Bigdata Assignment 8

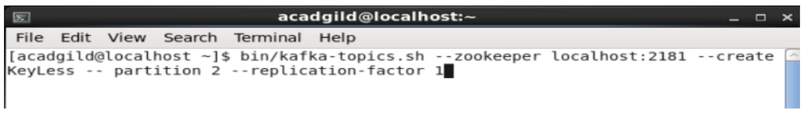
Task 1:
Create a kafka topic named KeyLessTopic.

Inside KeyLessTopic insert following data:
{"name":"John", "exp":16}
{"name":"Finn", "exp":20}
{"name":"Cylin", "exp":18}
{"name":"Mark", "exp":2}
{"name":"Akshay", "exp":14}

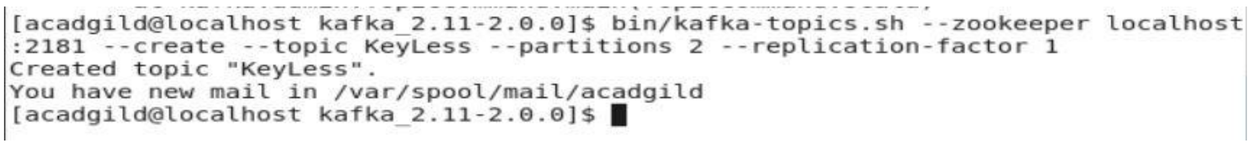
Code and Output: Same as Task 2

Task 2:
Create a console consumer that reads KeyLessTopic from beginning

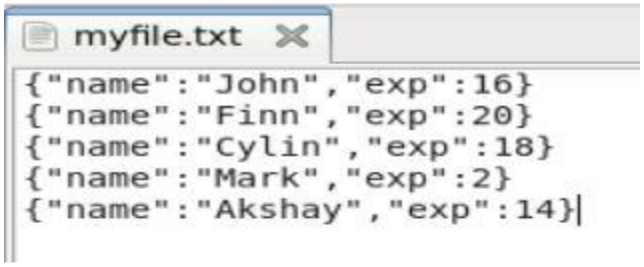
Output:



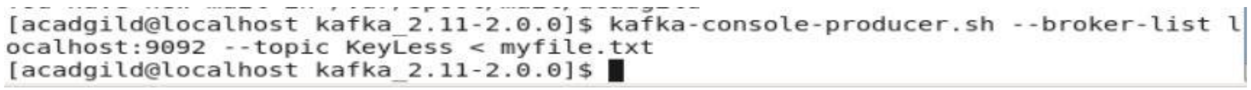
```
[acadgild@localhost ~]$ bin/kafka-topics.sh --zookeeper localhost:2181 --create
KeyLess -- partition 2 --replication-factor 1
```



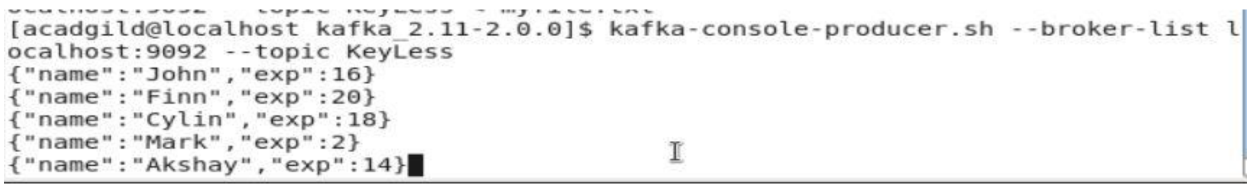
```
[acadgild@localhost kafka_2.11-2.0.0]$ bin/kafka-topics.sh --zookeeper localhost
:2181 --create --topic KeyLess --partitions 2 --replication-factor 1
Created topic "KeyLess".
You have new mail in /var/spool/mail/acadgild
[acadgild@localhost kafka_2.11-2.0.0]$
```



