Lab 6.6: Consumer with many threads

Welcome to the session 6 lab 6. The work for this lab is done in `~/kafka-training/labs/lab6.6`. In this lab, you are going to implement consumers with many threads.

Please refer to the Kafka course notes for any updates or changes to this lab.

Find the latest version of this lab here.

Lab Consumer with many threads

Recall that unlike Kafka producers, Kafka consumers are not thread-safe.

All network I/O happens in a thread of the application making calls. Kafka Consumers manage buffers, and connections state that threads can't share.

Remember only exception thread-safe method that the consumer has is `consumer.wakeup()`. The wakeup() method forces the consumer to throw a WakeupException on any thread the consumer client is blocking. You can use this to shut down a consumer from another thread. The solution and lab use wakeup. This is an easter egg.

Consumer with many threads

Decouple Consumption and Processing: One or more consumer threads that consume from Kafka and hands off ConsumerRecords instances to a thread pool where a worker thread can process it. This approach uses a blocking queue per topic partition to commit offsets to Kafka. This method is useful if per record processing is time-consuming.

An advantage is an option allows independently scaling consumers count and processors count. Processor threads are independent of topic partition count is also a significant advantage.

The problem with this approach is that guaranteeing order across processors requires care as threads execute independently and a later record could be processed before an earlier record and then you have to do consumer commits somehow with this out of order offsets.

How do you commit the position unless there is some order? You have to provide the ordering. (We use `ConcurrentHashMap` of `BlockingQueues` where the topic partition is the key (`TopicPartition`).)

One Consumer with Worker Threads

~/kafka-training/labs/lab6.6/src/main/java/com/cloudurable/kafka/consumer/StockPriceConsumerRunnable.java

Kafka Consumer: StockPriceConsumerRunnable

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.model.StockPrice;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Collections;
import java.util.Map;
import java.util.concurrent.*;
```

```
import java.util.concurrent.atomic.AtomicBoolean;

import static com.cloudurable.kafka.StockAppConstants.TOPIC;

public class StockPriceConsumerRunnable implements Runnable {
    private static final Logger logger =
            LoggerFactory.getLogger(StockPriceConsumerRunnable.class);

    private final Consumer<String, StockPrice> consumer;
    private final int readCountStatusUpdate;
    private final int threadIndex;
    private final AtomicBoolean stopAll;
    private boolean running = true;

    //Store blocking queue by TopicPartition
    Map<TopicPartition, BlockingQueue<ConsumerRecord>>
    commitQueueMap = new ConcurrentHashMap<>();

    //Worker pool.
    private final ExecutorService threadPool;
    ...
}
```

Map of queues per TopicPartition and threadPool;

StockPriceConsumerRunnable is still Runnable, but now it has many threads

You will need to add a worker `threadPool` to the `StockPriceConsumerRunnable`. The `StockPriceConsumerRunnable` uses a map of blocking queues per TopicPartition to manage sending offsets to Kafka.

Consumer with Worker Threads: Use Worker

~/kafka-training/labs/lab6.6/src/main/java/com/cloudurable/kafka/consumer/StockPriceConsumerRunnable.java

Kafka Consumer: StockPriceConsumerRunnable.pollRecordsAndProcess

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.model.StockPrice;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Collections;
import java.util.Map;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicBoolean;
```

```
import static com.cloudurable.kafka.StockAppConstants.TOPIC;

public class StockPriceConsumerRunnable implements Runnable {
...
    private void pollRecordsAndProcess(
            final Map<String, StockPrice> currentStocks,
            final int readCount) throws Exception {

        final ConsumerRecords<String, StockPrice> consumerRecords =
            consumer.poll(timeout: 100);

        if (consumerRecords.count() == 0) {
            if (stopAll.get()) this.setRunning(false);
            return;
        }

        consumer.Records.forEach(record ->
                currentStocks.put(record.key(),
                        new StockPriceRecord(record.value(), saved: true, record)
                ));

        threadPool.execute(() ->
                processRecords(currentStocks, consumerRecords));

        processCommits();

        if (readCount % readCountStatusUpdate == 0) {
            displayRecordsStatsAndStocks(currentStocks, consumerRecords);
        }
    }
...
}
```

Process the records async, and calls method `processCommits` will send offsets to Kafka.

