IKORN SOLUTIONS

DOCUMENT FOR IFP PROJECT

Tìm hiểu về Apache kafka và Apache Storm

:

Người thực hiện:
Nguyễn Quốc Long

TP.Hồ Chí Minh, 25/07/2018.

Mục lục

Tìm hiểu về Apache Kafka và Apache storm

Nguyễn Quốc Long[1]

No Institute Given

Tóm tắt nội dung. Tài liệu tìm hiểu về Apache Kafka và Apache Storm

Từ khóa: algorithm, apache storm, apache kafaka, big data ...

1 Giới thiệu

Apache kafka là hệ thống truyền tin nhắn được xây dựng cho big data. Kafka cho phép ứng dụng xây dựng trên các nền tảng khác nhau và giao tiếp với nhau thông qua việc truyền thông điệp theo cơ chế bất đồng bộ.

Apache storm là hệ thống để xử lý streaming data trong thời gian thực. Apache Storm hỗ trợ nhiều ngôn ngữ như: scala, python, Javascript, java, vv...

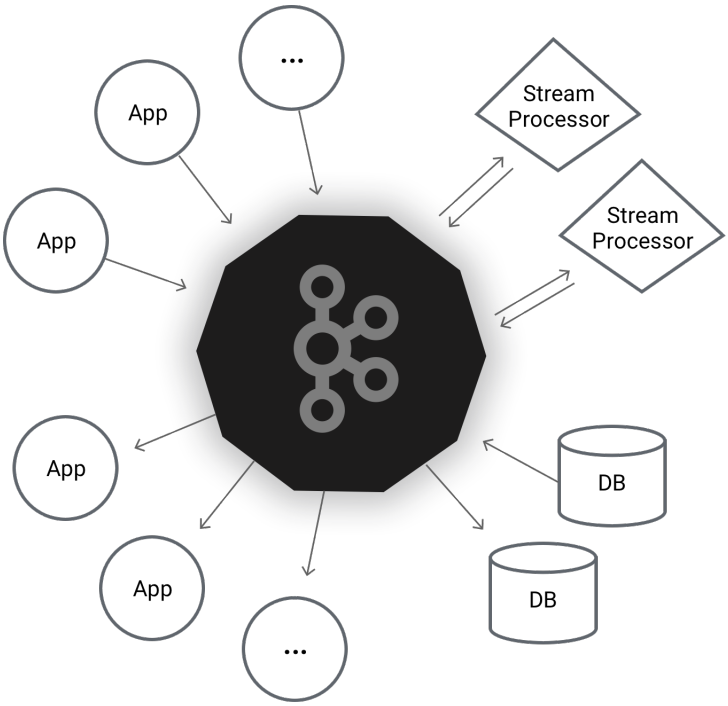
2 Tóm tắt nội dung

Bài tìm hiểu này sẽ trình bày cá nội dung sau đây:
– Khái niệm cơ bản về apache Kafka
– Khái niệm cơ bản về apache Storm
– Apache Kafka phối hợp với Apache storm

3 Nội dung tìm hiểu

3.1 Apache kafka

Apache kafka là hệ thống truyền tải thông điệp được xây dựng trên hệ thống có thể mở rộng dễ dàng cho big data.
Kafka khác so với các hệ thống truyền tin truyền thống ở những điểm sau:
– Được thiết kế để scale horizontally, bằng cách thêm vào nhiều hơn các server thông thường
– Cung cấp khả năng throughput cao cho cả producer and customer
– Hỗ trợ cả bath và real-time use case
– Không hỗ trợ JMS (Java mesage Service Concepts), Java's mesage-oriented middleware API.



kafka's architecture .
Trước khi ta tìm hiểu kiến trúc của kafka, ta cần biết các thuật ngữ (terminology) sau đây:

– A producer là process có thể publish a message to topic
– A consumer is a process that can subscribe to one or more topics and consume messages published to topics.
– A topic category is the name of the feed to which messages are published.
– A broker is the process running on single machine.
– A cluster is a group of brokers working together.

