In this chapter, we will learn how to integrate Kafka with Apache Storm.

About Storm

Storm was originally created by Nathan Marz and team at BackType. In a short time, Apache Storm became a standard for distributed real-time processing system that allows you to process a huge volume of data.

Storm is very fast and a benchmark clocked it at over a million tuples processed per second per node. Apache Storm runs continuously, consuming data from the configured sources (Spouts) and passes the data down the processing pipeline (Bolts). Com- bined, Spouts and Bolts make a Topology.

Integration with Storm

Kafka and Storm naturally complement each other, and their powerful cooperation enables real-time streaming analytics for fast-moving big data. Kafka and Storm integration is to make easier for developers to ingest and publish data streams from Storm topologies.

Conceptual flow

A spout is a source of streams. For example, a spout may read tuples off a Kafka Topic and emit them as a stream. A bolt consumes input streams, process and possibly emits new streams. Bolts can do anything from running functions, filtering tuples, do streaming aggregations, streaming joins, talk to databases, and more. Each node in a Storm topology executes in parallel. A topology runs indefinitely until you terminate it. Storm will automatically reassign any failed tasks.

Additionally, Storm guarantees that there will be no data loss, even if the machines go down and messages are dropped.

Let us go through the Kafka-Storm integration API's in detail. There are three main classes to integrate Kafka with Storm. They are as follows --

BrokerHosts - ZkHosts & StaticHosts

BrokerHosts is an interface and ZkHosts and StaticHosts are its two main implementations. ZkHosts is used to track the Kafka brokers dynamically by maintaining the details in ZooKeeper, while StaticHosts is used to manually / statically set the Kafka brokers and its details. ZkHosts is the simple and fast way to access the Kafka broker.

The signature of ZkHosts is as follows --

```
public  ZkHosts(String  brokerZkStr,  String  brokerZkPath)
public  ZkHosts(String  brokerZkStr)
```

Where brokerZkStr is ZooKeeper host and brokerZkPath is the ZooKeeper path to maintain the Kafka broker details.

KafkaConfig API

This API is used to define configuration settings for the Kafka cluster. The signature of KafkaCon- fig is defined as follows:

```
public KafkaConfig(BrokerHosts hosts, string topic)
```

- Hosts - The BrokerHosts can be ZkHosts / StaticHosts.
- Topic - topic name.

SpoutConfig API

Spoutconfig is an extension of KafkaConfig that supports additional ZooKeeper information.

```
public SpoutConfig(BrokerHosts hosts, string topic, string zkRoot, string id)
```

- Hosts - The BrokerHosts can be any implementation of BrokerHosts interface
- Topic - topic name.
- zkRoot - ZooKeeper root path.
- id - The spout stores the state of the offsets its consumed in Zookeeper. The id should uniquely identify your spout.

SchemeAsMultiScheme

SchemeAsMultiScheme is an interface that dictates how the ByteBuffer consumed from Kafka gets transformed into a storm tuple. It is derived from MultiScheme and accept implementation of Scheme class. There are lot of implementation of Scheme class and one such implementation is StringScheme, which parses the byte as a simple string. It also controls the naming of your output field. The signature is defined as follows.

```
public SchemeAsMultiScheme(Scheme scheme)
```

- Scheme - byte buffer consumed from kafka.

KafkaSpout API

KafkaSpout is our spout implementation, which will integrate with Storm. It fetches the mes- sages from kafka topic and emits it into Storm ecosystem as tuples. KafkaSpout get its config- uration details from SpoutConfig.

Below is a sample code to create a simple Kafka spout.

```
BrokerHosts hosts = new ZkHosts(zkConnString);

// ZooKeeper connection string
SpoutConfig spoutConfig = new SpoutConfig(hosts, topicName, "/" + topicName
UUID.randomUUID().toString());

/*ZooKeeper connection info, topic name, Zkroot will be used as root to store your
consumer's offset, id should uniquely identify your spout.*/

spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());

//convert the ByteBuffer to String.
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
//Assign SpoutConfig to KafkaSpout.
```

Bolt Creation

Bolt is a component that takes tuples as input, processes the tuple, and produces new tuples as output. Bolts will implement IRichBolt interface. In this program, two bolt classes WordSplitter- Bolt and WordCounterBolt are used to perform the operations.

IRichBolt interface has the following methods:

- Prepare − Provides the bolt with an environment to execute. The executors will run this method to initialize the spout.
- Execute − Process a single tuple of input.
- Cleanup − Called when a bolt is going to shut down.
- declareOutputFields − Declares the output schema of the tuple.

Let us create SplitBolt.java, which implements the logic to split a sentence into words and CountBolt.java, which implements logic to separate unique words and count its occurrence.