Consumer with Worker Threads: processCommits

~/kafka-training/labs/lab6.6/src/main/java/com/cloudurable/kafka/consumer/StockPriceConsumerRunnable.java

Kafka Consumer: StockPriceConsumerRunnable.processCommits

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.model.StockPrice;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Collections;
import java.util.Map;
import java.util.concurrent.*;
```

```
import java.util.concurrent.atomic.AtomicBoolean;

import static com.cloudurable.kafka.StockAppConstants.TOPIC;

public class StockPriceConsumerRunnable implements Runnable {
...
    private void processCommits() {

        commitQueueMap.entrySet().forEach(queueEntry -> {
            final BlockingQueue<ConsumerRecord> queue = queueEntry.getValue();
            final TopicPartition topicPartition = queueEntry.getKey();

            ConsumerRecord consumerRecord = queue.poll();
            ConsumerRecord highestOffset = consumerRecord;

            while (consumerRecord != null) {
                if (consumerRecord.offset() > highestOffset.offset()) {
                    highestOffset = consumerRecord;
                }
                consumerRecord = queue.poll();
            }


            if (highestOffset != null) {
                logger.info(String.format("Sending commit %s %d",
                        topicPartition, highestOffset.offset()));
                try {
                    consumer.commitSync(Collections.singletonMap(topicPartition,
                            new OffsetAndMetadata(highestOffset.offset())));
                } catch (CommitFailedException cfe) {
                    logger.error("Failed to commit record", cfe);
                }
            }
        });
    }
...
}
```

Track the highest offset per TopicPartition, then call `consumer.commitSync` to save offset in Kafka per partition

Submit Records to Commit Queues per Partition

~/kafka-training/labs/lab6.6/src/main/java/com/cloudurable/kafka/consumer/StockPriceConsumerRunnable.java

Kafka Consumer: StockPriceConsumerRunnable

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.model.StockPrice;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.TopicPartition;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;

import java.util.Collections;
import java.util.Map;
import java.util.concurrent.*;
import java.util.concurrent.atomic.AtomicBoolean;

import static com.cloudurable.kafka.StockAppConstants.TOPIC;

public class StockPriceConsumerRunnable implements Runnable {
...
    private void processRecords(final Map<String, StockPrice> currentStocks,
                        final ConsumerRecords<String, StockPrice> consumerRecords) {

        consumerRecords.forEach(record ->
                currentStocks.put(record.key(), record.value()));

        consumerRecords.forEach(record -> {

            try {
                startTransaction();           //Start DB Transaction
                processRecord(record);
                commitTransaction();          //Commit DB Transaction
                commitRecordOffsetToKafka(record); //Send record to commit queue for
Kafka
            } catch (DatabaseException dbe) {
                rollbackTransaction();
            }
        });

    }
    ...
    private void commitRecordOffsetToKafka(ConsumerRecord<String, StockPrice> record)
{
        final TopicPartition topicPartition =
                new TopicPartition(record.topic(), record.partition());
        final BlockingQueue<ConsumerRecord> queue = commitQueueMap.computeIfAbsent (
                topicPartition,
                k -> new LinkedTransferQueue<>());
        queue.add(record);
    }
...
}
```

The method ***processRecord()***, if successful send a record to commit queue where it can be processed by `processCommits`. The method ***commitRecordOffsetToKafka()***, commits the record to its topicPartition Queue were it can be processed later by processCommits.

ConsumerMain

`ConsumerMain` now uses `wakeup` to stop consumer threads gracefully. Check this out. `ConsumerMain` also passes the number of worker threads that each `StockPriceConsumerRunnable` runs.

