Hortonworks Data Platform

Using Apache Storm

(September 30, 2015)

docs.cloudera.com

Hortonworks Data Platform: Using Apache Storm

Copyright © 2012-2015 Hortonworks, Inc. Some rights reserved.

The Hortonworks Data Platform, powered by Apache Hadoop, is a massively scalable and 100% open source platform for storing, processing and analyzing large volumes of data. It is designed to deal with data from many sources and formats in a very quick, easy and cost-effective manner. The Hortonworks Data Platform consists of the essential set of Apache Hadoop projects including MapReduce, Hadoop Distributed File System (HDFS), HCatalog, Pig, Hive, HBase, ZooKeeper and Ambari. Hortonworks is the major contributor of code and patches to many of these projects. These projects have been integrated and tested as part of the Hortonworks Data Platform release process and installation and configuration tools have also been included.

Unlike other providers of platforms built using Apache Hadoop, Hortonworks contributes 100% of our code back to the Apache Software Foundation. The Hortonworks Data Platform is Apache-licensed and completely open source. We sell only expert technical support, training and partner-enablement services. All of our technology is, and will remain, free and open source.

Please visit the Hortonworks Data Platform page for more information on Hortonworks technology. For more information on Hortonworks services, please visit either the Support or Training page. Feel free to contact us directly to discuss your specific needs.

Except where otherwise noted, this document is licensed under
Creative Commons Attribution ShareAlike 4.0 License.
http://creativecommons.org/licenses/by-sa/4.0/legalcode

Table of Contents

List of Tables

1. Using Apache Storm

The exponential increase in data from real-time sources such as machine sensors creates a need for data processing systems that can ingest this data, process it, and respond in real time. A typical use case involves an automated system that responds to sensor data by sending email to support staff or placing an advertisement on a consumer's smart phone. Apache Storm enables such data-driven and automated activity by providing a realtime, scalable, and distributed solution for streaming data.

Apache Storm can be used with any programming language, and guarantees that data streams are processed without data loss.

Storm is datatype-agnostic; it processes data streams of any data type.

A complete introduction to the Storm API is beyond the scope of this documentation. However, the next section, Basic Storm Concepts, provides a brief overview of the most essential concepts and a link to the javadoc API. For a more thorough discussion of Apache Storm concepts, see the Apache Storm documentation for your version of Storm.

Experienced Storm developers may want to skip to later sections for information about streaming data to Hive; ingesting data with the Apache Kafka spout; writing data to HDFS, HBase, and Kafka; and deploying Storm topologies.

The last section, RollingTopWords Topology, lists the source code for a sample application included with the `storm-starter.jar`.

1.1. Basic Storm Concepts

Writing Storm applications requires an understanding of the following basic concepts.

Table 1.1. Storm Concepts

Storm Concept	Description
Tuple	A named list of values of any data type. The native data structure used by Storm.
Stream	An unbounded 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.
Stream Grouping	Controls the routing of tuples to bolts for processing.
Topology	A group of spouts and bolts wired together into a workflow. A Storm application.
Processing Reliability	Storm guarantee about the delivery of tuples in a topology.
Parallelism	Attribute of distributed data processing that determines how many jobs are processed simultaneously for a topology. Topology developers adjust parallelism to tune their applications.
Workers	A Storm process. A worker may run one or more executors.
Executors	A Storm thread launched by a Storm worker. An executor may run one or more tasks.
Tasks	A Storm job from a spout or bolt.

Storm Concept	Description
Process Controller	Monitors and restarts failed Storm processes. Examples include supervisord, monit, and daemontools.
Master/Nimbus Node	The host in a multi-node Storm cluster that runs a process controller, such as supervisord, and the Storm nimbus, ui, and other related daemons. The process controller is responsible for restarting failed process controller daemons, such as supervisor, on slave nodes. The Storm nimbus daemon is responsible for monitoring the Storm cluster and assigning tasks to slave nodes for execution.
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 restarting failed worker processes.

The following subsections describe several of these concepts in more detail.

1.1.1. Spouts

All spouts must implement the `backtype.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 backtype.storm.spout.SpoutOutputCollector;
import backtype.storm.task.TopologyContext;
import backtype.storm.topology.OutputFieldsDeclarer;
import backtype.storm.topology.base.BaseRichSpout;
import backtype.storm.tuple.Fields;
import backtype.storm.tuple.Values;
import backtype.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"));
}

}
```

1.1.2. 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 backtype.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) {
 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;
 }

}
```

1.1.3. Stream Groupings

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 1.2. 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.

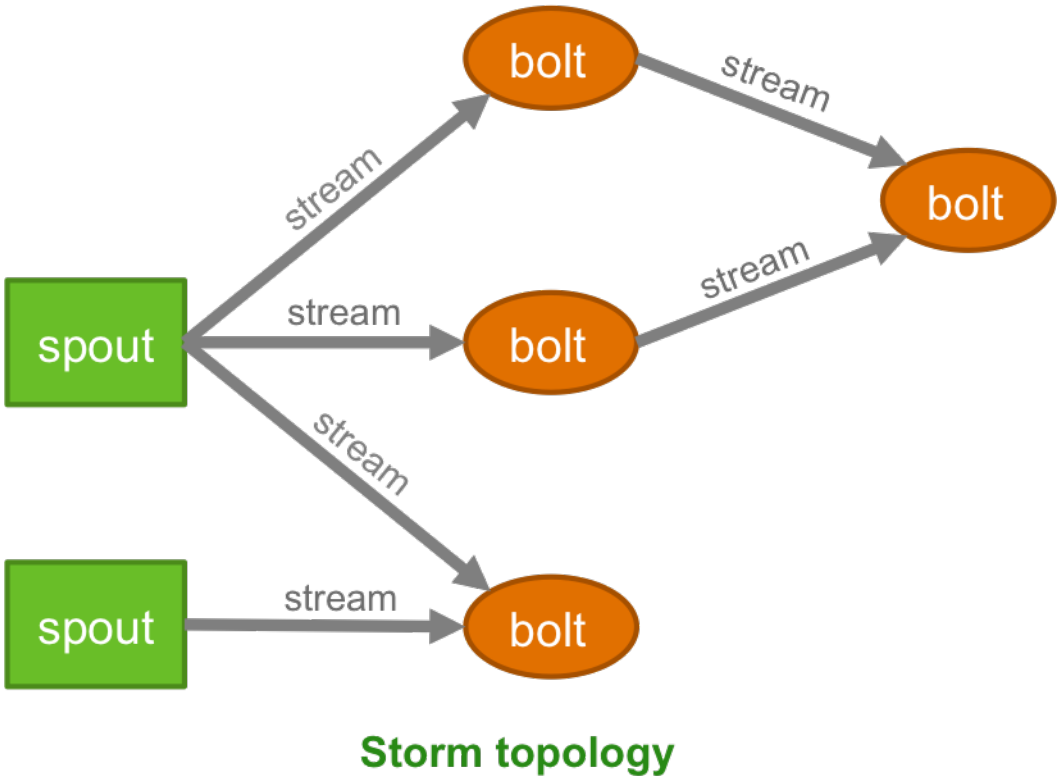
Storm developers specify the field grouping for each bolt using methods on the `TopologyBuilder.BoltGetter` inner class, as shown in the following excerpt from the the `WordCountTopology.java` example included with `storm-starter`.

```
...
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 `RandomSentenceSpout`. The second bolt uses fields grouping to segment and perform a count of individual words in the sentences.

1.1.4. Topologies

The following image depicts a Storm topology with a simple workflow.



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());
...
```

1.1.5. Processing Reliability

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

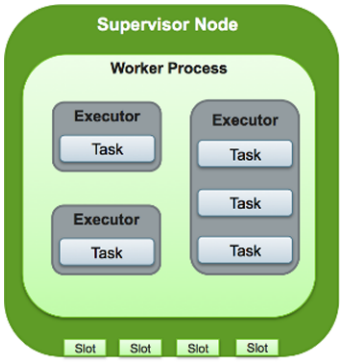
Table 1.3. Processing Guarantees

Guarantee	Description
At least once	Reliable; Tuples are processed at least once, but may be processed more than once. Use when subsecond latency is required and for unordered idempotent operations.
Exactly once	Reliable; Tuples are processed only once. Requires the use of a Trident spout and the Trident API.

1.1.6. 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, 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.



1.1.7. Parallelism

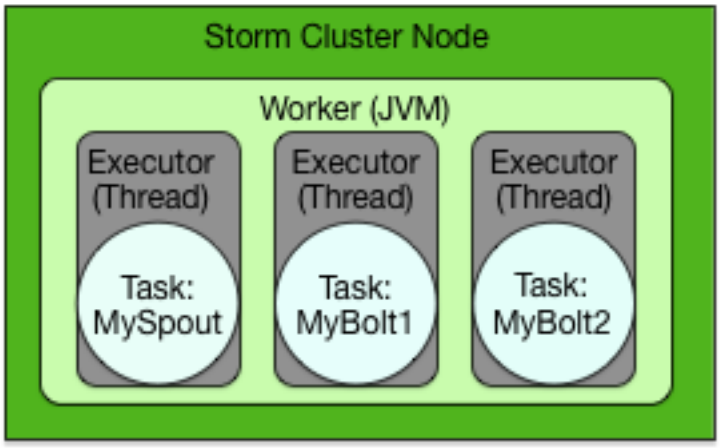
Distributed applications take advantage of horizontally-scaled clusters by dividing computation tasks across nodes in a cluster. Storm offers this and additional finer-grained ways to increase the parallelism of a Storm topology:

- Increase the number of workers
- Increase the number of executors
- Increase the number of tasks

By default, Storm uses a parallelism factor of 1. Assuming a single-node Storm cluster, a parallelism factor of 1 means that one worker, or JVM, is assigned to execute the topology, and each component in the topology is assigned to a single executor. The following diagram illustrates this scenario. The topology defines a data flow with three tasks, a spout and two bolts.