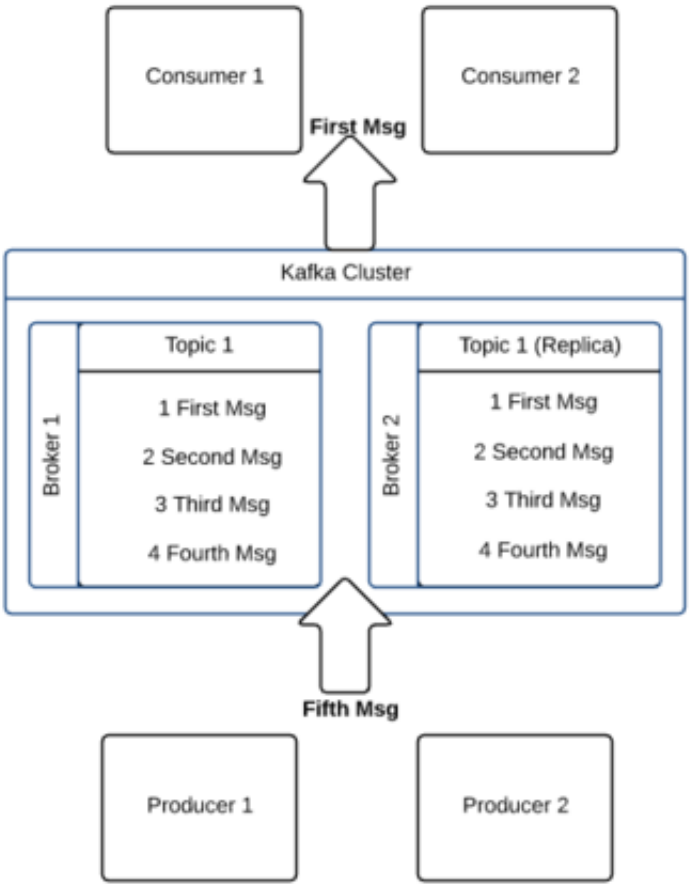


Figure 1. Architecture of a Kafka message system.

Kafka's architecture is very simple, which can result in better performance and throughput in some systems.
Every topic in Kafka is like a simple log file. When a producer publishes a message, the Kafka server appends it to the end of the log file for its given topic.

The Server also assigns an offset, which is a number used to permanently identify each message. As the number of messages grows, the value of each offset increases; for example if the producer publishes three message the first one might get an offset of 1, the second an offset of 2, and the thrid an offset of 3.

When the Kafka consumer first starts, it will send a pull request to the server, asking to retrieve any messanges for a particular topic with an offset value higher than 0. The server will check the log file for that topic and return three new massages. The consumer will process the messages, then send a request for messages with an offset higher than 3, so on.

In Kafka, the client is reponsible for remmberiing the the offset count aand retieving message. The Kafka server doesn't track or manage message consumption. By default, a Kafka server will keep message for seven days. A backgroud thread in the server checks and deletes messages that are seven day or older. A consumer can access messages as long as they are on the server. It can read a message multiple times, and even read messages in reverse order od receipt. But if the consumer fails to retrieve the message before the seven days are up, it will miss that message.

Quick setup and demo We'll build a custom application in this tutorial, but let's start by installing and testing a Kafka instance with an out-of-the-box producer and consumer.

1. Visit the Kafka download page to install the most recent version.
2. Extract the binaries into a software/kafka folder.
3. Change your current directory to point to the new folder.
4. Start the Zookeeper server by executing the command: bin/zookeeper-server-start.sh config/zookeeper.properties.
5. Start the Kafka server by executing: bin/kafka-server-start.sh config/server.properties.
6. Create a test topic that you can use for testing: bin/kafka-topics.sh –create –zookeeper localhost:2181 –replication-factor 1 –partitions 1 –topic javaworld.
7. Start a simple console consumer that can consume messages published to a given topic, such as javaworld: bin/kafka-console-consumer.sh –zookeeper localhost:2181 –topic javaworld –from-beginning.
8. Start up a simple producer console that can publish messages to the test topic: bin/kafka-console-producer.sh –broker-list localhost:9092 –topic javaworld.
9. Try typing one or two messages into the producer console. Your messages should show in the consumer console.

