] serialize(final String topic, final Void data) {
            if (data != null) {
                throw new IllegalArgumentException();
            } else {
                return null;
            }
        }

        @Override
        public byte[] serialize(final String topic, final Headers headers, final Void data) {
            return serialize(topic, data);
        }

        @Override
        public void close() {}

        @Override
        public Serializer<Void> serializer() {
            return this;
        }

        @Override
        public Deserializer<Void> deserializer() {
            return this;
        }
    }

    public static void main(String[] args) {
        final StreamsBuilder builder = new StreamsBuilder();

        // state store for maintaining the latest updated timestamp for the records
        final StoreBuilder<KeyValueStore<Long, Long>> storeBuilder = Stores.keyValueStoreBuilder(
            Stores.persistentKeyValueStore(STATE_STORE_NAME),
            Serdes.Long()/* keys are long valued in this example */,
            Serdes.Long()/* we store only the record timestamp, since it's all we're basing the purge on */
        );

        builder
            // first, we register the state store
            .addStateStore(storeBuilder)
            // then, we set up to consume from the topic (assuming the keys are long-valued and the values are
Strings
            .stream(TOPIC, Consumed.with(Serdes.Long(), Serdes.String()))
```

```
            // Then, we add our transformer,
            .transform(new TransformerSupplier<Long, String, KeyValue<Long, Void>>() {
                        @Override
                        public Transformer<Long, String, KeyValue<Long, Void>> get() {
                            return new Transformer<Long, String, KeyValue<Long, Void>>() {
                                private ProcessorContext context;
                                private KeyValueStore<Long, Long> stateStore;

                                @Override
                                public void init(final ProcessorContext context) {
                                    this.context = context;
                                    this.stateStore = (KeyValueStore<Long, Long>) context.getStateStore
(STATE_STORE_NAME);
                                    // This is where the magic happens. This causes Streams to invoke the
Punctuator
                                    // on an interval, using stream time. That is, time is only advanced by
the record timestamps
                                    // that Streams observes. This has several advantages over wall-clock
time for this application:
                                    // * It'll produce the exact same sequence of updates given the same
sequence of data.
                                    //   This seems nice, since the purpose is to modify the data stream
itself, you want to have
                                    //   a clear understanding of when stuff is going to get deleted. For
example, if something
                                    //   breaks down upstream for this topic, and it stops getting new data
for a while, wall clock
                                    //   time would just keep deleting data on schedule, whereas stream time
will wait for
                                    //   new updates to come in.
                                    context.schedule(
                                        SCAN_FREQUENCY,
                                        PunctuationType.STREAM_TIME,
                                        timestamp -> {
                                            final long cutoff = timestamp - MAX_AGE.toMillis();

                                            // scan over all the keys in this partition's store
                                            // this can be optimized, but just keeping it simple.
                                            // this might take a while, so the Streams timeouts should take
this into account
                                            try (final KeyValueIterator<Long, Long> all = stateStore.all()) {
                                                while (all.hasNext()) {
                                                    final KeyValue<Long, Long> record = all.next();
                                                    if (record.value != null && record.value < cutoff) {
                                                        // if a record's last update was older than our
cutoff, emit a tombstone.
                                                        context.forward(record.key, null);
                                                    }
                                                }
                                            }
                                        }
                                    );
                                }

                                @Override
                                public KeyValue<Long, Void> transform(final Long key, final String value) {
                                    // this gets invoked for each new record we consume. If it's a
tombstone, delete
                                    // it from our state store. Otherwise, store the record timestamp.
                                    if (value == null) {
                                        stateStore.delete(key);
                                    } else {
                                        stateStore.put(key, context.timestamp());
                                    }
                                    return null; // no need to return anything here. the punctuator will
emit the tombstones when necessary
                                }

                                @Override
                                public void close() {} // no need to close anything; Streams already closes
the state store.
```

```
                            };
                        }
                    },
                    STATE_STORE_NAME // register that this Transformer needs to be connected to our state
store.
            )
            // emit our tombstones back to the same topic.
            .to(TOPIC, Produced.with(Serdes.Long(), new VoidSerde()));


        // Just keeping it simple and testing inline:

        // build the topopology
        final Topology build = builder.build();
        System.out.println(
            build.describe().toString()
        );

        // testing a simple scenario with TopologyTestDriver:
        final Properties config = new Properties();
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "my-app");
        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "dummy-host");
        // you'll also want to pay close attention to how long that punctuation takes to run, and set the
appropriate timeouts accordingly
        // alternatively, since the punctuation doesn't modify the state store, you could put the scan in a
separate thread.
        final TopologyTestDriver topologyTestDriver = new TopologyTestDriver(build, config);

        final ConsumerRecordFactory<Long, String> recordFactory = new ConsumerRecordFactory<>(Serdes.Long().
serializer(), Serdes.String().serializer());

        // send one record at timestamp 0
        topologyTestDriver.pipeInput(recordFactory.create(TOPIC, 1L, "first", 0L));
        // prints "null" because the app doesn't emit anything
        System.out.println(topologyTestDriver.readOutput(TOPIC, Serdes.Long().deserializer(), Serdes.String().
deserializer()));

        // send another record two days later. This should cause the first record to get purged, since it's now
2 days old.
        topologyTestDriver.pipeInput(recordFactory.create(TOPIC, 2L, "second", 1000L * 60 * 60 * 24 * 2));
        // prints ProducerRecord(topic=input, partition=null, headers=RecordHeaders(headers = [], isReadOnly =
false), key=1, value=null, timestamp=172800000)
        // because the punctuation has run, scanning over the state store, and determined that key=1 can be
purged, since it is now 2 days old
        System.out.println(topologyTestDriver.readOutput(TOPIC, Serdes.Long().deserializer(), Serdes.String().
deserializer()));
        // prints "null" because the app doesn't emit anything else
        System.out.println(topologyTestDriver.readOutput(TOPIC, Serdes.Long().deserializer(), Serdes.String().
deserializer()));
    }
}
```