Note

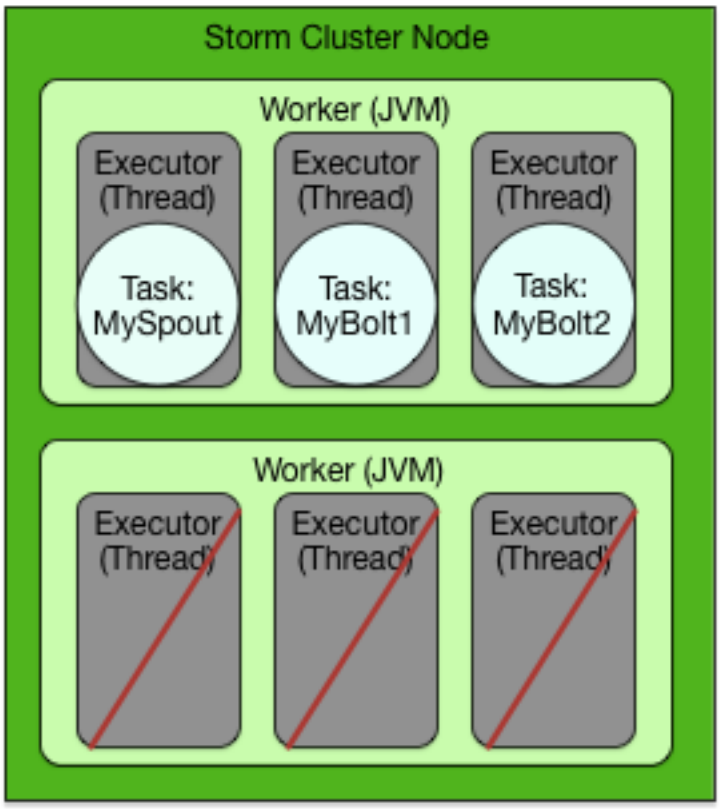
Hortonworks recommends that Storm developers store parallelism settings in a configuration file read by the topology at runtime rather than hard-coding the values passed to the Parallelism API. This topic describes and illustrates the use of the API, but developers can achieve the same effect by reading the parallelism values from a configuration file.



Increasing Parallelism with Workers

Storm developers can easily increase the number of workers assigned to execute a topology with the `Config.setNumWorkers()` method. This code assigns two workers to execute the topology, as the following figure illustrates.

```
...
Config config = new Config();
config.setNumWorkers(2);
...
```



Adding new workers comes at a cost: additional overhead for a new JVM.

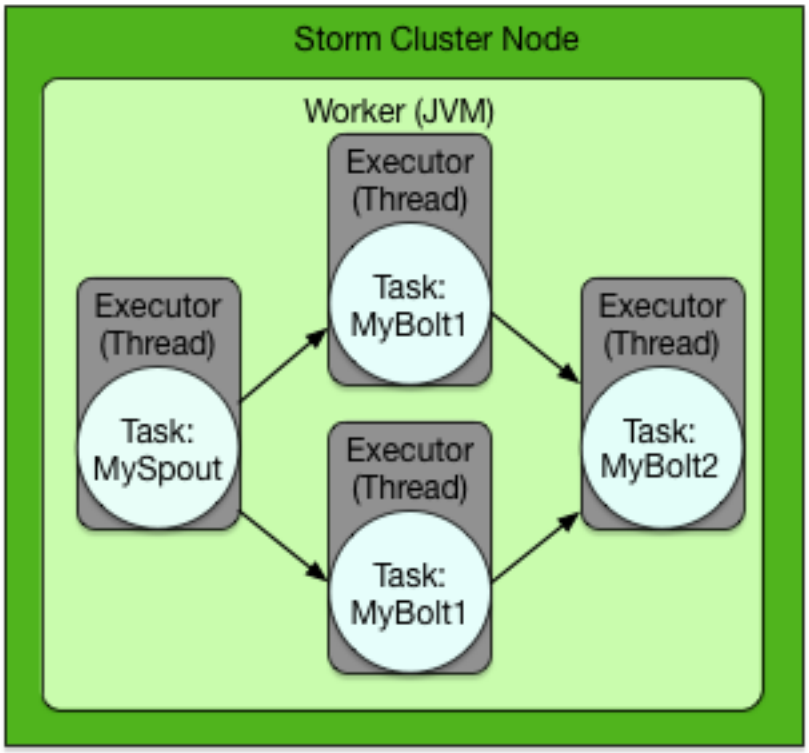
This example adds an additional worker without additional executors or tasks, but to take full advantage of this feature, Storm developers must add executors and tasks to the additional JVMs (described in the following examples).

Increasing Parallelism with Executors

The parallelism API enables Storm developers to specify the number of executors for each worker with a parallelism hint, an optional third parameter to the `setBolt()` method. The following code sample sets this parameter for the MyBolt1 topology component.

```
...
Config config = new Config();
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(MY_SPOUT_ID, mySpout);
builder.setBolt(MY_BOLT1_ID, myBolt1, 2).shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT2_ID, myBolt2).shuffleGrouping(MY_SPOUT_ID);
...
```

This code sample assigns two executors to the single, default worker for the specified topology component, MyBolt1, as the following figure illustrates.



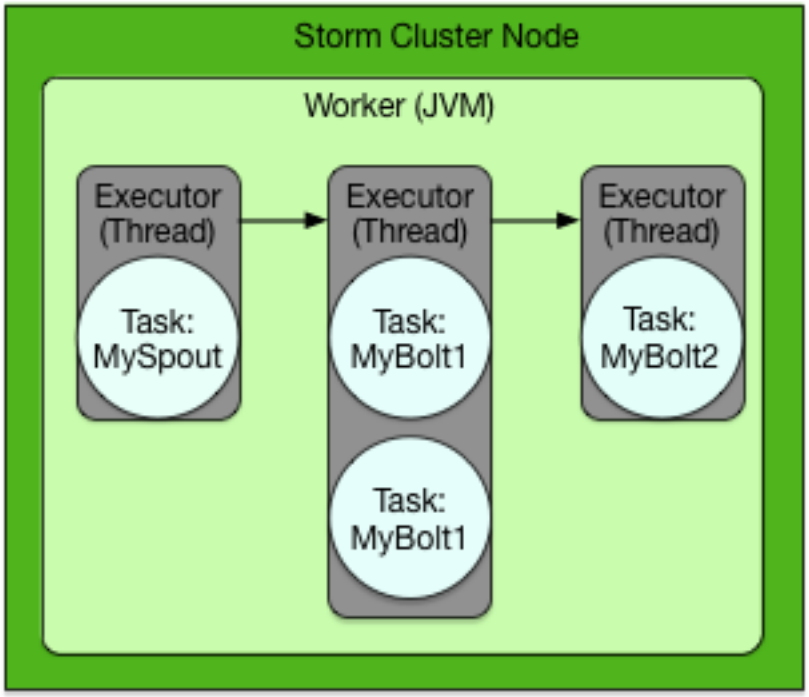
The number of executors is set at the level of individual topology components, so adding executors affects the code for the specified spouts and bolts. This differs from adding workers, which affects only the configuration of the topology.

Increasing Parallelism with Tasks

Finally, Storm developers can increase the number of tasks assigned to a single topology component, such as a spout or bolt. By default, Storm assigns a single task to each component, but developers can increase this number with the `setNumTasks()` method on the `BoltDeclarer` and `SpoutDeclarer` objects returned by the `setBolt()` and `setSpout()` methods.

```
...
Config config = new Config();
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(MY_SPOUT_ID, mySpout);
builder.setBolt(MY_BOLT1_ID, myBolt1).setNumTasks(2).
shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT1_ID, myBolt2).shuffleGrouping(MY_SPOUT_ID);
...
```

This code sample assigns two tasks to execute MyBolt1, as the following figure illustrates. This added parallelism might be appropriate for a bolt containing a large amount of data processing logic. However, adding tasks is like adding executors because the code for the corresponding spouts or bolts also changes.