3.2 Apache Storm

What is Apache Storm ?

Apache Storm is a distributed real-time computation system for processing large volumes of high-velocity data.
Apache storm is extremely fast, with the ability to process over a million records per second per node on a cluster of modest size.
Enterprises harness this speed and combine it with other data access applications in Hadoop to prevent undesirable events or to optimize positive outcomes.

Feature of Apache Storm

1. Storm is simple and developers can write Storm topologies using any programming language.
2. A Storm topology consists of “Spouts” and “Bolts”.
3. The “data” in a Storm topology is called a “Tuple”.

Characteristic of Apache Storm

Five characteristics make Storm ideal for real-time data processing workloads is :

1. **Fast** – benchmarked as processing one million 100 byte messages per second per node
2. **Scalable** – with parallel calculations that run across a cluster of machines
3. **Fault-tolerant** – when workers die, Storm will automatically restart them. If a node dies, the worker will be restarted on another node.
4. **Reliable** – Storm guarantees that each unit of data (tuple) will be processed at least once or exactly once. Messages are only replayed when there are failures.
5. **Easy to operate** – standard configurations are suitable for production on day one. Once deployed, Storm is easy to operate

Quick setup and demo

In sumary, the steps are:

- Download a Storm release, unpack it, and put the unpacked bin/ directory on your PATH.
- To be able to start and stop topologies on a remote cluster, put the cluster information in /.storm/storm.yaml.
- A Storm development environment has everything installed so that you can develop and test Storm topologies in local mode, package topologies for execution on a remote cluster, and submit/kill topologies on a remote cluster.

Abstractions in Storm: spouts, bolts, and topologies
There are just three abstractions in Storm: spouts, bolts, and topologies.

1. **A spout** is a source of streams in a computation. Typically a spout reads from a queueing broker such as Kestrel, RabbitMQ, or Kafka, but a spout can also generate its own stream or read from somewhere like the Twitter streaming API. Spout implementations already exist for most queueing systems.

2. **A bolt** processes any number of input streams and produces any number of new output streams. Most of the logic of a computation goes into bolts, such as functions, filters, streaming joins, streaming aggregations, talking to databases, and so on.

3. **A topology** is a network of spouts and bolts, with each edge in the network representing a bolt subscribing to the output stream of some other spout or bolt. A topology is an arbitrarily complex multi-stage stream computation. Topologies run indefinitely when deployed.

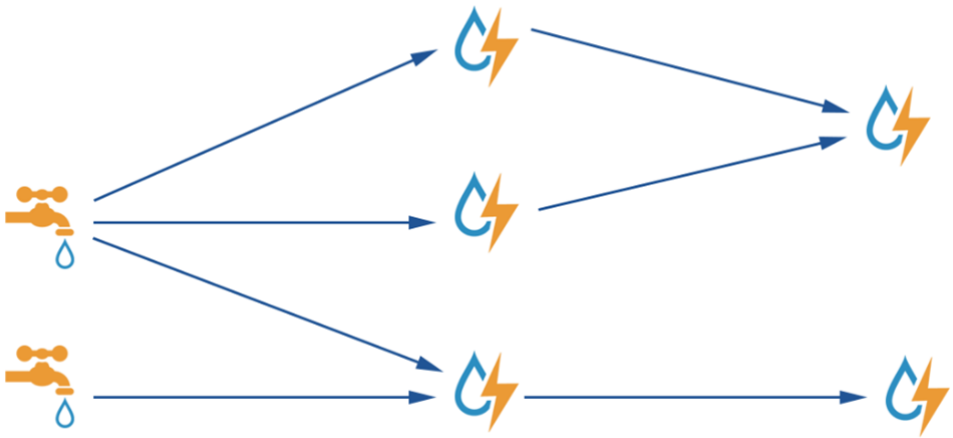


Figure 2. Three abstractions in Storm: spouts, bolts, and topologies..