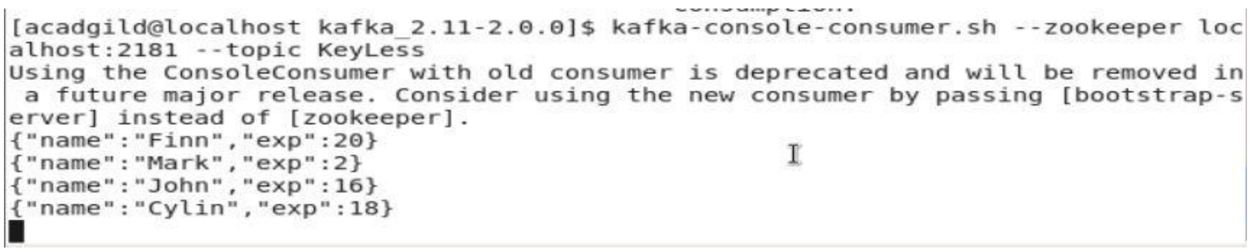
```
{"name":"John","exp":16}
{"name":"Finn","exp":20}
{"name":"Cylin","exp":18}
{"name":"Mark","exp":2}
{"name":"Akshay","exp":14}
```



```
[acadgild@localhost kafka_2.11-2.0.0]$ kafka-console-producer.sh --broker-list l
ocalhost:9092 --topic KeyLess < myfile.txt
[acadgild@localhost kafka_2.11-2.0.0]$
```



```
[acadgild@localhost kafka_2.11-2.0.0]$ kafka-console-producer.sh --broker-list l
ocalhost:9092 --topic KeyLess
{"name":"John","exp":16}
{"name":"Finn","exp":20}
{"name":"Cylin","exp":18}
{"name":"Mark","exp":2}
{"name":"Akshay","exp":14}
```



```
[acadgild@localhost kafka_2.11-2.0.0]$ kafka-console-consumer.sh --zookeeper loc
alhost:2181 --topic KeyLess
Using the ConsoleConsumer with old consumer is deprecated and will be removed in
 a future major release. Consider using the new consumer by passing [bootstrap-s
erver] instead of [zookeeper].
{"name":"Finn","exp":20}
{"name":"Mark","exp":2}
{"name":"John","exp":16}
{"name":"Cylin","exp":18}
```

Task 3:
Create a kafka topic named KeyedTopic. Inside KeyedTopic insert following data:
The part before comma(,) should be treated as key and after comma(,) should be treated as value
{"name":"John"},{"exp":16}
{"name":"Finn"},{"exp":20}
{"name":"Cylin"},{"exp":18}
{"name":"Mark"},{"exp":2}
{"name":"Akshay"},{"exp":14}

Output:

Task 4:
Create a console consumer that reads KeyedTopic from
beginning The key and value should be separated by '-'

Task 5:
Create a java program MyKafkaProducer.java that takes a file name and delimiter as input arguments. It should read the content of file line by line.
Fields in the file are in following order
1. Kafka Topic Name
2. Key
3. value
For every line, insert the key and value to the repsective Kafka broker in a fire and forget mode.

After record is sent, it should print appropriate message
on screen. Pass **dataset_producer.txt** as the input file and
-as delimiter.
LINK:
https://drive.google.com/file/d/0B_Qjau8wv1KoSnR5eHpKOF9rTFU/view?usp=sharing

Code:

```
import java.util.Properties;
import org.apache.kafka.clients.producer.Producer;
import org.apache.kafka.clients.producer.KafkaProducer;
import org.apache.kafka.clients.producer.ProducerRecord;

public class MyKafkaProducer {
public static void main(String[] args) throws Exception{
if(args.length == 0){
System.out.println("Enter topic name”);
return;
}

String topicName = args[0].toString();
Properties props = new Properties();
props.put("bootstrap.servers", “localhost:9092");
props.put("acks", “all");
props.put("retries", 0);
props.put("batch.size", 16384);
props.put("linger.ms", 1);
props.put("buffer.memory", 33554432);
props.put("key.serializer", "org.apache.kafka.common.serializa-tion.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serializa-tion.StringSerializer");
Producer<String, String> producer = new KafkaProducer
<String, String>(props);

for(int i = 0; i < 10; i++)
producer.send(new ProducerRecord<String, String>(topicName,
Integer.toString(i), Integer.toString(i)));
System.out.println(“Message sent successfully”);
producer.close();
}
}
```

