Lab 5.5: Kafka Batching Records

Welcome to the session 5 lab 5. The work for this lab is done in `~/kafka-training/labs/lab5.5`. In this lab, you are going to try Kafka producer batching and compression.

Note: Do not run scripts inside `bin` directory. Run scripts from `~/kafka-training/labs/lab5.5/solution` directory

Lab Batching and Compressing Kafka Records

Objectives is to understand Kafka batching. You will disable batching and observer metrics, then you will reenable batching and observe metrics. Next you increase batch size and linger and observe metrics. In this lab, we will run a consumer to see batch sizes change from a consumer perspective as we change batching on the producer side. Lastly you will enable compression, and then observe results.

SimpleStockPriceConsumer

We added a `SimpleStockPriceConsumer` to consume `StockPrices` and display batch lengths for poll(). We won't cover the consumer in detail just quickly, since this is a `Producer` lab not a `Consumer` lab. You will run this consumer while you are running the `StockPriceKafkaProducer`. While you are running SimpleStockPriceConsumer with various batch and linger config, observe output of Producer metrics and StockPriceKafkaProducer output.

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Producer: SimpleStockPriceConsumer to consumer records

```java
package com.fenago.kafka.consumer;
import com.fenago.kafka.StockAppConstants;
import com.fenago.kafka.producer.model.StockPrice;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.StringDeserializer;
...

public class SimpleStockPriceConsumer {

    private static Consumer<String, StockPrice> createConsumer() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG,
                StockAppConstants.BOOTSTRAP_SERVERS);
        props.put(ConsumerConfig.GROUP_ID_CONFIG,
                "KafkaExampleConsumer");
        props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
                StringDeserializer.class.getName());
        //Custom Deserializer
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StockDeserializer.class.getName());
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        // Create the consumer using props.
        final Consumer<String, StockPrice> consumer =
                new KafkaConsumer<>(props);
        // Subscribe to the topic.
        consumer.subscribe(Collections.singletonList(
                StockAppConstants.TOPIC));
        return consumer;
    }
    ...
}
```

The `SimpleStockPriceConsumer` is similar to other `Consumer` examples we have covered so far. `SimpleStockPriceConsumer` subscribes to `stock-prices` topic and uses a custom serializer (`StockDeserializer`).

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Producer: SimpleStockPriceConsumer runConsumer

```
package com.fenago.kafka.consumer;
...

public class SimpleStockPriceConsumer {
    ...

    static void runConsumer() throws InterruptedException {
        final Consumer<String, StockPrice> consumer = createConsumer();
        final Map<String, StockPrice> map = new HashMap<>();
        try {
            final int giveUp = 1000; int noRecordsCount = 0;
            int readCount = 0;
            while (true) {
                final ConsumerRecords<String, StockPrice> consumerRecords =
                        consumer.poll(1000);
                if (consumerRecords.count() == 0) {
                    noRecordsCount++;
                    if (noRecordsCount > giveUp) break;
                    else continue;
                }
                readCount++;
                consumerRecords.forEach(record -> {
                    map.put(record.key(), record.value());
                });
                if (readCount % 100 == 0) {
                    displayRecordsStatsAndStocks(map, consumerRecords);
                }
                consumer.commitAsync();
            }
        }
        finally {
            consumer.close();
        }
        System.out.println("DONE");
    }
    ...
    public static void main(String... args) throws Exception {
      runConsumer();
    }
    ...
}
```

The run method drains Kafka topic. It creates map of current stocks prices, and calls `displayRecordsStatsAndStocks()`.

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Producer: SimpleStockPriceConsumer displayRecordsStatsAndStocks

```
package com.fenago.kafka.consumer;
...

public class SimpleStockPriceConsumer {
    ...
    private static void displayRecordsStatsAndStocks(
        final Map<String, StockPrice> stockPriceMap,
        final ConsumerRecords<String, StockPrice> consumerRecords) {
    System.out.printf("New ConsumerRecords par count %d count %d\n",
            consumerRecords.partitions().size(),
            consumerRecords.count());
    stockPriceMap.forEach((s, stockPrice) ->
            System.out.printf("ticker %s price %d.%d \n",
                stockPrice.getName(),
                stockPrice.getDollars(),
                stockPrice.getCents()));
    System.out.println();
  }

  ...
}
```

The `displayRecordsStatsAndStocks` method prints out size of each partition read and total record count. Then it prints out each stock at its current price.

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/consumer/StockDeserializer.java

Kafka Producer: StockDeserializer

```
package com.fenago.kafka.consumer;

import com.fenago.kafka.producer.model.StockPrice;
import org.apache.kafka.common.serialization.Deserializer;

import java.nio.charset.StandardCharsets;
import java.util.Map;

public class StockDeserializer implements Deserializer<StockPrice> {

    @Override
    public StockPrice deserialize(final String topic, final byte[] data) {
        return new StockPrice(new String(data, StandardCharsets.UTF_8));
    }

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

    @Override
    public void close() {
    }
}
```

The `StockDeserializer` is used to deserialize StockPrice objects from the Kafka topic.