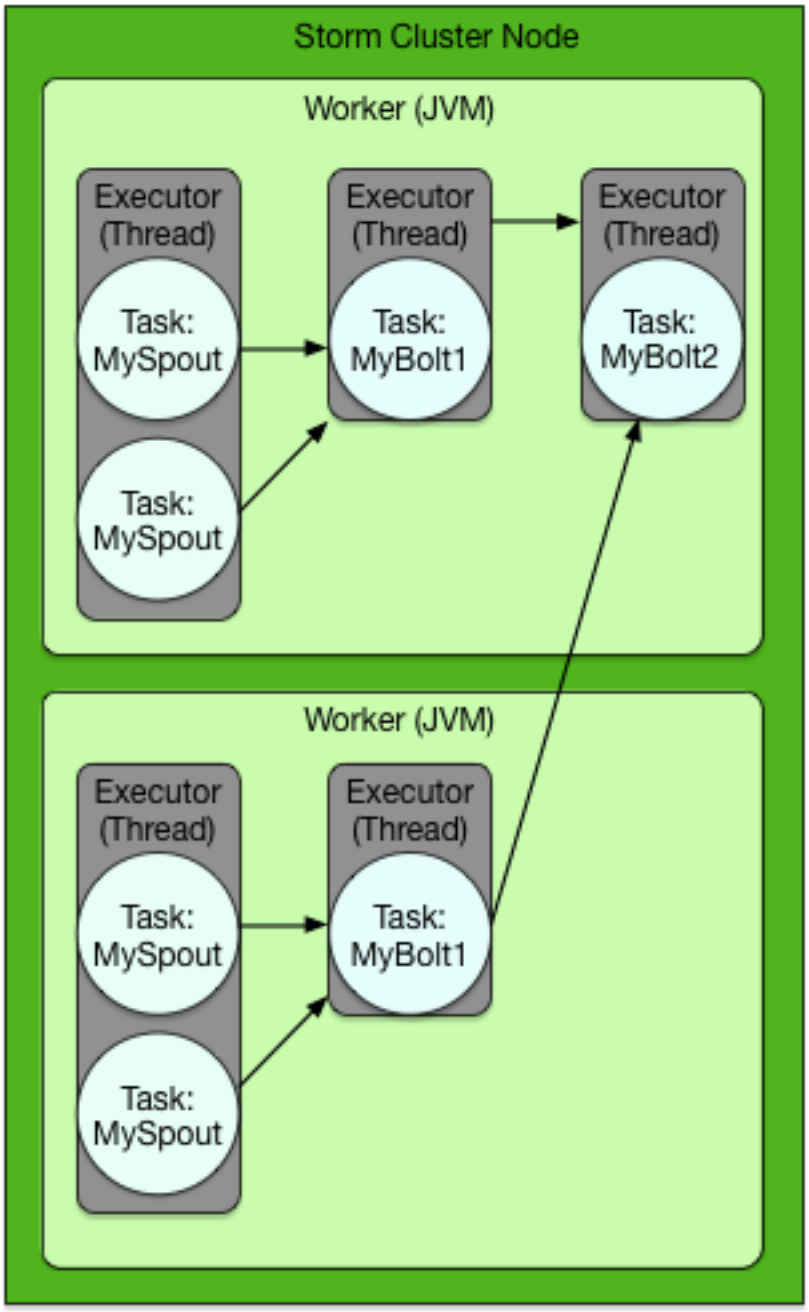
Putting it All Together

Storm developers can fine-tune the parallelism of their topologies by combining new workers, executors and tasks. The following code sample demonstrates all of the following:

- Split processing of the MySpout component between four tasks in two separate executors across two workers
- Split processing of the MyBolt1 component between two executors across two workers

- Centralize processing of the MyBolt2 component in a single task in a single executor in a single worker on a single worker

```
...
Config config = new Config();
config.setNumWorkers(2);
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout(MY_SPOUT_ID, mySpout, 2).setNumTasks(4);
builder.setBolt(MY_BOLT1_ID, myBolt1, 2).setNumTasks(2).
shuffleGrouping(MY_SPOUT_ID);
builder.setBolt(MY_BOLT2_ID, myBolt2).shuffleGrouping(MY_SPOUT_ID);
...
```



The degree of parallelism depicted might be appropriate for the following topology requirements:

- High-volume streaming data input
- Moderate data processing logic
- Low-volume topology output

2. Installing and Configuring Storm

To install Storm using Ambari, see Adding a Service to your Hadoop cluster in the *Ambari User's Guide*. To configure Storm for Kerberos security on an Ambari-managed cluster, see Configuring Storm for Kerberos Over Ambari.

To install Storm manually, see Installing and Configuring Apache Storm in the *Non-Ambari Cluster Installation Guide*.

If you are deploying a production cluster with Storm, you should configure the Storm components to operate under supervision. For more information, see Configuring Storm for Supervision in the *Ambari Reference Guide*. (The link is for Ambari version 2.2.2.0.)

3. Topology Development Guidelines

Hortonworks recommends the following guidelines for all Storm topologies.

Note

These recommendations focus on guidelines for writing and debugging Storm topologies, rather than hardware tuning. Typically, most of the computation burden falls on the Supervisor and Worker nodes in a Storm cluster. The Nimbus node usually has a lighter load. For this reason, Hortonworks recommends that organizations save their hardware resources for the relatively burdened Supervisor and Worker nodes.

Table 3.1. Storm Topology Guidelines

Guideline	Description
Read topology configuration parameters from a file.	Rather than hard coding configuration information in your Storm application, read the configuration parameters, including parallelism hints for specific components, from a file inside the `main()` method of the topology. This speeds up the iterative process of debugging by eliminating the need to rewrite and recompile code for simple configuration changes.
Use a cache.	Use a cache to improve performance by eliminating unnecessary operations over the network, such as making frequent external service or lookup calls for reference data needed for processing.
Tighten code in the `execute()` method.	Every tuple is processed by the `execute()` method, so verify that the code in this method is as tight and efficient as possible.
Perform benchmark testing to determine latencies.	Perform benchmark testing of the critical points in the network flow of your topology. Knowing the capacity of your data "pipes" provides a reliable standard for judging the performance of your topology and its individual components.

3.1. Debugging Topologies

This topic describes best practices for debugging Storm topologies, including basic guidelines for configuring parallelism for individual topology components.

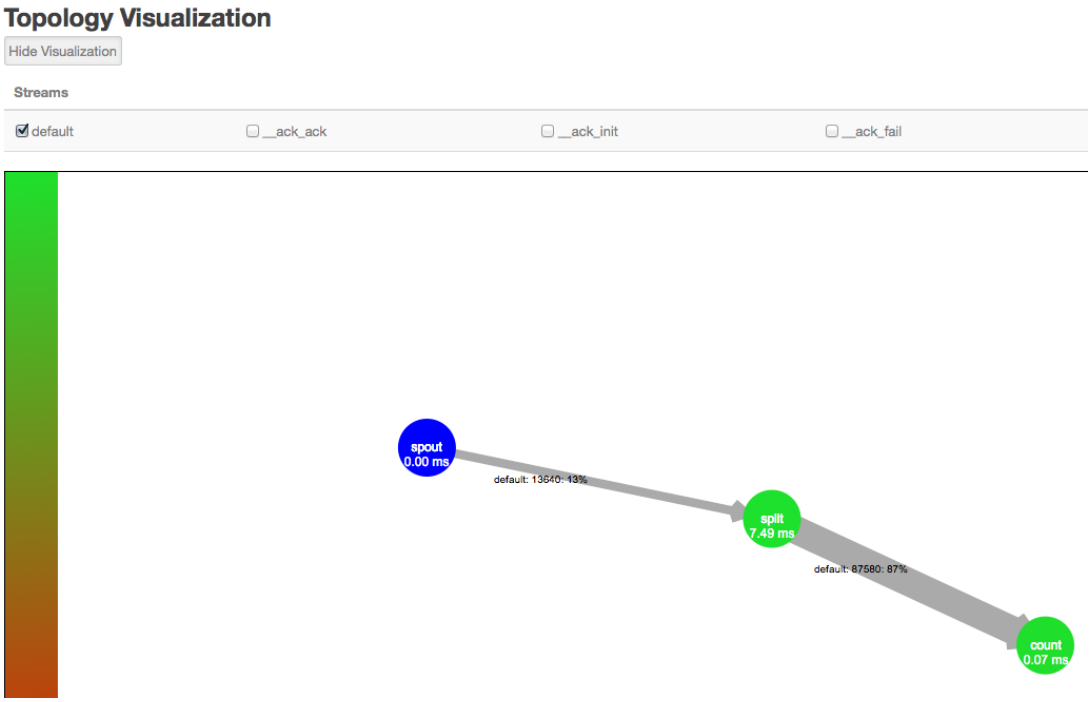
Debugging Storm topologies differs from debugging batch-oriented applications. Because Storm topologies operate on streaming data (rather than data at rest, as in HDFS) they are sensitive to data sources. When debugging Storm topologies, consider the following questions:

- What are my data sources?
- At what rate do these data sources deliver messages?
- What size are the messages?
- What is my slowest data sink?

The performance of a Storm topology degrades when it cannot ingest data fast enough to keep up with the data source. In addition, the velocity of incoming streaming data changes over time. When the data flow of the source exceeds what the topology can process, memory buffers fill up and the topology suffers frequent timeouts and must replay tuples to process them. (In contrast, MapReduce applications operate on data at rest in HDFS, with a constant data velocity. These applications suffer from poor latencies, but do not experience the buffer overflows and timeouts associated with streaming applications.)

Hortonworks recommends the following topology debugging technique to identify and overcome poor topology performance due to mismatched data flow rates between source and application.