Core Storm Concepts .
Developing a Storm application requires an understanding of the following basic concepts.
Table 4.1. Storm Concepts

Storm Concept	Descripition
Tuple	A named list of values of any data type. A Tuple is the native data structure used by storm.
Stream	An bounded sequence of tuples.
Spout	Generates a stream from a realtime data source.
Bolt	Contains data processing, persistence, and messaging alert logic. Can also emit tuples for downstream bolts.
Topology	A group of spouts and bolts wired together into a wordflow. A Storm application.
Processing Reliability	Storm guarantee about the delivery of tuples in a topology.
Workers	A storm process. A worker may run one oe more executors.
Executor	A storm thread launched by Storm worker. A executor may run one or more task.
Tasks	A Storm job from a spout or bolt.
Parallelism	Attribute of distributed data oricessing that determines how many jobs are processed simultaneously for a topology. Topology developers adjust parallelism to tune their applications.
Process Controller	Monotors and restarts failed storm processes. Examples include supervisord, monit, and daemontools.
Master/Numbus Node	The host isn a multi-node Storm cluster that runs a process controller (such as supervisord) and the Storm numbus, ui, and other related daemons. The process controller is responsible for restarting failed process controller deamons on slave node. The Nimbus node is a thrift service that is responsible for distributing code around the cluster, addigning tasks to machines, and monitoring for failures.
Slave Node	A host in a multi-node Storm cluster that runs a process controller daemon, such as supervisor, as well as the worker processes that run Storm topologies. The process controller daemon is responsible for restaring failed worker processes.

Spouts

All spouts must implement the org.apache.storm.topology.IRichSpout interface from the core-storm API. BaseRichSpout is the most basic implementation, but there are several others, including ClojureSpout, DRPCSpout, and FeederSpout. In addition, Hortonworks provides a Kafka spout to ingest data from a Kafka cluster. The following example, RandomSentenceSpout, is included with the storm-starter connector installed with Storm at /usr/lib/storm/contrib/storm-starter.

```
package storm.starter.spout;
        import org.apache.storm.spout.SpoutOutputCollector;
        import org.apache.storm.task.TopologyContext;
        import org.apache.storm.topology.OutputFieldsDeclarer;
        import org.apache.storm.topology.base.BaseRichSpout;
        import org.apache.storm.tuple.Fields;
        import org.apache.storm.tuple.Values;
        import org.apache.storm.utils.Utils;
        import java.util.Map;
        import java.util.Random;

public class RandomSentenceSpout extends BaseRichSpout {
    SpoutOutputCollector _collector;
    Random _rand;

    @Override
    public void open(Map conf, TopologyContext context, SpoutOutputCollector
            collector) {
```

```
        _collector = collector;
        _rand = new Random();
    }

    @Override
    public void nextTuple() {
        Utils.sleep(100);
        String[] sentences = new String[]{"the cow jumped over the moon", "an
                apple a day keeps the doctor away", " four score and seven years ago", " snow
                white and the seven dwarfs", " i am at two with nature" };
                String sentence = sentences[_rand.nextInt(sentences.length)];
        _collector.emit(new Values(sentence));
        }
        @Override
        public void ack (Object id){
        }
        @Override
        public void fail (Object id){
        }
        @Override
        public void declareOutputFields (OutputFieldsDeclarer declarer){
            declarer.declare(new Fields("word"));
        }
    }
```

