Kafka Streams

Kafka is an event streaming platform. Kafka Streams is a Java library for developing stream processing applications on top of Apache Kafka.

An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:

- Event key: "Alice"
- Event value: "Made a payment of $200 to Bob"
- Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

KStream

A KStream is an abstraction of **a record stream**, where each data record represents a self-contained datum in the unbounded data set. Using the table analogy, data records in a record stream are always interpreted as an "INSERT" -- think: adding more entries to an append-only ledger -- because no record replaces an existing row with the same key. Examples are a credit card transaction, a page view event, or a server log entry. To illustrate, let's imagine the following two data records are being sent to the stream:

("alice", 1) --> ("alice", 3)

If your stream processing application were to sum the values per user, it would return 4 for alice. Why? Because the second data record would not be considered an update of the previous record.

KTable

A KTable is an abstraction of **a changelog stream**, where each data record represents an update. More precisely, the value in a data record is interpreted as an "UPDATE" of the last value for the same record key, if any (if a corresponding key doesn't exist yet, the update will be considered an INSERT). Using the table analogy, a data record in a changelog stream is interpreted as an UPSERT aka INSERT/UPDATE because any existing row with the same key is overwritten. Also, null values are interpreted in a special way: a record with a null value represents a "DELETE" or tombstone for the record's key.To illustrate, let's imagine the following two data records are being sent to the stream:

("alice", 1) --> ("alice", 3)

If your stream processing application were to sum the values per user, it would return 3 for alice. Why? Because the second data record would be considered an update of the previous record. KTable also provides an ability to look up current values of data records by keys.

Stateless and stateful operations

There are stateless operations, such as `filter()`, `map()`, and `flatMap()`, which do not keep data around (do not maintain state) while moving from processing from one stream element to the next. And there are stateful operations, such as `distinct()`, `limit()`, `sorted()`, `reduce()`, and `collect()`, which may pass the state from previously processed elements to the processing of the next element. Stateless operations usually do not pose a problem when switching from a sequential stream to a parallel one. Each element is processed independently and the stream can be broken into any number of sub-streams for independent processing. [oreilly]

Stateless operation

1. filter

Use filter to omit or include records based on a criterion. For example, if the value sent to a topic contains a word and you want to include the ones which are greater than a specified length.

```
KStream<String, String> stream = builder.stream("words");

stream.filter(new Predicate<String, String>() {

        @Override

        public boolean test(String k, String v) {
                return v.length() > 5;

          }

})
```

2. filterNot

```
KStream<String, String> stream = builder.stream("words");

stream.filterNot((key,value) -> value.startsWith("foo"));
```

3. map

It can be used to transform each record in the input KStream by applying a mapper function. This is available in multiple flavors - map, mapValues, flatMap, flatMapValues.

Simply use the map method if you want to alter both key and the value. For e.g., to convert key and value to uppercase

```
stream.map(new KeyValueMapper<String, String, KeyValue<String, String>>() {
  @Override
  public KeyValue<String, String> apply(String k, String v) {
        return new KeyValue<>(k.toUpperCase(), v.toUpperCase());
    }
  });
```

Use mapValues if all you want to alter is the value: stream.mapValues(value -> value.toUpperCase());

4. **flatMap** similar to map, but it allows you to return multiple records (Key Values)

```
stream.flatMap(new KeyValueMapper<String, String, Iterable<? extends KeyValue<? extends String, ? extends String>>>() {
   @Override
   public Iterable<? extends KeyValue<? extends String, ? extends String>> apply(String k, String csv) {
       String[] values = csv.split(",");
       return Arrays.asList(values)
                .stream()
                .map(value -> new KeyValue<>(k, value))
                .collect(Collectors.toList());
         }
   })
```

In the above example, each record in the stream gets flatMap such that each CSV (comma separated) value is first split into its constituents and a KeyValue pair is created for each part of the CSV string. For e.g. if you have these records (foo <-> a,b,c) and (bar <-> d,e) (where foo and bar are keys), the resulting stream will have five entries - (foo,a), (foo,b), (foo,c), (bar,d), (bar,e)

Use flatMapValues if you only want to accept a value from the stream and return a collection of values

Terminal operations

A terminal operation in Kafka Streams is a method that returns void instead of an intermediate such as another KStream or KTable. You can use the to method to store the records of a KStream to a topic in Kafka.

```
KStream<String, String> stream = builder.stream("words");

stream.mapValues(value -> value.toUpperCase()).to("uppercase-words");
```

Stateful operations

The aggregation operation is applied to records of the same key. Kafka Streams supports the following aggregations: **aggregate**, **count**, and **reduce**. **Grouping** is a prerequisite for aggregation. You can run groupBy (or its variations) on a KStream or a KTable, which results in a KGroupedStream and KGroupedTable, respectively. This can be used for scenarios such as moving average, sum, count, etc.

