Lab 6.3: StockPriceConsumer At Most Once and At Least Once

Welcome to the session 6 lab 3. The work for this lab is done in `~/kafka-training/labs/lab6.3`. In this lab, you are going to implement At-Most-Once and At-Least-Once message semantics from the consumer perspective.

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

Find the latest version of this lab here.

Lab At-Most-Once and At-Least-Once Semantics

Consumer Alive Detection

Consumers join consumer group after `subscribe` and then `poll()` is called. Automatically, a consumer sends periodic heartbeats to Kafka brokers server. If consumer crashes or is unable to send heartbeats for a duration of ***session.timeout.ms***, then the consumer is deemed dead, and its partitions reassigned.

Manual Partition Assignment

Instead of subscribing to the topic using subscribe, you can call ***assign(Collection)*** with the full topic partition list

```
String topic = "log-replication";
TopicPartition part0 = new TopicPartition(topic, 0);
TopicPartition part1 = new TopicPartition(topic, 1);
consumer.assign(Arrays.asList(part0, part1);
```

Using consumer as before with ***poll()***. Manual Partition assignment negates the use of group coordination and auto consumer failover. Each consumer acts independently even if in a consumer group (use unique group id to avoid confusion). You have to use ***assign()*** or ***subscribe()***, but not both. Use ***subscribe()*** to allow Kafka to manage failover and load balancing with consumer groups. Use ***assign()*** if you want to work with partitions exact.

Consumer Alive if Polling

Calling ***poll()*** marks consumer as alive. If consumer continues to call ***poll()***, then consumer is alive and in consumer group and gets messages for partitions assigned (has to call before every ***max.poll.interval.ms*** interval), If not calling ***poll()***, even if consumer is sending heartbeats, consumer is still considered dead. Processing of records from ***poll*** has to be faster than ***max.poll.interval.ms*** interval or your consumer could be marked dead! The ***max.poll.records*** is used to limit total records returned from a `poll` method call and makes it easier to predict max time to process the records on each poll interval.

Message Delivery Semantics

There are three message delivery semantics: at most once, at least once and exactly once.

At most once is messages may be lost but never redelivered. At least once is messages are never lost but may be redelivered. Exactly once is each message is delivered once and only once. Exactly once is preferred but more expensive, and requires more bookkeeping for the producer and consumer.

"At-Least-Once" - Delivery Semantics

~/kafka-training/labs/lab6.3/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Consumer: SimpleStockPriceConsumer.pollRecordsAndProcess

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

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class SimpleStockPriceConsumer {
...
    private static void pollRecordsAndProcess(
            int readCountStatusUpdate,
            Consumer<String, StockPrice> consumer,
            Map<String, StockPrice> map, int readCount) {
```

```
        final ConsumerRecords<String, StockPrice> consumerRecords =
                consumer.poll(1000);

        try {
            startTransaction();         //Start DB Transaction
            processRecords(map, consumerRecords);    //Process the records
            consumer.commitSync();
            commitTransaction();        //Commit DB Transaction
        } catch (CommitFailedException ex) {
            logger.error("Failed to commit sync to log", ex);
            rollbackTransaction();      //Rollback Transaction
        } catch (DatabaseException dte) {
            logger.error("Failed to write to DB", dte);
            rollbackTransaction();      //Rollback Transaction
        }
        if (readCount % readCountStatusUpdate == 0) {
            displayRecordsStatsAndStocks(map, consumerRecords);
        }
    }
...
}
```

ACTION - EDIT `src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java` and implement At-Least-Once Semantics

ACTION - RUN ZooKeeper and Brokers if needed.

ACTION - RUN SimpleStockPriceConsumer from IDE

ACTION - RUN StockPriceKafkaProducer from IDE

ACTION - OBSERVE and then STOP consumer and producer

"At-Most-Once" - Delivery Semantics

~/kafka-training/labs/lab6.3/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Consumer: SimpleStockPriceConsumer.pollRecordsAndProcess

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

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class SimpleStockPriceConsumer {
...
    private static void pollRecordsAndProcess(
            int readCountStatusUpdate,
            Consumer<String, StockPrice> consumer,
            Map<String, StockPrice> map, int readCount) {

        final ConsumerRecords<String, StockPrice> consumerRecords =
                consumer.poll(1000);

        try {

            startTransaction();         //Start DB Transaction
```

```
            consumer.commitSync();        //Commit the Kafka offset
            processRecords(map, consumerRecords);    //Process the records
            commitTransaction();        //Commit DB Transaction
        } catch (CommitFailedException ex) {
            logger.error("Failed to commit sync to log", ex);
            rollbackTransaction();      //Rollback Transaction
        } catch (DatabaseException dte) {
            logger.error("Failed to write to DB", dte);
            rollbackTransaction();      //Rollback Transaction
        }
        if (readCount % readCountStatusUpdate == 0) {
            displayRecordsStatsAndStocks(map, consumerRecords);
        }
    }
...
}
```