1. Click Show Visualization in the Storm UI to display a visual representation of your topology and and find the data bottleneck in your Storm application. Thicker lines between components denote larger data flows. A blue component represents the the first component in the topology, such as the spout below from the WordCountTopology included with `storm-starter`. The color of the other topology components indicates whether the component is exceeding cluster capacity: red components denote a data bottleneck and green components indicate components operating within capacity.



Note

In addition to bolts in your topology, Storm uses its own bolts to perform background work when a topology component acknowledges that it either succeeded or failed to process a tuple. These names of these acker bolts are prefixed with an underscore in the visualization, but do not appear in the default view. You can display the component-specific data about successful acknowledgements by selecting the `_ack_ack` check box. Select the `_ack_fail` checkbox to display component-specific data about failed acknowledgements.

2. Verify that you have found the topology bottleneck by rewriting the `execute()` method of the target bolt or spout to perform no operations.

   If the performance of the topology improves, you have found the bottleneck. Alternatively, turn off each topology component, one at a time, to find the component responsible for the processing bottleneck.

3. Increase the timeout value for the topology.

   Edit the value of `topology.message.timeout.secs` in the storm.yaml configuration file. The default value is 30 seconds. This configuration parameter controls how long a tuple tree from the core-storm API or a batch from the Trident API has to complete processing before Storm times out and fails the operation.

4. Override the maximum number of tuples or batches waiting for processing before a spout temporarily stops emitting tuples to downstream bolts.

   Edit the value of `topology.max.spout.pending` in the storm.yaml configuration file. The default is no limit. Hortonworks recommends that topologies using the core-storm API start with a value of 1000 and slowly decrease the value as necessary. Toplogies using the Trident API should start with a much lower value, between 1 and 5.

5. Increase the parallelism for the target spout or bolt, as described in the next section.

3.2. Determining Topology Parallelism Units

Hortonworks recommends using the following calculation to determine the total number of parallelism units for a topology. Parallelism units are a useful conceptual tool for determining how to distribute processing tasks across a distributed application.

```
(number of worker nodes in cluster * number cores per worker node) - (number
 of acker tasks)
```

Acker tasks are topology components that acknowledge a successfully processed tuple. The following example assumes a Storm cluster with ten worker nodes, 16 CPU cores per worker node, and ten acker tasks in the topology. This Storm topology has 150 total parallelism units:

```
(10 * 16) - 10 = 150
```

Storm developers can mitigate the increased processing load associated with data persistence operations, such as writing to HDFS and generating reports, by distributing the most parallelism units to topology components that perform data persistence operations.

4. Streaming Data to Hive

Both the core-storm and Trident APIs support streaming data directly to Apache Hive using Hive transactions. Data committed in a transaction is immediately available to Hive queries from other Hive clients. Storm developers stream data to existing table partitions or configure the streaming Hive bolt to dynamically create desired table partitions. Use the following steps to perform this procedure:

1. Instantiate an implementation of the `HiveMapper` Interface.
2. Instantiate a `HiveOptions` class with the `HiveMapper` implementation.
3. Instantiate a `HiveBolt` with the `HiveOptions` class.

Note

Currently, data may be streamed only into bucketed tables using the ORC file format.

Core-storm API

```
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
 .withColumnFields(new Fields(colNames));
HiveOptions hiveOptions = new
HiveOptions(metaStoreURI,dbName,tblName,mapper);
HiveBolt hiveBolt = new HiveBolt(hiveOptions);
```

Trident API

```
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
 .withColumnFields(new Fields(colNames))
 .withTimeAsPartitionField("YYYY/MM/DD");

HiveOptions hiveOptions = new HiveOptions(metaStoreURI,dbName,tblName,mapper)
 .withTxnsPerBatch(10)
 .withBatchSize(1000)
 .withIdleTimeout(10)

StateFactory factory = new HiveStateFactory().withOptions(hiveOptions);
TridentState state = stream.partitionPersist(factory, hiveFields, new
 HiveUpdater(),
new Fields());
```

The rest of this topic describes these steps in greater detail.

Instantiate an Implementation of HiveMapper Interface

The storm-hive streaming bolt uses the `HiveMapper` interface to map the names of tuple fields to the names of Hive table columns. Storm provides two implementations: `DelimitedRecordHiveMapper` and `JsonRecordHiveMapper`. Both implementations take the same arguments.

Table 4.1. HiveMapper Arguments

Argument	Data Type	Description
`withColumnFields`	`backtype.storm.tuple.Fields`	The name of the tuple fields that you want to map to table column names.

Argument	Data Type	Description
withPartitionFields	backtype.storm.tuple.Fields	The name of the tuple fields that you want to map to table partitions.
withTimeAsPartitionField	String	Requests that table partitions be created with names set to system time. Developers can specify any Java-supported date format, such as "YYYY/MM/DD".

The following sample code illustrates how to use DelimitedRecordHiveMapper:

```
...
DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
 .withColumnFields(new Fields(colNames))
 .withPartitionFields(new Fields(partNames));

DelimitedRecordHiveMapper mapper = new DelimitedRecordHiveMapper()
 .withColumnFields(new Fields(colNames))
 .withTimeAsPartitionField("YYYY/MM/DD");
...
```

Instantiate a HiveOptions Class with the HiveMapper Implementation

Use the HiveOptions class to configure the transactions used by Hive to ingest the streaming data, as illustrated in the following code sample.

```
...
HiveOptions hiveOptions = new HiveOptions(metaStoreURI,dbName,tblName,mapper)
 .withTxnsPerBatch(10)
 .withBatchSize(1000)
 .withIdleTimeout(10);
...
```

The following table describes all configuration properties for the HiveOptions class.

Table 4.2. HiveOptions Class Configuration Properties

HiveOptions Configuration Property	Data Type	Description
metaStoreURI	String	Hive Metastore URI. Storm developers can find this value in hive-site.xml.
dbName	String	Database name
tblName	String	Table name
mapper	Mapper	Two properties that start with "org.apache.storm.hive.bolt.": mapper.DelimitedRecordHiveMapper mapperJsonRecordHiveMapper
withTxnsPerBatch	Integer	Configures the number of desired transactions per transaction batch. Data from all transactions in a single batch form a single compaction file. Storm developers use this property in conjunction with the withBatchSize property to control the size of compaction files. The default value is 100. Hive stores data in base files that cannot be updated by HDFS. Instead,

HiveOptions Configuration Property	Data Type	Description
		Hive creates a set of delta files for each transaction that alters a table or partition and stores them in a separate delta directory. Occasionally, Hive compacts, or merges, the base and delta files. Hive performs all compactions in the background without affecting concurrent reads and writes of other Hive clients. See Transactions for more information about Hive compactions.
`withMaxOpenConnections`	`Integer`	Specifies the maximum number of open connections. Each connection is to a single Hive table paritition. The default value is 500. When Hive reaches this threshold, an idle connection is terminated for each new connection request. A connection is considered idle if no data is written to the table partition to which the connection is made.
`withBatchSize`	`Integer`	Specifies the maximum number of Storm tuples written to Hive in a single Hive transaction. The default value is 15000 tuples.
`withCallTimeout`	`Integer`	Specifies the interval in seconds between consecutive heartbeats sent to Hive. Hive uses heartbeats to prevent expiration of unused transactions. Set this value to 0 to disable heartbeats. The default value is 240.
`withAutoCreatePartitions`	`Boolean`	Indicates whether HiveBolt should automatically create the necessary Hive partitions needed to store streaming data. The default value is true.
`withKerberosPrinicipal`	`String`	Kerberos user principal for accessing a secured Hive installation.
`withKerberosKeytab`	`String`	Kerberos keytab for accessing a secured Hive installation.

Instantiate the HiveBolt with the HiveOptions class

The next step is to instantiate the Hive streaming bolt. The core-storm and Trident APIs use different classes, as demonstrated in the following code samples:

Core-storm API

```
...
HiveBolt hiveBolt = new HiveBolt(hiveOptions);
...
```

Trident API

```
...
StateFactory factory = new HiveStateFactory().withOptions(hiveOptions);
TridentState state = stream.partitionPersist(factory, hiveFields, new
 HiveUpdater(),
new Fields());
...
```

5. Ingesting Data with the Apache Kafka Spout Connector

Apache Kafka is a high-throughput, distributed messaging system. Apache Storm provides a Kafka spout to facilitate ingesting data from Kafka 0.8x brokers. Storm developers should include downstream bolts in their topologies to process data ingested with the Kafka spout.

The `storm-kafka` components include a core-storm spout, as well as a fully transactional Trident spout. Storm-kafka spouts provide the following key features:

- 'Exactly once' tuple processing with the Trident API
- Dynamic discovery of Kafka brokers and partitions

Hortonworks recommends that Storm developers use the Trident API. However, use the core-storm API if sub-second latency is critical for your Storm topology.

- The core-storm API represents a Kafka spout with the `KafkaSpout` class.
- The Trident API provides a `OpaqueTridentKafkaSpout` class to represent the spout.

To initialize `KafkaSpout` and `OpaqueTridentKafkaSpout`, Storm developers need an instance of a subclass of the `KafkaConfig` class, which represents configuration information needed to ingest data from a Kafka cluster.

- The `KafkaSpout` constructor requires the `SpoutConfig` subclass.
- The `OpaqueTridentKafkaSpout` requires the `TridentKafkaConfig` subclass.