Bolts

All bolts must implement the IRichBolt interface. BaseRichBolt is the most basic implementation, but there are several others, including BatchBoltExecutor, ClojureBolt, and JoinResult. The following example, TotalRankingsBolt.java, is included with storm-starter and installed with Storm at /usr/lib/storm/contrib/storm-starter.

```
package storm.starter.bolt;
import org.apache.storm.tuple.Tuple;
import org.apache.log4j.Logger;
import storm.starter.tools.Rankings;
/**
* This bolt merges incoming {@link Rankings}.
* <p/>
* It can be used to merge intermediate rankings generated by {@link
IntermediateRankingsBolt} into a final,
* consolidated ranking. To do so, configure this bolt with a globalGrouping
on {@link IntermediateRankingsBolt}.
*/
public final class TotalRankingsBolt extends AbstractRankerBolt {
private static final long serialVersionUID = -8447525895532302198L;
private static final Logger LOG = Logger.getLogger(TotalRankingsBolt.class);
public TotalRankingsBolt() {
super();
}
public TotalRankingsBolt(int topN) {
super(topN);
}
public TotalRankingsBolt(int topN, int emitFrequencyInSeconds) {
Hortonworks Data Platform December 15, 2017
17
super(topN, emitFrequencyInSeconds);
}
@Override
```

```
void updateRankingsWithTuple(Tuple tuple) {
Rankings rankingsToBeMerged = (Rankings) tuple.getValue(0);
super.getRankings().updateWith(rankingsToBeMerged);
super.getRankings().pruneZeroCounts();
}
@Override
Logger getLogger() {
return LOG;
}
}
```

Processing Reliability

Storm provides two types of grarantees when processing tuples for a Storm topology.

Table 4.3. Processing Guarantees

Guarantee	Description
At least once	Reliable; Tuples are processed at least once, but may be processes more than once. use when subsecond latency is required and for unordered idempotent operations.
Exactly once	Reliable; Tuples are processed only once.(this feature requires the use of a Trident spout and the Trident API. For more information, see Trident concepts).

Stream Grouping

Stream grouping allows Storm developers to control how tuples are routed to bolts in a workflow. The following table describes the stream groupings available.

Table 4.2. Stream Groupings

Stream Grouping	Description
Shuffle	Sends tuples to bolts in random, round robin sequence. Use for atomic operations, such as math.
Fields	Sends tuples to a bolt based on one or more fields in the tuple. Use to segment an incoming stream and to count tuples of a specified type.
All	Sends a single copy of each tuple to all instances of a receiving bolt. Use to send a signal, such as clear cache or refresh state, to all bolts.
Custom	Customized processing sequence. Use to get maximum flexibility of topology processing based on factors such as data types, load, and seasonality.
Direct	Source decides which bolt receives a tuple. Global Sends tuples generated by all instances of a source to a single target instance. Use for global counting operations.

Storm developers specify the field grouping for each bolt using methods on the TopologyBuilder.BoltGetter inner class, as shown in the following excerpt from the the WordCountTopology.java example included with storm-starter. ...

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout", new RandomSentenceSpout(), 5);
builder.setBolt("split", new SplitSentence(), 8).shuffleGrouping("spout");
builder.setBolt("count", new WordCount(), 12).fieldsGrouping("split", new
Fields("word"));
```

The first bolt uses shuffle grouping to split random sentences generated with the RandomSentence-Spout. The second bolt uses fields grouping to segment and perform a count of individual words in the sentences.

Topologies

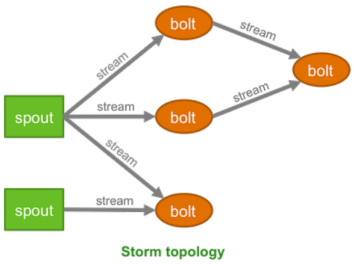


Figure 3. Storm Topology.

The TopologyBuilder class is the starting point for quickly writing Storm topologies with the storm-core API. The class contains getter and setter methods for the spouts and bolts that comprise the streaming data workflow, as shown in the following sample code.

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("spout1", new BaseRichSpout());
builder.setSpout("spout2", new BaseRichSpout());
builder.setBolt("bolt1", new BaseBasicBolt());
builder.setBolt("bolt2", new BaseBasicBolt());
builder.setBolt("bolt3", new BaseBasicBolt());
```

