263-3826-00 Data Stream Processing and Analytics **16 Mar 2019**
Assignment 4 - Kafka
Name: Zhifei Yang Legi: 17-941-998

1 Setup Apache Kafka and run an example

1.1 Parameters

1. `acks`: A parameter in producer configs. It specifies how many acknowledgments the producer requires the leader to have received before considering a request complete. This controls the durability of records that are sent.

   If it is set to 0, the producer will not wait for server acknowledgements at all, so there is no guarantee that the server has received the record, and no retries will be performed as a result.

   If it is set to 1, the leader writes to its local log and acknowledges the record without waiting for acknowledgements from followers.

   If it is set to `all` or -1, the leader will wait for the acknowledgements from all in-sync replicas.

2. `log.flush.interval.*`: A group of parameters in broker configs, controlling when messages should be flushed to disk: `messages` specifies the number of messages accumulated on a log partition before they are flushed, and `ms` specifies how many milliseconds a message in any topic is kept in memory before flushed to disk.

3. `log.retention.*`: A group of parameters in broker configs that controls the retention policy. `bytes` specifies the maximum bytes of the log before deleting it. `hours, minutes, ms` specifies the time to keep a log file before deleting it, and a finer-grained time unit (if set) will override the larger ones.

4. `retries`: A parameter in producer configs (also exists in Streams, AdminClient with the same semantics). Controls how many times at most the client should resend any record whose send fails with a potentially transient error.

1.2 Effects on performance and result correctness under potential failures

1. `acks`: If set to 0, there is no guarantee that the record will be successfully received at all, so the result can be wrong if the delivery has failed. But since there is no waiting for acknowledgement, the performance will be better than other cases.

   If set to 1, should the leader fail immediately after acknowledging the record but before the followers have replicated it then the record will be lost. This gives stronger correctness guarantee while not sacrificing too much performance on waiting for every acknowledgement.

   If set to `all`, the record will not be lost as long as at least one in-sync replica remains alive. This is the strongest availability guarantee, but the performance is worst due to the potentially long waiting.

2. `log.flush.interval.*`: Writing to disk is a slow operation, but it is necessary for reliablity. If the flushing is too frequent, it will add more latency to client request processing

but consistency will be well guaranteed. If the flushing is too infrequent, when there are failures, the messages that are not flushed yet will be lost; in addition, if every time there is a large amount of data to be flushed, the system may get blocked by I/O.

3. `log.retention.*`: The larger it is, the longer the logs will be reserved, and it will be more likely for the system to recover under failures. However, longer retention implies occupying more space, which affects the performance.

4. `retries`: Allowing retries without setting max.in.flight.requests.per.connection to 1 will potentially change the ordering of records because if two batches are sent to a single partition, and the first fails and is retried but the second succeeds, then the records in the second batch may appear first.

   Setting `retries` to a higher value makes the sending more likely to succeed (eventually), so the result is more likely to be correct under potential failures, but the performance will be degraded when there are too many retries.

2 Write a stream of records to Kafka using Flink

The code for this section can be found in `FlinkKafkaProducer.java` in the appendix.

2.1 Task 1

It can be done in three steps:

- First, convert the `WikipediaEditEvent` stream into a `String` stream.
- Second, create a `FlinkKafkaProducer011<String>` instance.
- Finally, add the producer as a sink of the stream.

2.2 Task 2

Looking at the code of the default partitioner `FlinkKafkaPartitioner`, the partitioning strategy is to distribute the message to a fixed partition numbered by `(this.parallelInstanceId % partitions.length)`. So when there are more sink tasks than partitions, the first few partitions will receive more messages than others; when there are more partitions than sink tasks, partitions numbered beyond the number of sink tasks will not receive any data.

Setting the degree of parallelism for Flink sink tasks with `flink run -p {parallelism}`, and setting the number of partitions with `kafka-topics.sh --partitions {partitions}`, then running the Flink job and inspecting the number of messages in each partition using the following command:

```
$ bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092
  --topic wiki-edits
```

The following phenomenons are observed and match the strategy concluded from the source code:

- When Flink sink tasks parallelism is 2 and number of partitions in the Kafka `wiki-edits` topic is 4: only partitions 0 and 1 get data.
- When Flink sink tasks parallelism is 3 and number of partitions in the Kafka `wiki-edits` topic is 2: both partitions get data, but partition 0 gets nearly double the number of messages of partition 1.