In turn, the constructors for both `KafkaSpout` and `OpaqueTridentKafkaSpout` require an implementation of the `BrokerHosts` interface, which is used to map Kafka brokers to topic partitions. The storm-kafka component provides two implementations of `BrokerHosts`: `ZkHosts` and `StaticHosts`.

- Use the `ZkHosts` implementation to dynamically track broker-to-partition mapping.
- Use the `StaticHosts` implementation for static broker-to-partition mapping.

5.1. KafkaSpout and OpaqueTridentKafkaSpout Examples

The following code samples demonstrate the use of the `KafkaSpout` and `OpaqueTridentKafkaSpout` classes and related interfaces.

Core-storm API

```
BrokerHosts hosts = new ZkHosts(zkConnString);
SpoutConfig spoutConfig = new SpoutConfig(hosts, topicName, "/" + zkrootDir,
 node);
```

```
spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
```

Trident API

```
TridentTopology topology = new TridentTopology();
BrokerHosts zk = new ZkHosts("localhost");
TridentKafkaConfig spoutConf = new TridentKafkaConfig(zk, "test-topic");
spoutConf.scheme = new SchemeAsMultiScheme(new StringScheme());
OpaqueTridentKafkaSpout spout = new OpaqueTridentKafkaSpout(spoutConf);
```

5.2. Storm-Kafka API Reference

Javadoc for the `storm-kafka` component is installed `at <$STORM_HOME>/contrib/storm-kafka/storm-kafka-0.9.3.2.2.6.0-<buildnumber>-javadoc.jar`. This section provides additional reference documentation for the primary classes and interfaces of the `storm-kafka` component.

BrokerHosts Interface

The `storm-kafka` component provides two implementations of the `BrokerHosts` interface: `ZkHosts` and `StaticHosts`. Use the `ZkHosts` implementation to dynamically track broker-to-partition mapping and the `StaticHosts` implementation when broker-to-partition mapping is static. The constructor for `StaticHosts` requires an instance of `GlobalPartitionInformation`:

```
Broker brokerForPartition0 = new Broker("localhost");//localhost:9092
Broker brokerForPartition1 = new Broker("localhost", 9092);//localhost:9092
 but we specified the port explicitly
Broker brokerForPartition2 = new Broker("localhost:9092");//localhost:9092
 specified as one string.
GlobalPartitionInformation partitionInfo = new GlobalPartitionInformation();
partitionInfo.add(0, brokerForPartition0)//mapping form partition 0 to
 brokerForPartition0
partitionInfo.add(1, brokerForPartition1)//mapping form partition 1 to
 brokerForPartition1
partitionInfo.add(2, brokerForPartition2)//mapping form partition 2 to
 brokerForPartition2
StaticHosts hosts = new StaticHosts(partitionInfo);
```

KafkaConfig Class

Instantiate an instance of `KafkaConfig` with one of the following constructors, each of which requires an implementation of the `BrokerHosts` interface:

```
public KafkaConfig(BrokerHosts hosts, String topic)
public KafkaConfig(BrokerHosts hosts, String topic, String clientId)
```

Table 5.1. KafkaConfig Parameters

KafkaConfig Parameter	Description
`hosts`	Any implementation of the `BrokerHosts` interface, currently either `ZkHosts` or `StaticHosts`.
`topic`	Name of the Kafka topic.
`clientId`	Optional parameter used as part of the ZooKeeper path where the spout's current offset is stored.

Both `SpoutConfig` from the core-storm API and `TridentKafkaConfig` from the Trident API extend `KafkaConfig`. Instantiate these classes with the following constructors:

Core-Storm API

Constructor `public SpoutConfig(BrokerHosts hosts, String topic, String zkRoot, String id)`

Table 5.2. SpoutConfig Parameters

SpoutConfig Parameter	Description
`hosts`	Any implementation of the `BrokerHosts` interface, currently either `ZkHosts` or `StaticHosts`.
`topic`	Name of the Kafka topic.
`zkroot`	Root directory in ZooKeeper where all topics and partition information is stored. By default, this is `/brokers`.
`id`	Unique identifier for this spout.

Trident API

Constructors `public TridentKafkaConfig(BrokerHosts hosts, String topic) public TridentKafkaConfig(BrokerHosts hosts, String topic, String id)`

Table 5.3. TridentKafkaConfig Parameters

TridentKafkaConfig	Description
`hosts`	Any implementation of the `BrokerHosts` interface, currently either `ZkHosts` or `StaticHosts`.
`topic`	Name of the Kafka topic.
`clientid`	Unique identifier for this spout.

`KafkaConfig` contains several fields used to configure the behavior of a Kafka spout in a Storm topology:

Table 5.4. KafkaConfig Fields

KafkaConfig Field	Description
`fetchSizeBytes`	Specifies the number of bytes to attempt to fetch in one request to a Kafka server. The default is 1MB.
`socketTimeoutMs`	Specifies the number of milliseconds to wait before a socket fails an operation with a timeout. The default value is 10 seconds.
`bufferSizeBytes`	Specifies the buffer size in bytes for network requests. The default is 1MB.
`scheme`	The interface that specifies how a byte[] from a Kafka topic is transformed into a Storm tuple. The default, `MultiScheme`, returns a tuple with the byte[] and no additional processing. The API provides the following implementations: * `storm.kafka.StringScheme` * `storm.kafka.KeyValueSchemeAsMultiScheme` * `storm.kafka.StringKeyValueScheme` * `storm.kafka.KeyValueSchemeAsMultiScheme`

KafkaConfig Field	Description
ignoreZKOffsets	To force the spout to ignore any consumer state information stored in ZooKeeper, set ignoreZkOffsets to true. If true, the spout always begins reading from the offset defined by startOffsetTime. For more information, see "How KafkaSpout stores offsets of a Kafka topic and recovers in case of failures."
startOffsetTime	Controls whether streaming for a topic starts from the beginning of the topic or whether only new messages are streamed. The following are valid values: * kafka.api.OffsetRequest.EarliestTime() - starts streaming from the beginning of the topic * kafka.api.OffsetRequest.LatestTime() - streams only new messages
maxOffsetBehind	Specifies how long a spout attempts to retry the processing of a failed tuple. If a failing tuple's offset is less then maxOffsetBehind, the spout stops retrying the tuple. The default is LONG.MAX_VALUE.
useStartOffsetTimeOfOffsetOutOfRange	Controls whether a spout streams messages from the beginning of a topic when the spout throws an exception for an out-of-range offset. The default value is true.
metricsTimeBucketSizeInSecs	Controls the time interval at which Storm reports spout-related metrics. The default is 60 seconds.

5.3. KafkaSpout Limitations

Limitations

The current version of the Kafka spout contains the following limitations:

- Does not support Kafka 0.7x brokers.
- Storm developers must include ${STORM_HOME}/lib/* in the CLASSPATH environment variable from the command line when running kafka-topology in local mode. Otherwise, developers will likely receive a java.lang.NoClassDefFoundError exception:

```
java -cp "/usr/lib/storm/contrib/storm-kafka-example-0.9.1.2.1.1.0-320-jar-
with-dependencies.jar:
 /usr/lib/storm/lib/*" org.apache.storm.kafka.TestKafkaTopology
 <zookeeper_host>
```

- Secure Hadoop clusters must comment out the following statement from ${STORM_HOME}/bin/kafka-server-start.sh:

```
EXTRA_ARGS="-name kafkaServer -loggc"
```

- Core-storm API Constructor

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

Table 5.5. SpoutConfig Parameters

SpoutConfig Parameter	Description
hosts	Any implementation of the BrokerHosts interface, currently either ZkHosts or StaticHosts.

SpoutConfig Parameter	Description
`topic`	Name of the Kafka topic.
`zkroot`	Root directory in ZooKeeper where all topics and partition information is stored. By default, this is `/brokers`.
`id`	Unique identifier for this spout.

- Trident API Constructors

```
public TridentKafkaConfig(BrokerHosts hosts, String topic)
public TridentKafkaConfig(BrokerHosts hosts, String topic, String id)
```

Table 5.6. TridentKafkaConfig Parameters

TridentKafkaConfig	Description
`hosts`	Any implementation of the `BrokerHosts` interface, currently either `ZkHosts` or `StaticHosts`.
`topic`	Name of the Kafka topic.
`clientid`	Unique identifier for this spout.

5.4. Configuring Kafka for Use with the Storm-Kafka Connector

The storm-kafka connector requires some configuration of the Apache Kafka installation. Kafka administrators must add a `zookeeper.connect` property, with the hostnames and port numbers of the HDP ZooKeeper nodes, to Kafka's `server.properties` file.