Disable batching for the Producer

Let's start by disabling batching in the StockPriceKafkaProducer. Setting `props.put(ProducerConfig.BATCH_SIZE_CONFIG, 0)` turns batching off. After you do this rerun `StockPriceKafkaProducer` and check `Consumer` stats and `Producer` stats.

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer disable batching

```

public class StockPriceKafkaProducer {

    private static Producer<String, StockPrice>
                                    createProducer() {
        final Properties props = new Properties();
        setupBootstrapAndSerializers(props);
        setupBatchingAndCompression(props);

        return new KafkaProducer<>(props);
    }
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  0);
    }
    ...
}
```

ACTION - EDIT StockPriceKafkaProducer.java and disable batching as described above.

ACTION - RUN Zookeeper, Brokers, and StockPriceKafkaProducer, and look at the stats.

ACTION - RUN SimpleStockPriceConsumer and look at record sizes, look at Producer metrics too.

Set batching to 16K and retest

Now let's enable batching in the `StockPriceKafkaProducer` by setting batch size to 16K. Setting `props.put(ProducerConfig.BATCH_SIZE_CONFIG, 16_384)` turns batching on and allows us to batch 16K of stock price records per partition. After you do this rerun `StockPriceKafkaProducer` and check `Consumer` stats and `Producer` stats.

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer set batch size to 16K

```
public class StockPriceKafkaProducer {
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384);
    }
    ...
}
```

Results Set batching to 16K

We saw the consumer records per poll averages around 7.5 and saw the batch size increase to 136.02 - 59% more batching. Look how much the request queue time shrunk! The record-send-rate is 200% faster! You can see record-send-rate in the metrics of the producer.

ACTION - EDIT StockPriceKafkaProducer.java and set batch size to 16K.

ACTION - RUN StockPriceKafkaProducer, and look at the stats.

ACTION - RUN SimpleStockPriceConsumer and look at record sizes, look at Producer metrics too.

Set batching to 16K and linger to 10ms

Now let's enable linger in the `StockPriceKafkaProducer` by setting the linger 10 ms. Setting `props.put(ProducerConfig.LINGER_MS_CONFIG, 10)` turns linger on and allows us to batch for 10 ms or 16K bytes of a stock price records per partition whichever comes first. After you do this rerun `StockPriceKafkaProducer` and check `Consumer` stats and `Producer` stats.

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer set linger to 10 ms

```
public class StockPriceKafkaProducer {
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {

          //Linger up to 10 ms before sending batch if size not met
        props.put(ProducerConfig.LINGER_MS_CONFIG, 10);
        props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384);
    }
    ...
}
```

Results Set batching to 16K and linger to 10 ms

We saw the consumer records per poll averages around 17 and saw the batch size increase to 796 - 585% more batching. The record-send-rate went down, but higher than without batching.

ACTION - EDIT StockPriceKafkaProducer.java and set linger to 10ms.

ACTION - RUN StockPriceKafkaProducer, and look at the stats.

ACTION - RUN SimpleStockPriceConsumer and look at record sizes, look at Producer metrics too.

Try different sizes and times

Try 16K, 32K and 64K batch sizes and then try 10 ms, 100 ms, and 1 second linger. Which is the best for which type of use case?

ACTION - EDIT StockPriceKafkaProducer.java try different batch sizes and linger times.

ACTION - RUN StockPriceKafkaProducer, and look at the stats.

ACTION - RUN SimpleStockPriceConsumer and look at record sizes, look at Producer metrics too.

Set compression to snappy, then batching to 64K and linger to 50ms

Now let's enable compression in the `StockPriceKafkaProducer` by setting the compression to linger. Setting `props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy")` turns compression. After you do this rerun `StockPriceKafkaProducer` and check `Consumer` stats and `Producer` stats as before.

~/kafka-training/labs/lab5.5/src/main/java/com/fenago/kafka/producer/StockPriceKafkaProducer.java

Kafka Producer: StockPriceKafkaProducer enable compression

```java
public class StockPriceKafkaProducer {
    ...
    private static void setupBatchingAndCompression(
            final Properties props) {

              //Linger up to 50 ms before sending batch if size not met
              props.put(ProducerConfig.LINGER_MS_CONFIG, 50);

              //Batch up to 64K buffer sizes.
              props.put(ProducerConfig.BATCH_SIZE_CONFIG,  16_384 * 4);

              //Use Snappy compression for batch compression.
              props.put(ProducerConfig.COMPRESSION_TYPE_CONFIG, "snappy");
    }
    ...
}
```

Results for turning on compression

The Snappy compression 64K/50ms should have the highest record-send-rate and 1/2 the queue time.

ACTION - EDIT StockPriceKafkaProducer.java set batch size to 64K and linger to 50ms.

ACTION - RUN StockPriceKafkaProducer, and look at the stats.

ACTION - RUN SimpleStockPriceConsumer and look at record sizes, look at Producer metrics too.