Task 6:
Modify the previous program MyKafkaProducer.java and create a new Java program KafkaProducerWithAck.java
This should perform the same task as of KafkaProducer.java with some modification. When passing any data to a topic, it should wait for acknowledgement.
After acknowledgement is received from the broker, it should print the key and value which has been written to a specified topic.
The application should attempt for 3 retries before giving any exception. Pass **dataset_producer.txt** as the input file and -as delimiter.

Code:

```
public class MyKafkaProducerWithAck {
private static Scanner in;
public static void main(String[] argv)throws Exception
        {
                if (argv.length != 1)
                        {
                                System.err.println("Please specify 1 parameters ");
                                System.exit(-1);
                        }
                String topicName = argv[0];
                in = new Scanner(System.in);
                System.out.println("Enter message(type exit to quit)");

                Properties configProperties = new Properties();

        configProperties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,"localhost:90
92");

        configProperties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,"org.apache.
kafka.common.serialization.ByteArraySerializer");

        configProperties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,"org.apac
he.kafka.common.serialization.StringSerializer");
                org.apache.kafka.clients.producer.Producer producer = new
KafkaProducer<String, String>(configProperties);
                String line = in.nextLine();

                while(!line.equals("exit"))
                        {
                                ProducerRecord<String, String> rec = new
ProducerRecord<String, String>(topicName, line);
                                producer.send(rec);
                                line = in.nextLine();
                        }

                in.close();
                producer.close();
        }
}
```

Task 7

Read a stream of Strings, fetch the words which can be converted to numbers. Filter out the rows, where the sum of numbers in that line is odd.

Provide the sum of all the remaining numbers in that batch.

Code:

```
import java.util.Properties

import org.apache.kafka.clients.producer.Producer
import org.apache.kafka.clients.producer.KafkaProducer
import org.apache.kafka.clients.producer.ProducerRecord

// Configuration for Kafka brokers
val kafkaBrokers = "10.0.0.11:9092,10.0.0.15:9092,10.0.0.12:9092"
val topicName = "azurefeedback"
val props = new Properties()
props.put("bootstrap.servers", kafkaBrokers)
props.put("acks", "all")
props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer")
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer")
val producer = new KafkaProducer[String, String](props)
def sendEvent(message: String) = {
 val key = java.util.UUID.randomUUID().toString()
 producer.send(new ProducerRecord[String, String](topicName, key, message))
 System.out.println("Sent event with key: '" + key + "' and message: '" + message + "'\n")
}
```

Task 8

Read two streams

1. List of strings input by user

2. Real-time set of offensive words

Find the word count of the offensive words inputted by the user as per the real-time set of offensive words