5.5. Configuring KafkaSpout to Connect to a Secure Kafka Cluster

To connect to a Kerberized Kafka topic:

1. Code: Add `spoutConfig.securityProtocol=PLAINTEXTSASL` to your Kafka Spout configuration.
2. Configuration: Add a KafkaClient section (excerpted from `/usr/hdp/current/kafka-broker/config/kafka_jaas.conf`) to `/usr/hdp/current/storm-supervisor/conf/storm_jaas.conf`:

```
  KafkaClient
  { com.sun.security.auth.module.Krb5LoginModule required useTicketCache=
true renewTicket=true serviceName="kafka"; }
  ;
  ...
```

3. Setup: Add a Kafka ACL for the topic:

```
/usr/hdp/current/kafka-broker/bin/kafka-acls.sh --topic TEST --
add allowhosts * --allowprincipals "User:stormusr" --operations
DESCRIBE,READ --config /usr/hdp/current/kafka-broker/config/
server.properties
```

6. Ingesting Data from HDFS

The HDFS spout actively monitors a specified HDFS directory and consumes any new files that appear in the directory, feeding data from HDFS to Storm.

Important

HDFS spout assumes that files visible in the monitored directory are not actively being updated. Only after a file is completely written should it be made visible to the spout. Following are two approaches for ensuring this:

- Write the file to another directory. When the write operation is finished, move the file to the monitored directory.
- Create the file in the monitored directory with an '.ignore' suffix; HDFS spout ignores files with an '.ignore' suffix. When the write operation is finished, rename the file to omit the suffix.

When the spout is actively consuming a file, it renames the file with an `.inprogress` suffix. After consuming all contents in the file, the file is moved to a configurable *done* directory and the `.inprogress` suffix is dropped.

Concurrency

If multiple spout instances are used in the topology, each instance consumes a different file. Synchronization among spout instances relies on lock files created in a subdirectory called `.lock` (by default) under the monitored directory. A file with the same name as the file being consumed (without the `.inprogress` suffix) is created in the lock directory. Once the file is completely consumed, the corresponding lock file is deleted.

Recovery from failure

Periodically, the spout records information about how much of the file has been consumed in the lock file. If the spout instance crashes or there is a force kill of topology, another spout can take over the file and resume from the location recorded in the lock file.

Certain error conditions (such as a spout crash) can leave residual lock files. Such a stale lock file indicates that the corresponding input file has not been completely processed. When detected, ownership of such stale lock files will be transferred to another spout.

The `hdfsspout.lock.timeout.sec` property specifies the duration of inactivity after which lock files should be considered stale. The default timeout is five minutes. For lock file ownership transfer to succeed, the HDFS lease on the file (from the previous lock owner) should have expired. Spouts scan for stale lock files before selecting the next file for consumption.

Lock on .lock Directory

HDFS spout instances create a `DIRLOCK` file in the `.lock` directory to coordinate certain accesses to the .lock directory itself. A spout will try to create it when it needs access to the .lock directory, and then delete it when done. In error conditions such as a topology crash, force kill, or untimely death of a spout, this file may not be deleted. Future instances

of the spout will eventually recover the file once the `DIRLOCK` file becomes stale due to inactivity for `hdfsspout.lock.timeout.sec` seconds.

API Support

HDFS spout supports core Storm, but does not currently support Trident.

6.1. Configuring HDFS Spout

The following member functions are required for `HdfsSpout`:

`.setReaderType()`	Specifies which file reader to use: • To read sequence files, set this to `'seq'`. • To read text files, set this to `'text'`. • If you want to use a custom file reader class that implements interface `org.apache.storm.hdfs.spout.FileReader`, set this to the fully qualified class name.
`.withOutputFields()`	Specifies names of output fields for the spout. The number of fields depends upon the reader being used. For convenience, built-in reader types expose a static member called `defaultFields` that can be used for setting this.
`.setHdfsUri()`	Specifies the HDFS URI for HDFS NameNode; for example: `hdfs://namenodehost:8020`.
`.setSourceDir()`	Specifies the HDFS directory from which to read files; for example, `/data/inputdir`.
`.setArchiveDir()`	Specifies the HDFS directory to move a file after the file is completely processed; for example, `/data/done`. If this directory does not exist, it will be created automatically.
`.setBadFilesDir()`	Specifies a directory to move a file if there is an error parsing the contents of the file; for example, `/data/badfiles`. If this directory does not exist it will be created automatically.

For additional configuration settings, see Apache HDFS spout Configuration Settings.

6.2. HDFS Spout Example

The following example creates an HDFS spout that reads text files from HDFS path `hdfs://localhost:54310/source`.

```
// Instantiate spout to read text files
HdfsSpout textReaderSpout = newHdfsSpout().setReaderType("text")
```

```
                                        .withOutputFields(TextFileReader.
defaultFields)
                                        .setHdfsUri("hdfs://
localhost:54310")  // reqd
                                        .setSourceDir("/data/in")
   // reqd
                                        .setArchiveDir("/data/done")
   // reqd
                                        .setBadFilesDir("/data/badfiles");
   // required

// If using Kerberos
HashMap hdfsSettings = new HashMap();
hdfsSettings.put("hdfs.keytab.file", "/path/to/keytab");
hdfsSettings.put("hdfs.kerberos.principal","user@EXAMPLE.com");

textReaderSpout.setHdfsClientSettings(hdfsSettings);

// Create topology
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("hdfsspout", textReaderSpout, SPOUT_NUM);

// Set up bolts and wire up topology
   ...

// Submit topology with config
Config conf = new Config();
StormSubmitter.submitTopologyWithProgressBar("topologyName", conf, builder.
createTopology());
```

A sample topology `HdfsSpoutTopology` is provided in the `storm-starter` module.

7. Writing Data with Storm

Hortonworks provides a set of connectors that enable Storm developers to quickly write streaming data to a Hadoop cluster. These connectors are located at `/usr/lib/storm/contrib`. Each contains a `.jar` file containing the connector's packaged classes and dependencies, and another `.jar` file with javadoc reference documentation.

This chapter describes how to use several connectors, and how to configure connectors in a Kerberos-enabled cluster. For a more thorough discussion of Apache Storm connectors and APIs, see the Apache Storm documentation for your version of Storm.

7.1. Writing Data to HDFS

The `storm-hdfs` connector supports core Storm and Trident APIs. You should use the trident API unless your application requires sub-second latency.

7.1.1. Storm-HDFS: Core Storm APIs

The primary classes of the `storm-hdfs` connector are `HdfsBolt` and `SequenceFileBolt`, both located in the `org.apache.storm.hdfs.bolt` package. Use the `HdfsBolt` class to write text data to HDFS and the `SequenceFileBolt` class to write binary data.

Specify the following information when instantiating the bolt:

`HdfsBolt` Methods

`withFsUrl`	Specifies the target HDFS URL and port number.
`withRecordFormat`	Specifies the delimiter that indicates a boundary between data records. Storm developers can customize by writing their own implementation of the `org.apache.storm.hdfs.format.RecordFormat` interface. Use the provided `org.apache.storm.hdfs.format.DelimitedRecordFormat` class as a convenience class for writing delimited text data with delimiters such as tabs, comma-separated values, and pipes. The `storm-hdfs` bolt uses the `RecordFormat` implementation to convert tuples to byte arrays, so this method can be used with both text and binary data.
`withRotationPolicy`	Specifies when to stop writing to a data file and begin writing to another. Storm developers can customize by writing their own implementation of the `org.apache.storm.hdfs.rotation.FileSizeRotationSizePolicy` interface.
`withSyncPolicy`	Specifies how frequently to flush buffered data to the HDFS filesystem. This action enables other HDFS clients to read the synchronized data, even as the Storm

	client continues to write data. Storm developers can customize by writing their own implementation of the `org.apache.storm.hdfs.sync.SyncPolicy` interface.
`withFileNameFormat`	Specifies the name of the data file. Storm developers can customize by writing their own interface of the `org.apache.storm.hdfs.format.FileNameFormat` interface. The provided `org.apache.storm.hdfs.format.DefaultFileNameFormat` creates file names with the following naming format: `{prefix}-{componentId}-{taskId}-{rotationNum}-{timestamp}-{extension}`. Example: `MyBolt-5-7-1390579837830.txt`.

Example: Cluster Without High Availability ("HA")

The following example writes pipe-delimited files to the HDFS path `hdfs://localhost:8020/foo`. After every 1,000 tuples it will synchronize with the filesystem, making the data visible to other HDFS clients. It will rotate the files when they reach 5 MB in size.

Note that the HdfsBolt is instantiated with an HDFS URL and port number.

```
```java
// use "|" instead of "," for field delimiter
RecordFormat format = new DelimitedRecordFormat()
        .withFieldDelimiter("|");

// Synchronize the filesystem after every 1000 tuples
SyncPolicy syncPolicy = new CountSyncPolicy(1000);

// Rotate data files when they reach 5 MB
FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f, Units.
MB);

// Use default, Storm-generated file names
FileNameFormat fileNameFormat = new DefaultFileNameFormat()
        .withPath("/foo/");


// Instantiate the HdfsBolt
HdfsBolt bolt = new HdfsBolt()
        .withFsUrl("hdfs://localhost:8020")
        .withFileNameFormat(fileNameFormat)
        .withRecordFormat(format)
        .withRotationPolicy(rotationPolicy)
        .withSyncPolicy(syncPolicy);
```
```

Example: HA-Enabled Cluster

The following example shows how to modify the previous example for an HA-enabled cluster.

