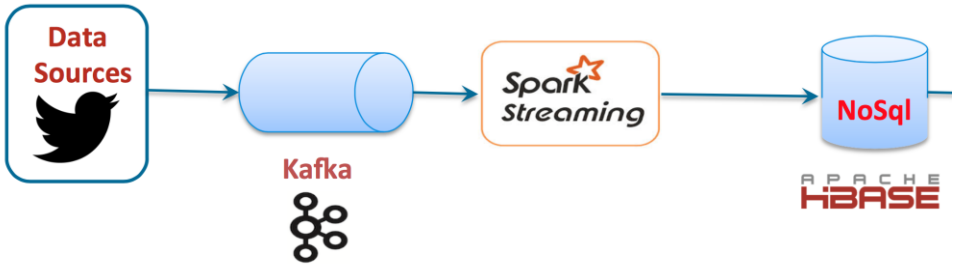
SPARK STREAMING with KAFA and HBASE

Task Requirement:

Write an application that produce the messages to kafka-topic, by using Spark Streaming consume the messages from kafka-topic, then commit the consumed messages to HBASE.



STREAMING DATA PIPELINE WITH APACHE KAFKA, SPARK STREAMING, HBASE

Data Ingestion:

The java program 'Producer.java' is a simulator for 'IOT devices', the program will generate the data according to the following template.

```
{
        "data": {
                "deviceId: UUID",
                "temperature: Integer",
                "location": {
                        "latitude": "long",
                        "Longitude": "long"
                },
                "time": "Timestamp"
        }
}
```

The program simulates 3 IOT devices and sends signal to kafka every second.

Configuration properties: Kafka uses key-value pairs in the property file for configuration.

```
Properties configProperties = new Properties();

configProperties.put("bootstrap.servers","localhost:9092");
configProperties.put("key.serializer","org.apache.kafka.common.serialization.StringSerializer");
configProperties.put("value.serializer","org.apache.kafka.common.serialization.StringSerializer");
```

bootstrap.servers : The property bootstrap.servers defines list of brokers.

key.serializer : The property “key.serializer” defines what Serializer to use when preparing the message for transmission to the Broker. In our example we use a simple String Serializer for the key, because key is simply a String.

value.serializer : The property “value.serializer” defines what Serializer to use when preparing the message for transmission to the Broker. In our example we use a simple String Serializer for the value, because value is simply a String.

The Kafka Producer class provides an option to connect a Kafka broker in its constructor.

```
final KafkaProducer<String, String> producer = new KafkaProducer<String, String> (configProperties);
```

KafkaProducer class provides send method to send messages asynchronously to a topic.

ProducerRecord − The producer manages a buffer of records waiting to be sent.

```
Final String weather1 =  "{'data': {'deviceId': '11c1310e-c0c2-461b-a4eb-f6bf8da2d23c',
'temperature':" + (int) (Math.random() * ((40 - 10) + 1)) + (int) (Math.random() * ((40 - 10) +
1))+ ",'location': { 'latitude': '52.14691120000001','longitude':'11.658838699999933'},
'time':"+ new Date().getTime() + "}}";

ProducerRecord<String,String> rec1 = new ProducerRecord<String,String>("topic1", weather1)

producer.send(rec1);
```

The above code sends the record to topic ‘topic1’, in ‘json’ format.

```
Runnable runnable = new Runnable() {
    public void run() {
        while (true) {
        ProducerRecord<String, String> rec1 = new ProducerRecord<String, String>("topic1", weather1);
        producer.send(rec1);
            try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                            }
                    }
                }
            };
        Thread thread = new Thread(runnable);
         thread.start();
```

The above module sends a signal to Apache Kafka every second.

Data Transformation:

The java program 'Consumer.java' is a 'SparkStreaming' job which reads data from 'apache-kafka' and stores it into a 'HBase' table.

Spark Streaming to receive data from Kafka:

Initialize the Streaming Context:

To initialize a Spark Streaming program, a **StreamingContext** object has to be created which is the main entry point of all Spark Streaming functionality.