SplitBolt.java

```
import java.util.Map;
import backtype.storm.tuple.Tuple;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.storm.task.OutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.IRichBolt;
import backtype.storm.task.TopologyContext;
public class SplitBolt implements IRichBolt {
        private OutputCollector collector;
        @Override
        public void prepare(Map stormConf, TopologyContext context, OutputCollector
collector) {
```

```
                this.collector  =  collector;
        }

        @Override
        public void execute(Tuple input)  {
                String sentence = input.getString(0);
                String[] words = sentence.split("  ");
                for(String word: words)  {
                        word  =  word.trim();
                        if(!word.isEmpty())  {
                                word  =  word.toLowerCase();
                                collector.emit(new  Values(word));
                        }
                }
                collector.ack(input);
        }
        @Override
        public void declareOutputFields(OutputFieldsDeclarer declarer)  {
declarer.declare(new Fields("word"));
        }
        @Override
        public  void  cleanup()  {
        }
        @Override
        public  Map<String,  Object>  getComponentConfiguration()  {
                return  null;
        }
}
```

CountBolt.java

```
import java.util.Map;
import java.util.HashMap;
import backtype.storm.tuple.Tuple;
import backtype.storm.task.OutputCollector;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.IRichBolt;
import backtype.storm.task.TopologyContext;

public class CountBolt implements IRichBolt{

        Map<String,  Integer>  counters;
        private  OutputCollector  collector;
        @Override
        public  void  prepare(Map  stormConf,  TopologyContext  context,
                        OutputCollector  collector)  {
                this.counters  =  new  HashMap<String,  Integer>();
                this.collector  =  collector;
        }

        @Override
        public  void  execute(Tuple  input)  {
                String  str  =  input.getString(0);
                if(!counters.containsKey(str)){
                        counters.put(str,  1);
```

```
                }else{
                        Integer  c  =  counters.get(str)  +1;
                        counters.put(str,  c);
                }
                collector.ack(input);
        }
        @Override
        public  void  cleanup()  {
                for(Map.Entry<String,  Integer>  entry:counters.entrySet()){
System.out.println(entry.getKey()+"  :  "  +  entry.getValue());
                }
        }

        @Override
        public  void  declareOutputFields(OutputFieldsDeclarer  declarer)  {
        }

        @Override
        public  Map<String,  Object>  getComponentConfiguration()  {
                 return  null;
        }

}
```

Submitting to Topology

The Storm topology is basically a Thrift structure. TopologyBuilder class provides simple and easy methods to create complex topologies. The TopologyBuilder class has methods to set spout (setSpout) and to set bolt (setBolt). Finally, TopologyBuilder has createTopology to create to- pology. shuffleGrouping and fieldsGrouping methods help to set stream grouping for spout and bolts.

Local Cluster -- For development purposes, we can create a local cluster using \"LocalCluster\" object and then submit the topology using \"submitTopology\" method of \"LocalCluster\" class.

KafkaStormSample.java

```
import backtype.storm.Config;
import backtype.storm.LocalCluster;
import backtype.storm.topology.TopologyBuilder;

import java.util.ArrayList;
import java.util.List;
import java.util.UUID;

import backtype.storm.spout.SchemeAsMultiScheme;
import storm.kafka.trident.GlobalPartitionInformation;
import storm.kafka.ZkHosts;
import storm.kafka.Broker;
import storm.kafka.StaticHosts;
import storm.kafka.BrokerHosts;
import storm.kafka.SpoutConfig;
```

```
import storm.kafka.KafkaConfig;
import storm.kafka.KafkaSpout;
import storm.kafka.StringScheme;

public class KafkaStormSample {

        public static void main(String[] args) throws Exception{
                Config config = new Config();
                config.setDebug(true);
                config.put(Config.TOPOLOGY_MAX_SPOUT_PENDING, 1);


                String zkConnString = "localhost:2181";
                String topic = "my-first-topic";
                BrokerHosts hosts = new ZkHosts(zkConnString);


                SpoutConfig kafkaSpoutConfig = new SpoutConfig (hosts, topic,
"/" + topic, UUID.randomUUID().toString());


                kafkaSpoutConfig.bufferSizeBytes = 1024 * 1024 * 4;
                kafkaSpoutConfig.fetchSizeBytes = 1024 * 1024 * 4;
                kafkaSpoutConfig.forceFromStart = true;
                kafkaSpoutConfig.scheme = new SchemeAsMultiScheme(new
StringScheme());


                TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("kafka-spout", new KafkaSpout(kafkaSpoutConfig));

                builder.setBolt("word-spitter", new
SplitBolt()).shuffleGrouping("kafka-spout");
                builder.setBolt("word-counter", new
CountBolt()).shuffleGrouping("word-spitter");

                LocalCluster cluster = new LocalCluster();
cluster.submitTopology("KafkaStormSample", config, builder.createTopology());

                Thread.sleep(10000);

                cluster.shutdown();

        }


}
```

Before moving compilation, Kakfa-Storm integration needs curator ZooKeeper client java library. Curator version 2.9.1 support Apache Storm version 0.9.5 (which we use in this tutorial). Download the below specified jar files and place it in java class path.

- curator-client-2.9.1.jar
- curator-framework-2.9.1.jar

After including dependency files, compile the program using the following command,

```
javac  -cp  "/path/to/Kafka/apache-storm-0.9.5/lib/*"  *.java
```

Execution

Start Kafka Producer CLI (explained in previous chapter), create a new topic called "my-first- topic" and provide some sample messages as shown below:

```
hello
kafka
storm
spark
test  message
another test message
```

Now execute the application using the following command:

```
java  -cp  "/path/to/Kafka/apache-storm-0.9.5/lib/*":.  KafkaStormSample
```

The sample output of this application is specified below --

```
storm : 1
test : 2
spark : 1
another : 1
kafka  : 1
hello  : 1
message : 2
```