Here the HdfsBolt is instantiated with a nameservice ID, instead of using an HDFS URL and port number.

```
...
HdfsBolt bolt = new HdfsBolt()
           .withFsURL("hdfs://myNameserviceID")
           .withFileNameFormat(fileNameformat)
           .withRecordFormat(format)
           .withRotationPolicy(rotationPolicy)
           .withSyncPolicy(syncPolicy);
...
```

To obtain the nameservice ID, check the `dfs.nameservices` property in your `hdfs-site.xml` file; `nnha` in the following example:

```
<property>
  <name>dfs.nameservices</name>
  <value>nnha</value>
</property>
```

7.1.2. Storm-HDFS: Trident APIs

The Trident API implements a `StateFactory` class with an API that resembles the methods from the `storm-code` API, as shown in the following code sample:

```
...
Fields hdfsFields = new Fields("field1", "field2");

FileNameFormat fileNameFormat = new DefaultFileNameFormat()
 .withPrefix("trident")
 .withExtension(".txt")
 .withPath("/trident");

RecordFormat recordFormat = new DelimitedRecordFormat()
 .withFields(hdfsFields);

FileRotationPolicy rotationPolicy = new FileSizeRotationPolicy(5.0f,
 FileSizeRotationPolicy.Units.MB);

HdfsState.Options options = new HdfsState.HdfsFileOptions()
 .withFileNameFormat(fileNameFormat)
 .withRecordFormat(recordFormat)
 .withRotationPolicy(rotationPolicy)
 .withFsUrl("hdfs://localhost:8020");

StateFactory factory = new HdfsStateFactory().withOptions(options);

TridentState state = stream.partitionPersist(factory, hdfsFields, new
 HdfsUpdater(), new Fields());
```

See the javadoc for the Trident API, included with the `storm-hdfs` connector, for more information.

Limitations

Directory and file names changes are limited to a prepackaged file name format based on a timestamp.

7.2. Writing Data to HBase

The `storm-hbase` connector supports the following key features:

- Apache HBase 0.96 and above
- Incrementing counter columns
- Tuples failure if an update to an HBase table fails
- Ability to group puts in a single batch
- Writing to Kerberized HBase clusters (for more information, see Configuring Connectors for a Secure Cluster)

The `storm-hbase` connector enables Storm developers to collect several *PUTS* in a single operation and write to multiple HBase column families and counter columns. A PUT is an HBase operation that inserts data into a single HBase cell. Use the HBase client's write buffer to automatically batch: `hbase.client.write.buffer`. The primary interface in the `storm-hbase` connector is the `org.apache.storm.hbase.bolt.mapper.HBaseMapper` interface. However, the default implementation, `SimpleHBaseMapper`, writes a single column family. Storm developers can implement the `HBaseMapper` interface themselves or extend `SimpleHBaseMapper` if they want to change or override this behavior.

Table 7.1. SimpleHBaseMapper Methods

SimpleHBaseMapper Method	Description
`withRowKeyField`	Specifies the row key for the target HBase row. A row key uniquely identifies a row in HBase.
`withColumnFields`	Specifies the target HBase column.
`withCounterFields`	Specifies the target HBase counter.
`withColumnFamily`	Specifies the target HBase column family.

Example

The following example specifies the 'word' tuple as the row key, adds an HBase column for the tuple 'word' field, adds an HBase counter column for the tuple 'count' field, and writes data to the 'cf' column family.

```
SimpleHBaseMapper mapper = new SimpleHBaseMapper()
 .withRowKeyField("word")
 .withColumnFields(new Fields("word"))
 .withCounterFields(new Fields("count"))
 .withColumnFamily("cf");
```

The `storm-hbase` connector supports the following versions of HBase:

- 0.96
- 0.98

Limitations

The current version of the `storm-hbase` connector has the following limitations:

- HBase table must be predefined

- Cannot dynamically add new HBase columns; can write to only one column family at a time
- Assumes that `hbase-site.xml` is in the $CLASSPATH environment variable
- Tuple field names must match HBase column names
- Does not support the Trident API
- Supports writes but not lookups

7.3. Writing Data to Kafka

Storm provides a Kafka Bolt for both the core-storm and Trident APIs that writes data to a Kafka cluster, also known as publishing to a topic using Kafka's terminology. Use the following procedure to add a Storm component to your topology that writes data to a Kafka cluster:

1. Instantiate a Kafka Bolt.
2. Configure the Kafka Bolt with a Tuple-to-Message mapper.
3. Configure the Kafka Bolt with a Kafka Topic Selector.
4. Configure the Kafka Bolt with the Kafka Producer properties.

The following code samples for each API illustrate the construction of a simple Kafka Bolt. The rest of this topic breaks the samples down to better describe each step.

Core-storm API

```
TopologyBuilder builder = new TopologyBuilder();

Fields fields = new Fields("key", "message");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
new Values("storm", "1"),
new Values("trident", "1"),
new Values("needs", "1"),
new Values("javadoc", "1")
);

spout.setCycle(true);
builder.setSpout("spout", spout, 5);
KafkaBolt bolt = new KafkaBolt()
.withKafkaTopicSelector(new DefaultTopicSelector("test"))
.withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
builder.setBolt("forwardToKafka", bolt, 8).shuffleGrouping("spout");

Config conf = new Config();
//set producer properties.
Properties props = new Properties();
props.put("metadata.broker.list", "localhost:9092");
props.put("request.required.acks", "1");
props.put("serializer.class", "kafka.serializer.StringEncoder");
conf.put(TridentKafkaState.KAFKA_BROKER_PROPERTIES, props);

StormSubmitter.submitTopology("kafkaboltTest", conf, builder.
createTopology());
```

Trident API

```
Fields fields = new Fields("word", "count");
FixedBatchSpout spout = new FixedBatchSpout(fields, 4,
new Values("storm", "1"),
new Values("trident", "1"),
new Values("needs", "1"),
new Values("javadoc", "1")
);

spout.setCycle(true);

TridentTopology topology = new TridentTopology();
Stream stream = topology.newStream("spout1", spout);

TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
.withKafkaTopicSelector(new DefaultTopicSelector("test"))
.withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper("word",
"count"));
stream.partitionPersist(stateFactory, fields, new TridentKafkaUpdater(), new
Fields());

Config conf = new Config();
//set producer properties.
Properties props = new Properties();
props.put("metadata.broker.list", "localhost:9092");
props.put("request.required.acks", "1");
props.put("serializer.class", "kafka.serializer.StringEncoder");
conf.put(TridentKafkaState.KAFKA_BROKER_PROPERTIES, props);
StormSubmitter.submitTopology("kafkaTridentTest", conf, topology.build());
```

Instantiate a KafkaBolt

The core-storm API uses the `storm.kafka.bolt.KafkaBolt` class to instantiate a Kafka Bolt. The Trident API uses a combination of the `storm.kafka.trident.TridentStateFactory` and `storm.kafka.trident.TridentKafkaStateFactory` classes.

Core-storm API

```
KafkaBolt bolt = new KafkaBolt();
```

Trident API

```
TridentTopology topology = new TridentTopology();
 Stream stream = topology.newStream("spout");
 TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory();
 stream.partitionPersist(stateFactory, fields, new TridentKafkaUpdater(), new
 Fields());
```

Configure the KafkaBolt with a Tuple-to-Message Mapper

The KafkaBolt must map Storm tuples to Kafka messages. By default, KafkaBolt looks for fields named "key" and "message." Storm provides the `storm.kafka.trident.mapper.FieldNameBasedTupleToKafkaMapper` class to support this default behavior and provide backward compatibility. The class is used by both the core-storm and Trident APIs.

Core-storm API

```
KafkaBolt bolt = new KafkaBolt()
 .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
```

Trident API

```
TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
 .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper("word",
 "count"));
```

Storm developers must specify the field names for the Storm tuple key and the Kafka message for any implementation of the `TridentKafkaState` in the Trident API. This interface does not provide a default constructor.

However, some Kafka bolts may require more than two fields. Storm developers may write their own implementation of the `TupleToKafkaMapper` and `TridentTupleToKafkaMapper` interfaces to customize the mapping of Storm tuples to Kafka messages. Both interfaces define 2 methods:

```
K getKeyFromTuple(Tuple/TridentTuple tuple);
```

```
V getMessageFromTuple(Tuple/TridentTuple tuple);
```

Configure the Kafka Bolt with a Kafka Topic Selector

Note

To ignore a message, return NULL from the `getTopics()` method.

Core-storm API

```
KafkaBolt bolt = new KafkaBolt().withTupleToKafkaMapper(new
 FieldNameBasedTupleToKafkaMapper())
 .withTopicSelector(new DefaultTopicSelector());
```

Trident API

```
TridentKafkaStateFactory stateFactory = new TridentKafkaStateFactory()
 .withKafkaTopicSelector(new DefaultTopicSelector("test"))
 .withTridentTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper("word",
 "count"));
```

Storm developers can write their own implementation of the `KafkaTopicSelector` interface if they need to write to multiple Kafka topics:

```
public interface KafkaTopicSelector {
 String getTopics(Tuple/TridentTuple tuple);
 }
```

7.4. Configuring Connectors for a Secure Cluster

If your topology uses Storm-HDFS, Storm-HBase, or Storm-Hive connectors, and if the corresponding components HDFS, HBase, and/or Hive are secured with Kerberos, then you will need to complete the following additional configuration steps.