1. **aggregate**

   Here is an example of how you can calculate the count (i.e. the number of times a specific key was received)

   ```
   StreamsBuilder builder = new StreamsBuilder();

   KStream<String, String> stream = builder.stream(INPUT_TOPIC);

   KTable<String, Count> aggregate =

       stream.groupByKey()

           .aggregate(new Initializer<Count>() {

                 @Override

                 public Count apply() {

                   return new Count("", 0);

                 }

               }, new Aggregator<String, String, Count>() {

                 @Override

                 public Count apply(String k, String v, Count aggKeyCount) {

                   Integer currentCount = aggKeyCount.getCount();
   ```

```
                    return new Count(k, currentCount + 1);
                  }
                });

        aggregate.toStream()
            .map((k,v) -> new KeyValue<>(k, v.getCount()))
            .to(COUNTS_TOPIC, Produced.with(Serdes.String(), Serdes.Integer()));
```

2. group

If you want to perform stateful aggregations on the contents of a KStream, you will first need to group its records by their key to create a **KGroupedStream**. Here is an example of how you can do this using **groupByKey**

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> stream = builder.stream(INPUT_TOPIC);
KGroupedStream<String,String> kgs = stream.groupByKey();
stream.groupBy(new KeyValueMapper<String, String, String>() {
  @Override
  public String apply(String k, String v) {
    return k.toUpperCase();
  }
});
```

A generalized version of groupByKey is groupBy which gives you the ability to group based on a different key using a KeyValueMapper.

3. count

count is such a commonly used form of aggregation that it is offered as a first-class operation. Once you have the stream records grouped by key (KGroupedStream), you can count the number of records of a specific key by using this operation.

```
StreamsBuilder builder = new StreamsBuilder();

KStream<String, String> stream = builder.stream(INPUT_TOPIC);

stream.groupByKey().count();
```

4. reduce

You can use reduce to combine the stream of values. The aggregate operation that was covered earlier is a generalized form of reduce. You can implement functionality such as sum, min, max, etc. Here is an example of max:

```
StreamsBuilder builder = new StreamsBuilder();
KStream<String, Long> stream = builder.stream(INPUT_TOPIC,
Consumed.with(Serdes.String(), Serdes.Long()));
stream.groupByKey()
    .reduce(new Reducer<Long>() {
            @Override
             public Long apply(Long currentMax, Long v) {
               Long max = (currentMax > v) ? currentMax : v;
                  return max;
               }
            })
    .toStream().to(OUTPUT_TOPIC);
return builder.build();
```

Windowing

Stateful Kafka Streams operations also support Windowing. This allows you to scope your stream processing pipelines to a specific time window/range (e.g. track the number of link clicks per minute or unique page views per hour).
To perform Windowed aggregations on a group of records, you will have to create a using groupBy on a KStream and then using the windowedBy operation. You can choose between traditional windows (**tumbling**, **hopping**, or **sliding**) or **session-based** time windows. Best explain here:

https://kafka.apache.org/10/documentation/streams/developer-guide/dsl-api.html#sliding-time-windows

https://docs.microsoft.com/en-us/azure/stream-analytics/stream-analytics-window-functions

https://kafka.apache.org/10/javadoc/org/apache/kafka/streams/kstream/SessionWindows.html

Using windowedBy(Windows<W> windows) on a KGroupedStream returns a TimeWindowedKStream on top of which you can invoke the above-mentioned aggregate operations.

For example, if you want the number of clicks over a specific time range (say five minutes), choose a tumbling time window. This will ensure that the records are clearly segregated across the given time boundaries. In other words, clicks from user 1 from 10-10:05 a.m. will be aggregated (counted) separately and a new time block (window) starts from 10:06 a.m., during which the clicks counter is reset to zero and counted again.

Other window types include:

- Tumbling time windows, which never overlap. A record will only be part of one window.
- Hopping time windows where records can be present in one or more time ranges/windows.
- Sliding time windows are meant for use with Joining operations.
- Session windows: Dynamically-sized, non-overlapping, data-driven windows

Good reference

https://kafka.apache.org/27/documentation/streams/developer-guide/dsl-api.html

http://kafka.apache.org/intro#intro_producers