Workers, Executors, and tasks

Apache Storm processes, called workers, run on predefined ports on the machine that hosts Storm.
* Each worker process can run one or more executors, or threads, or threads, where each executor is a thread spawned by the worker process.
* Each executor runs one or more tasks from the same component, where a component is a spout or bolt from a topology.
Here is a simple illustration of their relationships:

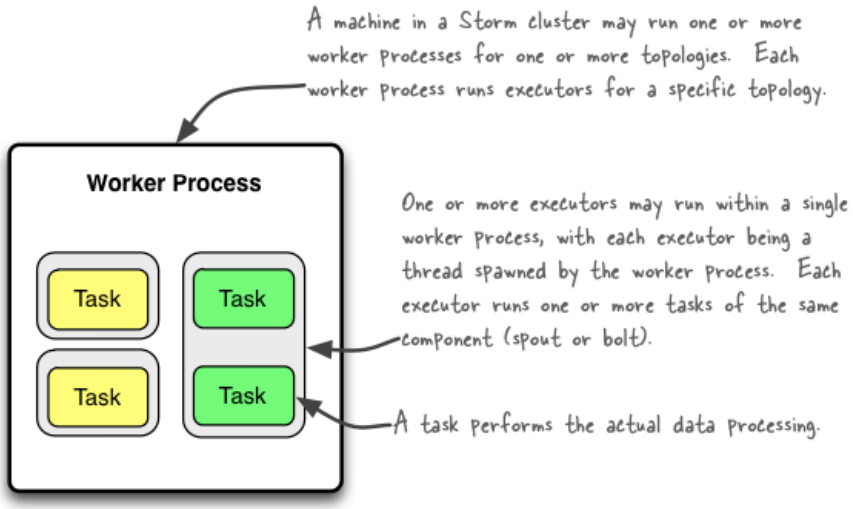


Figure 4. What makes a running topology: worker processes, executors and tasks.

Parallelism

Distributed applications take advantage of horizontally-scaled clusters by dividing computation tasks across nodes in a cluster. Storm offers this and additional finer-grained ways to increase the parallelism of a Storm topology:

- Inscease the number of work
- Increase the number of executors
- Increase the number of task

Here is the relevant code:

```
Config conf = new Config();
conf.setNumWorkers(2);
//use two worker processes

topologyBuilder.setSpout("blue-spout", new BlueSpout(), 2);
// set parallelism hint to 2

topologyBuilder.setBolt("green-bolt", new GreenBolt(), 2)
               .setNumTasks(4)
               .shuffleGrouping("blue-spout");

topologyBuilder.setBolt("yellow-bolt", new YellowBolt(), 6)
               .shuffleGrouping("green-bolt");

StormSubmitter.submitTopology(
        "mytopology",
        conf,
        topologyBuilder.createTopology()
    );
```