Code:

```
import java.io.File
import java.util.Properties

import com.miguno.avro.Tweet
import com.miguno.kafkastorm.integration.IntegrationTest
import com.miguno.kafkastorm.kafka.{BaseKafkaProducerAppFactory,
ConsumerTaskContext, KafkaProducerApp, PooledKafkaProducerAppFactory}
import com.miguno.kafkastorm.logging.LazyLogging
import com.miguno.kafkastorm.spark.serialization.KafkaSparkStreamingRegistrator
import com.miguno.kafkastorm.testing.{EmbeddedKafkaZooKeeperCluster,
KafkaTopic}
import com.twitter.bijection.Injection
import com.twitter.bijection.avro.SpecificAvroCodecs
import kafka.message.MessageAndMetadata
import kafka.serializer.DefaultDecoder
import org.apache.commons.io.FileUtils
import org.apache.commons.pool2.impl.{GenericObjectPool, GenericObjectPoolConfig}
import org.apache.spark.SparkConf
import org.apache.spark.storage.StorageLevel
import org.apache.spark.streaming.kafka.KafkaUtils
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.scalatest._
import scala.collection.mutable
import scala.concurrent.duration._
import scala.language.reflectiveCalls
import scala.util.{Failure, Success}

@DoNotDiscover

class KafkaSparkStreamingSpec extends FeatureSpec with Matchers with
BeforeAndAfterEach with GivenWhenThen with LazyLogging {
  implicit val specificAvroBinaryInjectionForTweet =
SpecificAvroCodecs.toBinary[Tweet]
  private val inputTopic = KafkaTopic("testing-input")
  private val outputTopic = KafkaTopic("testing-output")
  private val sparkCheckpointRootDir = {
    val r = (new scala.util.Random).nextInt()
    val path = Seq(System.getProperty("java.io.tmpdir"), "spark-test-checkpoint-" +
r).mkString(File.separator)
    new File(path)
  }
  private val kafkaZkCluster = new EmbeddedKafkaZooKeeperCluster(topics =
Seq(inputTopic, outputTopic))
  private var ssc: StreamingContext = _
```

```
 override def beforeEach() {
  kafkaZkCluster.start()
  prepareSparkStreaming()
 }

 private def prepareSparkStreaming(): Unit = {
  val sparkConf = {
    val conf = new SparkConf().setAppName("kafka-spark-streaming-starter")
    // Make sure you give enough cores to your Spark Streaming application. You need
cores for running "receivers"
    // and for powering the actual the processing. In Spark Streaming, each receiver is
responsible for 1 input
    // DStream, and each receiver occupies 1 core. If all your cores are occupied by
receivers then no data will be
    // processed!
    // https://spark.apache.org/docs/1.1.0/streaming-programming-guide.html
    val cores = inputTopic.partitions + 1
    conf.setMaster(s"local[$cores]")
    // Use Kryo to speed up serialization, recommended as default setup for Spark
Streaming
    // http://spark.apache.org/docs/1.1.0/tuning.html#data-serialization
    conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
    conf.set("spark.kryo.registrator",
classOf[KafkaSparkStreamingRegistrator].getName)
    // Enable experimental sort-based shuffle manager that is more memory-efficient in
environments with small
    // executors, such as YARN. Will most likely become the default in future Spark
versions.
    // https://spark.apache.org/docs/1.1.0/configuration.html#shuffle-behavior
    conf.set("spark.shuffle.manager", "SORT")
    // Force RDDs generated and persisted by Spark Streaming to be automatically
unpersisted from Spark's memory.
    // The raw input data received by Spark Streaming is also automatically cleared.
(Setting this to false will
    // allow the raw data and persisted RDDs to be accessible outside the streaming
application as they will not be
    // cleared automatically. But it comes at the cost of higher memory usage in Spark.)
    // http://spark.apache.org/docs/1.1.0/configuration.html#spark-streaming
    conf.set("spark.streaming.unpersist", "true")
    conf
  }
  val batchInterval = Seconds(1)
  ssc = new StreamingContext(sparkConf, batchInterval)
  ssc.checkpoint(sparkCheckpointRootDir.toString)
 }
 override def afterEach() {
  kafkaZkCluster.stop()
  terminateSparkStreaming()
 }
 private def terminateSparkStreaming() {
  ssc.stop(stopSparkContext = true, stopGracefully = true)
  ssc = null