2.3 Task 3

To use a custom partitioner, another constructor method of `FlinkKafkaProducer011` that takes the partitioner as a parameter has to be used. It also asks for a `Property` instead of the `brokerList`, so we borrow the `getPropertiesFromBrokerList` method from `FlinkKafkaProducerBase` to do the conversion.

The implemented custom partitioner does round-robin distributing by updating a private counter every time `partition()` is called: `this.next = (this.next + 1) % num_partitions`, and returns the partition indexed by `this.next`.

Please see the submitted source code for details.

3 Read back from Kafka with Flink

Please refer to the code in `FlinkKafkaConsumer.java` in the appendix. This is basically done in two steps:

1. Setup the Kafka source with the code provided in the assignment sheet, and fill in the correct parameters.

2. Implement the `CustomDeserializationSchema` which extends `AbstractDeserializationSchema<WikipediaEditEvent>`.

   Since the received message is a string, which represents a `WikipediaEditEvent` object serialized by `WikipediaEditEvent::toString`, we need to deserialize it back into a `WikipediaEditEvent` object by parsing the string. This is done using a regular expression to match the correpsonding fields. The boolean flags are encoded in a integer field `flags`, so we also need to decode it back to six booleans when re-constructing the object.

Code

FlinkKafkaProducer.java

```
package wikiedits;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.functions.ReduceFunction;
import org.apache.flink.api.common.serialization.SimpleStringSchema;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import
org.apache.flink.streaming.api.windowing.assigners.TumblingEventTimeWindow
s;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducer011;
import
org.apache.flink.streaming.connectors.kafka.partitioner.FlinkKafkaPartitio
ner;
import org.apache.flink.streaming.connectors.wikiedits.WikipediaEditEvent;
import
org.apache.flink.streaming.connectors.wikiedits.WikipediaEditsSource;
import org.apache.flink.util.Preconditions;

import java.util.Optional;

import static
org.apache.flink.streaming.connectors.kafka.FlinkKafkaProducerBase.getProp
ertiesFromBrokerList;

public class FlinkKafkaProducer {

    public static void main(String[] args) throws Exception {
        // set up the streaming execution environment
        final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();
        DataStream<WikipediaEditEvent> edits = env.addSource(new
WikipediaEditsSource());

        DataStream<String> stream =
edits.map(WikipediaEditEvent::toString);
        FlinkKafkaProducer011<String> myProducer = new
FlinkKafkaProducer011<String>(
                "wiki-edits", // target topic
                new SimpleStringSchema(),  // serialization schema
                getPropertiesFromBrokerList("localhost:9092"), // broker
list
```

```
                Optional.of(new CustomPartitioner<>()));  // round robin
partitioner
        stream.addSink(myProducer);

        DataStream<Tuple2<String, Integer>> result = edits
                // project the event user and the diff
                .map(new MapFunction<WikipediaEditEvent, Tuple2<String,
Integer>>() {
                    @Override
                    public Tuple2<String, Integer> map(WikipediaEditEvent
event) {
                        return new Tuple2<>(
                                event.getUser(), event.getByteDiff());
                    }
                })
                // group by user
                .keyBy(0)
                // aggregate changes per user
                .reduce(new ReduceFunction<Tuple2<String, Integer>>() {
                    @Override
                    public Tuple2<String, Integer> reduce(Tuple2<String,
Integer> e1, Tuple2<String, Integer> e2) {
                        return new Tuple2<>(e1.f0, e1.f1 + e2.f1);
                    }
                })
                // filter out negative byte changes
                .filter(new FilterFunction<Tuple2<String, Integer>>() {
                    @Override
                    public boolean filter(Tuple2<String, Integer> e)
throws Exception {
                        return e.f1 >= 0;
                    }
                });

        // execute program
        env.execute("Flink Streaming Java API Skeleton");
    }

    static class CustomPartitioner<T> extends FlinkKafkaPartitioner<T> {
        private int next = 0;

        @Override
        public int partition(T record, byte[] key, byte[] value, String
targetTopic, int[] partitions) {
            Preconditions.checkArgument(partitions != null &&
partitions.length > 0, "Partitions of the target topic is empty.");
            this.next = (this.next + 1) % partitions.length;
            return partitions[this.next];
        }
    }
}
```