4 Apache Storm integration with Apache Kafka

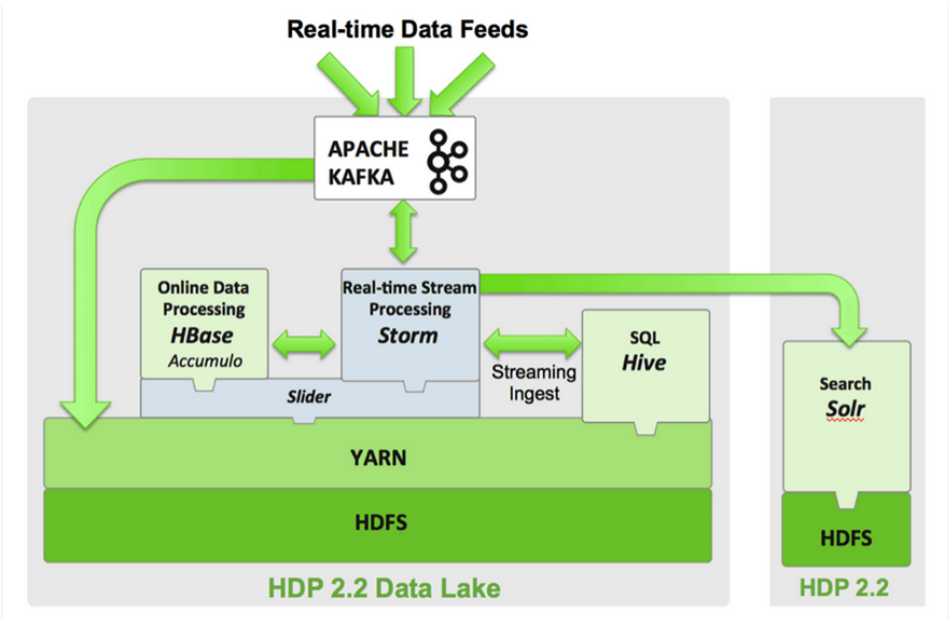


Figure 3. Apache Storm integration with Apache Kafka (1).

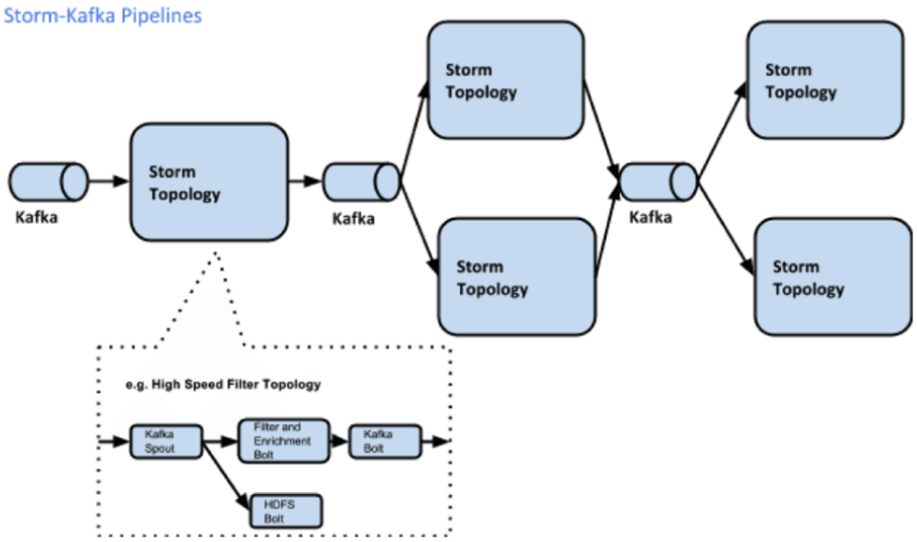


Figure 4. Apache Storm integration with Apache Kafka (2).