ACTION - EDIT `src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java` and implement At-Most-Once Semantics

ACTION - RUN SimpleStockPriceConsumer from IDE

ACTION - RUN StockPriceKafkaProducer from IDE

ACTION - OBSERVE and then STOP consumer and producer

Fine Grained "At-Least-Once"

~/kafka-training/labs/lab6.3/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Consumer: SimpleStockPriceConsumer.pollRecordsAndProcess

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

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class SimpleStockPriceConsumer {
...
    private static void pollRecordsAndProcess(
            int readCountStatusUpdate,
            Consumer<String, StockPrice> consumer,
            Map<String, StockPrice> map, int readCount)
            consumerRecords.forEach(record -> {
                try {
                    startTransaction();         //Start DB Transaction
                    //Commit Kafka at exact location for record, and only this record.
                    final TopicPartition recordTopicPartition =
                        new TopicPartition(record.topic(), record.partition());
                    final Map<TopicPartition, OffsetAndMetadata> commitMap =
                        Collections.singletonMap(recordTopicPartition,
                        new OffsetAndMetadata( offset: record.offset() + 1));
                    consumer.commitSync(commitMap); //Kafka Commit
                    processRecords(record);             //Process the record
                    commitTransaction();        //Commit DB Transaction
                } catch (CommitFailedException ex) {
                    logger.error("Failed to commit sync to log", ex);
                    rollbackTransaction();      //Rollback Transaction
```

```
                } catch (DatabaseException dte) {
                    logger.error("Failed to write to DB", dte);
                    rollbackTransaction();      //Rollback Transaction
                }
            });
        if (readCount % readCountStatusUpdate == 0) {
            displayRecordsStatsAndStocks(map, consumerRecords);
        }
    }
...
}
```

ACTION - EDIT `src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java` and implement fine-grained At-Most-Once Semantics

ACTION - RUN SimpleStockPriceConsumer from IDE

ACTION - RUN StockPriceKafkaProducer from IDE

ACTION - OBSERVE and then STOP consumer and producer

Fine Grained "At-Most-Once"

~/kafka-training/labs/lab6.3/src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java

Kafka Consumer: SimpleStockPriceConsumer.pollRecordsAndProcess

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

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Properties;

public class SimpleStockPriceConsumer {
...
    private static void pollRecordsAndProcess(
            int readCountStatusUpdate,
            Consumer<String, StockPrice> consumer,
            Map<String, StockPrice> map, int readCount)
            consumerRecords.forEach(record -> {
                try {
                    startTransaction();         //Start DB Transaction
                    processRecords(record);             //Process the record
                            //Commit Kafka at exact location for the record, and only this record.
                    final TopicPartition recordTopicPartition =
                        new TopicPartition(record.topic(), record.partition());
                    final Map<TopicPartition, OffsetAndMetadata> commitMap =
                        Collections.singletonMap(recordTopicPartition,
                        new OffsetAndMetadata( offset: record.offset() + 1));
                    consumer.commitSync(commitMap); //Kafka Commit
                    commitTransaction();        //Commit DB Transaction
                } catch (CommitFailedException ex) {
                    logger.error("Failed to commit sync to log", ex);
                    rollbackTransaction();      //Rollback Transaction
                } catch (DatabaseException dte) {
                    logger.error("Failed to write to DB", dte);
                    rollbackTransaction();      //Rollback Transaction
                }
            });
```

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

ACTION - EDIT `src/main/java/com/cloudurable/kafka/consumer/SimpleStockPriceConsumer.java` and implement fine-grained At-Least-Once Semantics

ACTION - RUN SimpleStockPriceConsumer from IDE

ACTION - RUN StockPriceKafkaProducer from IDE

ACTION - OBSERVE and then STOP consumer and producer

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