  FileUtils.deleteQuietly(sparkCheckpointRootDir)
 }
 val fixture = {
  val BeginningOfEpoch = 0.seconds
  val AnyTimestamp = 1234.seconds
  val now = System.currentTimeMillis().millis
  new {
    val t1 = new Tweet("ANY_USER_1", "ANY_TEXT_1", now.toSeconds)
    val t2 = new Tweet("ANY_USER_2", "ANY_TEXT_2", BeginningOfEpoch.toSeconds)
    val t3 = new Tweet("ANY_USER_3", "ANY_TEXT_3", AnyTimestamp.toSeconds)
    val messages = Seq(t1, t2, t3)
  }
 }
 info("As a user of Spark Streaming")
 info("I want to read Avro-encoded data from Kafka")
 info("so that I can quickly build Kafka<->Spark Streaming data flows")
 feature("Basic functionality") {
  scenario("User creates a Spark Streaming job that reads from and writes to Kafka",
IntegrationTest) {
    Given("a ZooKeeper instance")
    And("a Kafka broker instance")
    And("some tweets")
    val tweets = fixture.messages
    And(s"a synchronous Kafka producer app that writes to the topic $inputTopic")
    val producerApp = {
     val config = {
        val c = new Properties
        c.put("producer.type", "sync")
        c.put("client.id", "kafka-spark-streaming-test-sync-producer")
        c.put("request.required.acks", "1")
        c
      }

      kafkaZkCluster.createProducer(inputTopic.name, config).get
    }
    And(s"a single-threaded Kafka consumer app that reads from topic $outputTopic
and Avro-decodes the incoming data")
    val actualTweets = new mutable.SynchronizedQueue[Tweet]
    def consume(m: MessageAndMetadata[Array[Byte], Array[Byte]], c:
ConsumerTaskContext) {
      val tweet = Injection.invert(m.message())
      for {t <- tweet} {
        logger.info(s"Consumer thread ${c.threadId}: received Tweet $t from
${m.topic}:${m.partition}:${m.offset}")
        actualTweets += t
      }
    }
    kafkaZkCluster.createAndStartConsumer(outputTopic.name, consume)
    val waitForConsumerStartup = 300.millis
    logger.debug(s"Waiting $waitForConsumerStartup for the Kafka consumer to start
up")
    Thread.sleep(waitForConsumerStartup.toMillis)

   When("I Avro-encode the tweets and use the Kafka producer app to sent them to
Kafka")
    tweets foreach {
      case tweet =>
        val bytes = Injection(tweet)
        logger.info(s"Synchronously sending Tweet $tweet to topic
${producerApp.defaultTopic}")
       producerApp.send(bytes)
    }
    And(s"I run a streaming job that reads tweets from topic $inputTopic and writes
them as-is to topic $outputTopic")
    // Required to gain access to RDD transformations via implicits. We include this
```

```
import here to highlight its
   // importance and where it will take effect.
   import org.apache.spark.SparkContext._

   val kafkaStream = {
    val sparkStreamingConsumerGroup = "spark-streaming-consumer-group"
    val kafkaParams = Map[String, String](
     "zookeeper.connect" -> kafkaZkCluster.zookeeper.connectString,
     "group.id" -> sparkStreamingConsumerGroup,
     // CAUTION: Spark's use of auto.offset.reset is DIFFERENT from Kafka's behavior!
     // https://issues.apache.org/jira/browse/SPARK-2383
     "auto.offset.reset" -> "smallest", // must be compatible with when/how we are
writing the input data to Kafka
     "zookeeper.connection.timeout.ms" -> "1000")
    // The code below demonstrates how to read from all the topic's partitions. We
create an input DStream for each
    // partition of the topic, unify those streams, and then repartition the unified
stream. This last step allows
    // us to decouple the desired "downstream" parallelism (data processing) from the
"upstream" parallelism
    // (number of partitions).
    //
    // Note: In our case the input topic has only 1 partition, so you won't see a real
effect of this fancy setup.
    //
    // And yes, the way we do this looks quite strange -- we combine a hardcoded `1` in
the topic map with a
    // subsequent `(1..N)` construct. But this approach is the recommended way.
    // See http://spark.apache.org/docs/1.1.0/streaming-programming-
guide.html#reducing-the-processing-time-of-each-batch
    val streams = (1 to inputTopic.partitions) map { _ =>
     KafkaUtils.createStream[Array[Byte], Array[Byte], DefaultDecoder,