~/kafka-training/labs/lab6.6/src/main/java/com/cloudurable/kafka/consumer/StockPriceConsumerRunnable.java

Kafka Consumer: ConsumerMain

```
package com.cloudurable.kafka.consumer;

import com.cloudurable.kafka.StockAppConstants;
import com.cloudurable.kafka.model.StockPrice;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.ConsumerConfig;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.apache.kafka.common.serialization.StringDeserializer;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.ArrayList;
import java.util.List;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.stream.IntStream;

import static java.util.concurrent.Executors.newFixedThreadPool;

public class ConsumerMain {
    private static final Logger logger =
            LoggerFactory.getLogger(ConsumerMain.class);


    private static Consumer<String, StockPrice> createConsumer() {
        final Properties props = new Properties();
        props.put(ConsumerConfig.ENABLE_AUTO_COMMIT_CONFIG, false);
        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());
        props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
                StockDeserializer.class.getName());
        props.put(ConsumerConfig.MAX_POLL_RECORDS_CONFIG, 500);
        return new KafkaConsumer<>(props);
    }

    public static void main(String... args) throws Exception {
        final int threadCount = 5;
        final int workerThreads = 5;
        final ExecutorService executorService = newFixedThreadPool(threadCount);
        final AtomicBoolean stopAll = new AtomicBoolean();
        final List<Consumer> consumerList = new ArrayList<Consumer>(threadCount);
```

```
        IntStream.range(0, threadCount).forEach(index -> {
            final Consumer<String, StockPrice> consumer = createConsumer();
            final StockPriceConsumerRunnable stockPriceConsumer =
                    new StockPriceConsumerRunnable(consumer,
                            1000, index, stopAll, workerThreads);
            consumerList.add(consumer);
            executorService.submit(stockPriceConsumer);
        });

        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            logger.info("Stopping app");
            stopAll.set(true);
            sleep();
            consumerList.forEach(Consumer::wakeup);
            executorService.shutdown();
            try {
                executorService.awaitTermination(5_000, TimeUnit.MILLISECONDS);
                if (!executorService.isShutdown())
                    executorService.shutdownNow();
            } catch (InterruptedException e) {
                logger.warn("shutting down", e);
            }
            sleep();
            consumerList.forEach(Consumer::close);
        }));
    }

    private static void sleep() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            logger.error("", e);
        }
    }

}
```

Lab Work

Use the slides for Session 6 as a guide.

ACTION - EDIT `com.cloudurable.kafka.consumer.StockPriceConsumerRunnable` and follow the instructions in the file.

ACTION - EDIT `com.cloudurable.kafka.consumer.ConsumerMain` and follow the instructions in the file.

ACTION - RECREATE the topic with more partitions (HINT: bin/create-topic.sh).

ACTION - RUN ZooKeeper and Brokers if needed.

ACTION - RUN ConsumerMain from IDE

ACTION - RUN StockPriceKafkaProducer from IDE

ACTION - OBSERVE and then STOP consumers and producer

Expected behavior

It should run and should get messages like this:

Expected output

```
15:57:34.945 [pool-1-thread-2] INFO  c.c.k.c.StockPriceConsumerRunnable -
Sending commit stock-prices-9 320
15:57:34.947 [pool-1-thread-5] INFO  c.c.k.c.StockPriceConsumerRunnable -
Sending commit stock-prices-20 161
15:57:34.947 [pool-1-thread-3] INFO  c.c.k.c.StockPriceConsumerRunnable -
Sending commit stock-prices-12 317
15:57:34.947 [pool-1-thread-4] INFO  c.c.k.c.StockPriceConsumerRunnable -
Sending commit stock-prices-18 320
15:57:34.950 [pool-1-thread-5] INFO  c.c.k.c.StockPriceConsumerRunnable -
Sending commit stock-prices-24 305
15:57:34.952 [pool-1-thread-5] INFO  c.c.k.c.StockPriceConsumerRunnable -
Sending commit stock-prices-21 489
```

Try the following

Try using different worker pool sizes and different consumer thread pool sizes. Try adding a small wait for the processing. Try 10ms.

It should all run. Stop consumer and producer when finished.