FlinkKafkaConsumer.java

```
package wikiedits;

import java.util.Properties;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import org.apache.flink.api.common.functions.FilterFunction;
import org.apache.flink.api.common.functions.MapFunction;
import org.apache.flink.api.common.functions.ReduceFunction;
import
org.apache.flink.api.common.serialization.AbstractDeserializationSchema;
import org.apache.flink.api.java.tuple.Tuple2;
import org.apache.flink.streaming.api.datastream.DataStream;
import
org.apache.flink.streaming.api.environment.StreamExecutionEnvironment;
import org.apache.flink.streaming.api.windowing.time.Time;
import org.apache.flink.streaming.connectors.kafka.FlinkKafkaConsumer011;
import org.apache.flink.streaming.connectors.wikiedits.WikipediaEditEvent;

public class FlinkKafkaConsumer {

    public static void main(String[] args) throws Exception {
        // set up the streaming execution environment
        final StreamExecutionEnvironment env =
StreamExecutionEnvironment.getExecutionEnvironment();

        Properties kafkaProps = new Properties();
        kafkaProps.setProperty("zookeeper.connect", "localhost:2181");
        kafkaProps.setProperty("bootstrap.servers", "localhost:9092");
        kafkaProps.setProperty("group.id", "test-consumer-group");
        // always read the Kafka topic from the start
        kafkaProps.setProperty("auto.offset.reset", "earliest");
        DataStream<WikipediaEditEvent> edits = env
                .addSource(new FlinkKafkaConsumer011<>("wiki-edits",
                        new CustomDeserializationSchema(), kafkaProps));

        DataStream<Tuple2<String, Integer>> result = edits
                // project the event user and the diff
                .map(new MapFunction<WikipediaEditEvent, Tuple2<String,
Integer>>() {
                    @Override
                    public Tuple2<String, Integer> map(WikipediaEditEvent
event) {
                        return new Tuple2<>(
                                event.getUser(), event.getByteDiff());
                    }
                })
                // group by user
                .keyBy(0)
```

```
                // tumbling event-time windows
                .timeWindow(Time.seconds(10))
                // aggregate changes per user
                .reduce(new ReduceFunction<Tuple2<String, Integer>>() {
                    @Override
                    public Tuple2<String, Integer> reduce(Tuple2<String,
Integer> e1, Tuple2<String, Integer> e2) {
                        return new Tuple2<>(e1.f0, e1.f1 + e2.f1);
                    }
                })
                // filter out negative byte changes
                .filter(new FilterFunction<Tuple2<String, Integer>>() {
                    @Override
                    public boolean filter(Tuple2<String, Integer> e)
throws Exception {
                        return e.f1 >= 0;
                    }
                });
        result.print();

        // execute program
        env.execute("Flink Streaming Java API Skeleton");
    }

    static class CustomDeserializationSchema extends
AbstractDeserializationSchema<WikipediaEditEvent> {
        @Override

        public WikipediaEditEvent deserialize(byte[] message) {
            // format:
            // "WikipediaEditEvent{timestamp=" + this.timestamp + ",
channel='" + this.channel + '\'' + ", title='" + this.title + '\'' + ",
diffUrl='" + this.diffUrl + '\'' + ", user='" + this.user + '\'' + ",
byteDiff=" + this.byteDiff + ", summary='" + this.summary + '\'' + ",
flags=" + this.flags + '}';
            // example:
            // WikipediaEditEvent{timestamp=1552870599316,
channel='#en.wikipedia', title='Talk:Shooting of Trayvon Martin',
diffUrl='https://en.wikipedi=888128575', user='Psalm84', byteDiff=-2,
summary='/* Proposed split */ minor word changes', flags=33}

            String s = new String(message);
            Pattern p = Pattern.compile("WikipediaEditEvent\\{timestamp=
(.*), channel='(.*)', title='(.*)', diffUrl='(.*)', user='(.*)', byteDiff=
(.*), summary='(.*)', flags=(.*)}");
            Matcher m = p.matcher(s);
            if (m.find()) {
                int flags = Integer.valueOf(m.group(8));
                boolean isMinor = (flags & 1) != 0;
                boolean isNew = (flags & 2) != 0;
                boolean isUnpatrolled = (flags & 4) != 0;
                boolean isBotEdit = (flags & 8) != 0;
                boolean isSpecial = (flags & 16) != 0;
                boolean isTalk = (flags & 32) != 0;
```

```
                return new WikipediaEditEvent(Long.valueOf(m.group(1)),
m.group(2), m.group(3), m.group(4), m.group(5),
Integer.valueOf(m.group(6)), m.group(7), isMinor, isNew, isUnpatrolled,
isBotEdit, isSpecial, isTalk);
            } else {
                throw new RuntimeException("error in parsing!");
            }
        }
    }
}
```