A JavaStreamingContext object can be created from a SparkConf object.

```
SparkConf conf = new SparkConf().setAppName("Kafka Spark StreamingAPP").setMaster("local") ;
JavaStreamingContext ssc = new JavaStreamingContext(conf, new Duration(2000));
```

Configuration properties:

The topics of kafka from which data has to be read are stored in a variable of type ‘Set’. Parameters for Kafka Connection are stored in a ‘Map’ object.

```
Set<String> topics = new HashSet<String>(Arrays.asList("topic1","topic2","topic3"));

Map<String, String> kafkaParams = new HashMap<String, String>();

kafkaParams.put("metadata.broker.list", "localhost:9092");

kafkaParams.put(ConsumerConfig.GROUP_ID_CONFIG, UUID.randomUUID().toString());

kafkaParams.put(ConsumerConfig.CLIENT_ID_CONFIG, "Consumer");

kafkaParams.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "smallest");
```

metadata.broker.list : “metadata.broker.list” defines where the Consumer can find a one or more Brokers, to connect to Kafka Cluster.

ConsumerConfig.Group_ID_CONFIG : identifier of the group consumer belongs to.

ConsumerConfig.CLIENT_ID_CONFIG : the id string to pass to the server when making requests.

ConsumerConfig.AUTO_OFFSET_RESET_CONFIG : ‘smallest’, reads the data using Kafka Consumer api ‘from beginning’.

The Spark-Kafka integration jar contains a class ‘kafkaUtils’ using which we can directly integrate Kafka with Spark Streaming. It helps to build a connection with Kafka broker. Provide the necessary parameters like Streaming Context, type of data transferred, Kafka Params containing broker hosts, and topic names. The outputs received from this stream of data is a ‘DStream’ object and we call it ‘directKafkaStream’ and this is essentially a sequence of RDDs. This stream of data is filled up RDD objects and each RDD is essentially a paired RDD filled with a tuple.

```
import org.apache.spark.streaming.kafka.*;

JavaPairInputDStream<String, String> directKafkaStream = KafkaUtils.createDirectStream(ssc, String.class, String.class,StringDecoder.class, StringDecoder.class, kafkaParams,topics);
```

Go over the content of the 'DStream' object and Extract each RDD from it, and within java lambda function extract the partition from each object that we iterate through. each partition contains key value pairs, where value contains the actual data which it reads from the Kafka topic.

For each partition get the value and convert it into json object.

```
directKafkaStream.foreachRDD(rdd -> {
        rdd.foreachPartition(iterator -> {
            while (iterator.hasNext()) {
                Tuple2<String, String> next = iterator.next();
                        JSONObject jobj = new JSONObject(next._2());
                ….............
        }}}
```

Commit the messages to HBASE:

Create a configuration with HBase resources, and set the zookeeper properties:

```
Configuration hconf = HBaseConfiguration.create();
hconf.set("hbase.zookeeper.quorum", "localhost");
hconf.set("hbase.zookeeper.property.clientPort","2181");
```

Connect to HBase and get the table:

```
Connection connection = ConnectionFactory.createConnection(hconf);
Table table = connection.getTable(TableName.valueOf("ultratendency"));
```

Insert data into HBase table

To insert data into table addColumn() method of Put class is used.

```
Put p = new Put(Bytes.toBytes("row" + i));
```

addColumn(byte[] family, byte[] qualifier, byte[] value)

Add the specified column and value to this Put operation.

Family is column family , qualifier is column name, value is actual value

The method addColumn() accepts the byte array, so get the values from json object (which is read from kafka topic). Convert every value to byte and put the values into HBase Table.

```
JSONObject dataobj = jobj.getJSONObject("data");

int temp =  (int) dataobj.get("temperature");

String temps = String.valueOf(temp);

p.addColumn(Bytes.toBytes("weather-data"), Bytes.toBytes("temperature"),

Bytes.toBytes(temps));

Table.put(p);
```

Foreach RDD object, for each partition get the messages from Kafka topic, convert it into json object, create HBase table, read the values from json object and put it into HBase table.

```
ssc.start();             // Start the computation

ssc.awaitTermination();   // Wait for the computation to terminate
```