DefaultDecoder](
      ssc,
      kafkaParams,
      Map(inputTopic.name -> 1),
      storageLevel = StorageLevel.MEMORY_ONLY_SER // or:
StorageLevel.MEMORY_AND_DISK_SER
     ).map(_._2)
```

```
      }
      val unifiedStream = ssc.union(streams) // Merge the "per-partition" DStreams
      val sparkProcessingParallelism = 1 // You'd probably pick a higher value than 1 in
production.
      // Repartition distributes the received batches of data across specified number of
machines in the cluster
      // before further processing. Essentially, what we are doing here is to decouple
processing parallelism from
      // reading parallelism (limited by #partitions).
      unifiedStream.repartition(sparkProcessingParallelism)
    }
    // We use accumulators to track the number of consumed and produced messages
across all tasks. Named accumulators
    // are also said to be displayed in Spark's UI but we haven't found them yet. :-)
    val numInputMessages = ssc.sparkContext.accumulator(0L, "Kafka messages
consumed")
    val numOutputMessages = ssc.sparkContext.accumulator(0L, "Kafka messages
produced")
    // We use a broadcast variable to share a pool of Kafka producers, which we use to
write data from Spark to Kafka.
    val producerPool = {
      val pool = createKafkaProducerPool(kafkaZkCluster.kafka.brokerList,
outputTopic.name)
      ssc.sparkContext.broadcast(pool)
    }
    // We also use a broadcast variable for Bijection/Injection.
    val converter = ssc.sparkContext.broadcast(specificAvroBinaryInjectionForTweet)
    // Note: When working on PairDStreams (which we are not doing here) do not
forget to import the corresponding
    // implicits (see import statement below) in order to pick up implicits that allow
`DStream.reduceByKey` etc.
    // (versus `DStream.transform(rddBatch => rddBatch.reduceByKey()`). In other
words, DStreams appear to be
    // relatively featureless until you import this implicit -- if you don't, you must
operate on the underlying RRDs
    // explicitly which is not ideal.
    //
    // import org.apache.spark.streaming.StreamingContext.toPairDStreamFunctions
    //
    // See https://www.mail-archive.com/user@spark.apache.org/msg10105.html
    // Define the actual data flow of the streaming job
    kafkaStream.map { case bytes =>
      numInputMessages += 1
      converter.value.invert(bytes) match {
        case Success(tweet) => tweet
        case Failure(e) =>
      }
    }.foreachRDD(rdd => {
      rdd.foreachPartition(partitionOfRecords => {
        val p = producerPool.value.borrowObject()
        partitionOfRecords.foreach { case tweet: Tweet =>
          val bytes = converter.value.apply(tweet)
          p.send(bytes)
          numOutputMessages += 1
        }
        producerPool.value.returnObject(p)
      })
    })

   // Run the streaming job (but run it for a maximum of 2 seconds)
    ssc.start()
    ssc.awaitTermination(2.seconds.toMillis)
    Then("the Spark Streaming job should consume all tweets from Kafka")
    numInputMessages.value should be(tweets.size)
    And("the job should write back all tweets to Kafka")
    numOutputMessages.value should be(tweets.size)
    And("the Kafka consumer app should receive the original tweets from the Spark
Streaming job")
    val waitToReadSparkOutput = 300.millis
    logger.debug(s"Waiting $waitToReadSparkOutput for Kafka consumer to read Spark
Streaming output from Kafka")
    Thread.sleep(waitToReadSparkOutput.toMillis)
    actualTweets.toSeq should be(tweets.toSeq)
    // Cleanup
    producerApp.shutdown()
   }
 }
 private def createKafkaProducerPool(brokerList: String, topic: String):
```

```
GenericObjectPool[KafkaProducerApp] = {
   val producerFactory = new BaseKafkaProducerAppFactory(brokerList, defaultTopic
= Option(topic))
   val pooledProducerFactory = new
PooledKafkaProducerAppFactory(producerFactory)
   val poolConfig = {
     val c = new GenericObjectPoolConfig
     val maxNumProducers = 10
     c.setMaxTotal(maxNumProducers)
     c.setMaxIdle(maxNumProducers)
     c
   }
   new GenericObjectPool[KafkaProducerApp](pooledProducerFactory, poolConfig)
 }
}
```