Here is Code

```
   public class KafkaSpoutTopology
{
// ------------------------------ FIELDS ------------------------------

    private final static String KAFKA_HOST = "localhost:2181";
    private final static String NIMBUS_HOST = "192.168.1.152";
    private final static String KAFKA_TOPIC = "storm-test-topic";

    private final BrokerHosts brokerHosts;

// -------------------------- CONSTRUCTORS --------------------------

    private KafkaSpoutTopology(String kafkaZookeeper)
    {
        brokerHosts = new ZkHosts(kafkaZookeeper);
    }

// -------------------------- main() method --------------------------

    public static void main(String[] args) throws Exception
    {
        KafkaSpoutTopology kafkaSpoutTopology = new KafkaSpoutTopology(KAFKA_HOST);
        Config config = new Config();
//        config.put(Config.TOPOLOGY_TRIDENT_BATCH_EMIT_INTERVAL_MILLIS, 2000);
        config.put(Config.TOPOLOGY_MESSAGE_TIMEOUT_SECS, 30);

        StormTopology stormTopology = kafkaSpoutTopology.buildTopology();
        if (args != null && args.length > 0)
        {
            String name = args[0];
            config.put(Config.NIMBUS_HOST, NIMBUS_HOST); //YOUR NIMBUS'S IP
            config.put(Config.NIMBUS_THRIFT_PORT, 6627); //int is expected here
            config.setNumWorkers(20);
            config.setMaxSpoutPending(5000);
```

```
            StormSubmitter.submitTopology(name, config, stormTopology);
        }
        else
        {
            config.setNumWorkers(2);
            config.setMaxTaskParallelism(Runtime.getRuntime().availableProcessors());
            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("kafka-storm-cassandra", config, stormTopology);
        }
    }

    private StormTopology buildTopology()
    {
        SpoutConfig kafkaConfig = new SpoutConfig(brokerHosts, KAFKA_TOPIC, "/tmp/kafka-logs",
            "storm-integration");
        kafkaConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
//        kafkaConfig.startOffsetTime = System.currentTimeMillis();
        TopologyBuilder builder = new TopologyBuilder();
        builder.setSpout("words", new KafkaSpout(kafkaConfig), 10);
        builder.setBolt("write-to-cassandra", new
            CassandraWriterBolt()).shuffleGrouping("words");
//        builder.setBolt("notify", new NotifyBolt()).shuffleGrouping("write-to-cassandra");
//        builder.setBolt("write-to-elasticsearch", new
    ElasticSearchBolt()).shuffleGrouping("write-to-cassandra");
        return builder.createTopology();
    }
}
```

```
class CassandraWriterBolt extends BaseRichBolt
{

    private static final Logger LOG = LoggerFactory.getLogger(CassandraWriterBolt.class);

    private final static String USERNAME = "vndev";
    private final static String PASSWORD = "123";
    private final static String HOST = "192.168.1.158";
    private final static String KEYSPACE = "dev";

    private Session session;
    private OutputCollector collector;

    @Override
    public void prepare(Map map, TopologyContext topologyContext, OutputCollector
        outputCollector)
    {
        this.collector = outputCollector;
    }

    @Override
    public void execute(Tuple tuple)
    {
        try
        {
            CassandraConnection connection = new CassandraConnection(HOST, KEYSPACE, USERNAME,
                PASSWORD);
            session = connection.getSession();
            LOG.info("content " + tuple.getString(0));
            boundCQLStatement(tuple, null);
            connection.close();
```

```
        }
        catch (Throwable t)
        {
            collector.reportError(t);
            collector.fail(tuple);
            LOG.error("tuple data error " + t.toString());
        }
    }

// ---------------------- Interface IComponent ---------------------


    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer)
    {
        declarer.declare(new Fields("name", "post_id", "author_id", "content", "channel_ids",
            "published_time"));
    }

// -------------------------- OTHER METHODS -------------------------

    private void boundCQLStatement(Tuple input, UserActivity userActivity)
    {
        PreparedStatement statement = session.prepare(
                "INSERT INTO wordcounttable (source,word,count) VALUES (?, ?, ?);");
        BoundStatement boundStatement = new BoundStatement(statement);
        session.execute(boundStatement.bind(
                input.getString(0), "year", 1
        ));

    }
}
```

5 Kết Luận

Kafka có thể sử dụng làm "Spout" cho Apache Storm
Apache Storm có thể sử dụng các API của Trident (filter và function) để lọc và "emit" (phát ra) các Tuple mới cho Stream.
Một máy Storm cluster có thể có một hay nhiều Worker cho một hay nhiều topologies. Mỗi tiến trình worker thực thi executor cho riêng một topology.
với một tiến trình worker, có thể có một hay nhiều executor có thể thực thi, với mỗi executor trở thành thread được sinh ra bởi worker process. Mỗi executor có thể thực thi một hay nhiều tasks ó the same componemt(spout and bolt)
Mỗi task biểu thị quá trình xử lý dữ liệu thực tế...

6 Tài liệu Tham Khảo

1. https://www.javaworld.com/article/3060078/big-data/big-data-messaging-with-kafka-part-1.html
2. http://storm.apache.org