Stateless transformations allow you to process records individually, but what if you need some information about multiple records at the same time? Aggregations allow you to process groups of records that share the same key and to maintain the state of your processing in a state store managed in the Kafka cluster. In this lesson, we will discuss what aggregations are, and we will demonstrate how to use three different types of aggregations in a Java application.

Relevant Documentation

- Kafka Streams Developer Guide — Aggregating

Lesson Reference

1. Clone the starter project, if you haven't already done so in a previous lesson.

```
cd ~/
git clone https://github.com/linuxacademy/content-ccdak-kafka-streams.git
cd content-ccdak-kafka-streams
```

2. Edit the `AggregationsMain` class.

```
vi src/main/java/com/linuxacademy/ccdak/streams/AggregationsMain.java
```

3. Implement a Streams application that performs a variety of aggregations.

```
package com.linuxacademy.ccdak.streams;

import java.util.Properties;
import java.util.concurrent.CountDownLatch;
import org.apache.kafka.common.serialization.Serdes;
import org.apache.kafka.streams.KafkaStreams;
import org.apache.kafka.streams.StreamsBuilder;
import org.apache.kafka.streams.StreamsConfig;
import org.apache.kafka.streams.Topology;
import org.apache.kafka.streams.kstream.KGroupedStream;
import org.apache.kafka.streams.kstream.KStream;
import org.apache.kafka.streams.kstream.KTable;
import org.apache.kafka.streams.kstream.Materialized;
import org.apache.kafka.streams.kstream.Produced;

public class AggregationsMain {<pre><code>public static void main(String[] args) {
    // Set up the configuration.
    final Properties props = new Properties();
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "aggregations-example");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
    props.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG, 0);
    // Since the input topic uses Strings for both key and value, set the default Serdes to String.
    props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
    props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());

    // Get the source stream.
    final StreamsBuilder builder = new StreamsBuilder();
    KStream&lt;String, String&gt; source = builder.stream("aggregations-input-topic");

    // Group the source stream by the existing Key.
    KGroupedStream&lt;String, String&gt; groupedStream = source.groupByKey();

    // Create an aggregation that totals the length in characters of the value for all records sharing th
    KTable&lt;String, Integer&gt; aggregatedTable = groupedStream.aggregate(
        () -&gt; 0,
        (aggKey, newValue, aggValue) -&gt; aggValue + newValue.length(),
        Materialized.with(Serdes.String(), Serdes.Integer()));
    aggregatedTable.toStream().to("aggregations-output-charactercount-topic", Produced.with(Serdes.String
```

```
    // Count the number of records for each key.
    KTable&lt;String, Long&gt; countedTable = groupedStream.count(Materialized.with(Serdes.String(), Serd
    countedTable.toStream().to("aggregations-output-count-topic", Produced.with(Serdes.String(), Serdes.L

    // Combine the values of all records with the same key into a string separated by spaces.
    KTable&lt;String, String&gt; reducedTable = groupedStream.reduce((aggValue, newValue) -&gt; aggValue
    reducedTable.toStream().to("aggregations-output-reduce-topic");

    final Topology topology = builder.build();
    final KafkaStreams streams = new KafkaStreams(topology, props);
    // Print the topology to the console.
    System.out.println(topology.describe());
    final CountDownLatch latch = new CountDownLatch(1);

    // Attach a shutdown handler to catch control-c and terminate the application gracefully.
    Runtime.getRuntime().addShutdownHook(new Thread("streams-shutdown-hook") {
        @Override
        public void run() {
            streams.close();
            latch.countDown();
        }
    });

    try {
        streams.start();
        latch.await();
    } catch (final Throwable e) {
        System.out.println(e.getMessage());
        System.exit(1);
    }
    System.exit(0);
}</code></pre>}
```

4. In a separate session, start a `kafka-console-producer` to produce data to the input topic.

```
kafka-console-producer --broker-list localhost:9092 --topic aggregations-input-topic --property parse.key
```

5. Publish an initial record to automatically create the topic.

```
a:a
```

6. In the previous session, run your code.

```
./gradlew runAggregations
```

7. Open three more sessions. In each one, start a `kafka-console-consumer` to view records being published to the three output topics, then publish some records to the input topic and examine how your Streams application modifies them.

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic aggregations-output-charactercount-topic
```

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic aggregations-output-count-topic --proper
```

```
kafka-console-consumer --bootstrap-server localhost:9092 --topic aggregations-output-reduce-topic --prope
```