Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Systems@ETH Zürich

Spring Term 2019 Dr. Vasiliki Kalavri

Data Stream Processing and Analytics

Assignment 4

In this assignment you will learn:
- How to install and setup Apache Kafka on your computer
- How to read and write to a Kafka topic
- How to use Kafka as a source and sink in Apache Flink

Apache Kafka is a distributed and fault-tolerant publish-subscribe messaging system and serves as the ingestion, storage, and messaging layer for large production streaming pipelines. Kafka is commonly deployed on a cluster of one or more servers and relies on Apache Zookeeper for reliable distributed coordination.

Let's start with basic Kafka terminology:

- A *topic* identifies a category of stream records stored in a Kafka cluster. Records consist of a key, a value, and a timestamp.
- A *producer* publishes a stream of records to a Kafka topic and a *consumer* subscribes to one or more topics and processes the stream of records published in them.
- Topics are multi-subscriber, i.e. a topic can have zero, one, or many consumers that subscribe to the data written to it. For each topic, the Kafka cluster maintains a *partitioned log*. Each *partition* is an ordered, immutable sequence of records that is continually appended to—a structured commit log.
- An *offset* is a sequential id number assigned to records within a partition. It uniquely identifies records within each partition.
- The *retention policy* defines a time period after a record is published that it is available for consumption. Records are discarded after their retention time to free up disk space.

1. Setup Apache Kafka and run an example

Follow the Apache Kafka Quickstart steps 1-6. You can skip steps 7-8 as we will be using Kafka together with Flink instead.

Tasks

#1: Refer to the Apache Kafka documentation and describe what the following parameters control:

- `acks`
- `log.flush.interval.*`
- `log.retention.*`
- `retries`

#2: What effect do you think the above parameters can have on (i) performance, and (ii) result correctness under potential failures?

2. Write a stream of records to Kafka using Flink

Manually sending messages to Kafka and reading them back with the console consumer is not much fun. Flink has a Kafka connector and provides implementations for both a producer and a consumer. Use your Apache Flink wiki-edits project from Assignment #1 as a starting point and add the following dependency to your `pom.xml` to access the Kafka connector:

```
<dependency>
    <groupId>org.apache.flink</groupId>
    <artifactId>flink-connector-kafka-0.11_2.11</artifactId>
    <version>${flink.version}</version>
</dependency>
```

Make sure to replace `${flink.version}` with your Flink version.

The following code show how to use `FlinkKafkaProducer011` as a sink in a Flink program:

```
DataStream<String> stream = ...;

FlinkKafkaProducer011<String> myProducer = new FlinkKafkaProducer011<String>(
        "localhost:9092",            // broker list
        "my-topic",                  // target topic
        new SimpleStringSchema());   // serialization schema

stream.addSink(myProducer);
```

Tasks

#1: Use the `WikipediaEditsSource` of Assignment #1 and write a program that ingests wikipedia edits and writes them to a Kafka topic. Create the Kafka topic, launch your Flink job, and inspect the topic contents with the console consumer.

#2: If not specified, the default partitioner maps each sink task to a single Kafka partition, i.e., all records that are emitted by the same sink task are written to the same partition. Try the default partitioner and vary the (i) degree of parallelism for Flink sink tasks, and (ii) the number of partitions in the Kafka topic. What happens when:

- there are more sink tasks than partitions?
- there are more partitions than sink tasks?

#3: Implement a `FlinkKafkaPartitioner` that works in a round-robin fashion and evenly distributes events among available partitions.

3. Read back from Kafka with Flink

We will now use Flink's Kafka consumer, `FlinkKafkaConsumer011`, as a source. The Kafka source provides access to one or more Kafka topics and receives the following configuration parameters:

1. The topic name / list of topic names
2. A `DeserializationSchema` / `KeyedDeserializationSchema` for deserializing the data from Kafka
3. Properties for the Kafka consumer:
    - "bootstrap.servers": comma separated list of Kafka brokers
    - "zookeeper.connect": comma separated list of Zookeeper servers
    - "group.id": the id of the consumer group

The following code sets up a Kafka source with all required properties and additionally instructs it to read the topic from the beginning:

```
Properties kafkaProps = new Properties();
kafkaProps.setProperty("zookeeper.connect", LOCAL_ZOOKEEPER_HOST);
kafkaProps.setProperty("bootstrap.servers", LOCAL_KAFKA_BROKER);
kafkaProps.setProperty("group.id", YOUR_GROUP);
// always read the Kafka topic from the start
kafkaProps.setProperty("auto.offset.reset", "earliest");

DataStream<String> stream = env
      .addSource(new FlinkKafkaConsumer011<>("topic",
                  new SimpleStringSchema(), properties));
```

Tasks

#1: Change the `WikipediaAnalysis` program of Assignment #1 to use a Flink Kafka consumer as the source. You will need to implement your own `DeserializationSchema` to let the consumer know how to turn the binary data in Kafka into `WikipediaEditEvent` events. Refer to the relevant Flink documentation for details on the required methods.