Storm-HDFS Connector Configuration

To use the `storm-hdfs` connector in topologies that run on secure clusters:

1. Provide your own Kerberos keytab and principal name to the connectors. The `Config` object that you pass into the topology must contain the storm keytab file and principal name.

2. Specify an HdfsBolt `configKey`, using the method `HdfsBolt.withConfigKey("somekey")`. The value map of this key should have the following two properties:

   `hdfs.keytab.file: "<path-to-keytab>"`

   `hdfs.kerberos.principal: "<principal>@<host>"`

   where

   `<path-to-keytab>` specifies the path to the keytab file on the supervisor hosts

   `<principal>@<host>` specifies the user and domain; for example, `storm-admin@EXAMPLE.com`.

   For example:

   ```
   Config config = new Config();
   config.put(HdfsSecurityUtil.STORM_KEYTAB_FILE_KEY, "$keytab");
   config.put(HdfsSecurityUtil.STORM_USER_NAME_KEY, "$principal");

   StormSubmitter.submitTopology("$topologyName", config, builder.
   createTopology());
   ```

   On worker hosts the bolt/trident-state code will use the keytab file and principal to authenticate with the NameNode. Make sure that all workers have the keytab file, stored in the same location.

3. Distribute the keytab file that the Bolt is using in the Config object, to all supervisor nodes. This is the keytab that is being used to authenticate to HDFS, typically the Storm service keytab, `storm`. The user ID that the Storm worker is running under should have access to it.

   On an Ambari-managed cluster this is `/etc/security/keytabs/storm.service.keytab` (the `"path-to-keytab"`), where the worker runs under `storm`.

4. If you set `supervisor.run.worker.as.user` to `true` (see Running Workers as Users in *Configuring Storm for Kerberos over Ambari*), make sure that the user that the workers are running under (typically the `storm` keytab) has read access on those keytabs. This is a manual step; an admin needs to go to each supervisor node and run chmod to give file system permissions to the users on these keytab files.

Note

You do not need to create separate keytabs or principals; the general guideline is to create a principal and keytab for each group of users that requires the same access to these resources, and use that single keytab.

All of these connectors accept topology configurations. You can specify the keytab location on the host and the principal through which the connector will login to that system.

5. Configure the connector(s). Here is a sample configuration for the Storm-HDFS connector (see Writing Data to HDFS with the Storm-HDFS Connector for a more extensive example):

```
HdfsBolt bolt = new HdfsBolt()
        .withFsUrl("hdfs://localhost:8020")
        .withFileNameFormat(fileNameFormat)
        .withRecordFormat(format)
        .withRotationPolicy(rotationPolicy)
        .withSyncPolicy(syncPolicy);
        .withConfigKey("hdfs.config");

Map<String, Object> map = new HashMap<String,Object>();
map.put("hdfs.keytab.file","/etc/security/keytabs/storm.service.keytab");
map.put("hdfs.kerberos.principal","storm@TEST.HORTONWORKS.COM");

Config config = new Config();
config.put("hdfs.config", map);

StormSubmitter.submitTopology("$topologyName",config,builder.
createTopology());
```

Important

For the Storm-HDFS connector, you must package `hdfs-site.xml` and `core-site.xml` (from your cluster configuration) in the topology .jar file.

In addition, include any configuration files for HDP components used in your Storm topology, such as hive-site.xml and hbase-site.xml. This fulfills the requirement that all related configuration files appear in the CLASSPATH of your Storm topology at runtime.

Storm-HBase Connector Configuration

To use the `storm-hbase` connector in topologies that run on secure clusters:

1. Provide your own Kerberos keytab and principal name to the connectors. The `Config` object that you pass into the topology must contain the storm keytab file and principal name.

2. Specify an HBaseBolt `configKey`, using the method `HBaseBolt.withConfigKey("somekey")`. The value map of this key should have the following two properties:

   `storm.keytab.file: "<path-to-keytab-file>"`

   `storm.kerberos.principal: "<principal>@<host>"`

   For example:

```
Config config = new Config();
config.put(HBaseSecurityUtil.STORM_KEYTAB_FILE_KEY, "$keytab");
config.put(HBaseSecurityUtil.STORM_USER_NAME_KEY, "$principal");

StormSubmitter.submitTopology("$topologyName", config, builder.
createTopology());
```

On worker hosts the bolt/trident-state code will use the keytab file and principal to authenticate with the NameNode. Make sure that all workers have the keytab file, stored in the same location.

3. Distribute the keytab file that the Bolt is using in the Config object, to all supervisor nodes. This is the keytab that is being used to authenticate to HBase,typically the Storm service keytab, `storm`. The user ID that the Storm worker is running under should have access to it.

   On an Ambari-managed cluster this is `/etc/security/keytabs/storm.service.keytab` (the `"path-to-keytab"`), where the worker runs under `storm`.

4. If you set `supervisor.run.worker.as.user` to `true` (see Running Workers as Users in *Configuring Storm for Kerberos over Ambari*), make sure that the user that the workers are running under (typically the `storm` keytab) has read access on those keytabs. This is a manual step; an admin needs to go to each supervisor node and run chmod to give file system permissions to the users on these keytab files.

   Note

   You do not need to create separate keytabs or principals; the general guideline is to create a principal and keytab for each group of users that requires the same access to these resources, and use that single keytab.

   All of these connectors accept topology configurations. You can specify the keytab location on the host and the principal through which the connector will login to that system.

5. Configure the connector(s). Here is a sample configuration for the Storm-HBase connector:

```
HBaseBolt hbase = new HBaseBolt("WordCount", mapper).withConfigKey("hbase.
config");

Map<String, Object> mapHbase = new HashMap<String,Object>();
mapHbase.put("storm.keytab.file","/etc/security/keytabs/storm.service.
keytab");
mapHbase.put("storm.kerberos.principal","storm@TEST.HORTONWORKS.COM");

Config config = new Config();
config.put("hbase.config",mapHbase);

StormSubmitter.submitTopology("$topologyName",config,builder.
createTopology());
```

Important

For the Storm-HBase connector, you must package `hdfs-site.xml`, `core-site.xml`, and `hbase-site.xml` (from your cluster configuration) in the topology .jar file.

In addition, include any other configuration files for HDP components used in your Storm topology, such as hive-site.xml. This fulfills the requirement that all related configuration files appear in the CLASSPATH of your Storm topology at runtime.

Storm-Hive Connector Configuration

The Storm-Hive connector accepts configuration settings as part of the HiveOptions class.

There are two required settings for accessing secure Hive:

- `withKerberosPrincipal`, the Kerberos principal for accessing Hive:

```
public HiveOptions withKerberosPrincipal(String kerberosPrincipal)
```

- `withKerberosKeytab`, the Kerberos keytab for accessing Hive:

```
public HiveOptions withKerberosKeytab(String kerberosKeytab)
```

8. Packaging Storm Topologies

Storm developers should verify that the following conditions are met when packaging their topology into a .jar file:

- Use the maven-shade-plugin, rather than the maven-assembly-plugin to package your Apache Storm topologies. The maven-shade-plugin provides the ability to merge JAR manifest entries, which are used by the Hadoop client to resolve URL schemes.
- Include a dependency for the Hadoop version used in the Hadoop cluster.
- Include both of the Hadoop configuration files, `hdfs-site.xml` and `core-site.xml`, in the .jar file. In addition, include any configuration files for HDP components used in your Storm topology, such as `hive-site.xml` and `hbase-site.xml`. This is the easiest way to meet the requirement that all required configuration files appear in the CLASSPATH of your Storm topology at runtime.

Maven Shade Plugin

Use the maven-shade-plugin, rather than the maven-assembly-plugin to package your Apache Storm topologies. The maven-shade-plugin provides the ability to merge JAR manifest entries, which are used by the Hadoop client to resolve URL schemes.

Use the following Maven configuration file to package your topology:

```
 <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>1.4</version>
      <configuration>
            <createDependencyReducedPom>true</createDependencyReducedPom>
      </configuration>
      <executions>
           <execution>
                <phase>package</phase>
                <goals>
                     <goal>shade</goal>
                </goals>
                <configuration>
                     <transformers>
                          <transformer implementation="org.apache.maven.
plugins.shade.resource.ServicesResourceTransformer"/>
                          <transformer implementation="org.apache.maven.
plugins.shade.resource.ManifestResourceTransformer">
                               <mainClass></mainClass>
                          </transformer>
                     </transformers>
                </configuration>
           </execution>
      </executions>
</plugin>
```

Hadoop Dependency

Include a dependency for the Hadoop version used in the Hadoop cluster; for example:

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-client</artifactId>
    <version>2.7.1.2.3.2.0-2950</version>
    <exclusions>
        <exclusion>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-log4j12</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

Troubleshooting

The following table describes common packaging errors.

Table 8.1. Topology Packing Errors

Error	Description
`com.google.protobuf. InvalidProtocolBufferException: Protocol message contained an invalid tag (zero)`	Hadoop client version incompatibility
`java.lang.RuntimeException: Error preparing HdfsBolt: No FileSystem for scheme: hdfs`	The .jar manifest files have not properly merged in the topology.jar

9. Deploying and Managing Apache Storm Topologies

Use the command-line interface to deploy a Storm topology after packaging it in a jar. For example, use the following command to deploy WordCountTopology from the `storm-starter` jar:

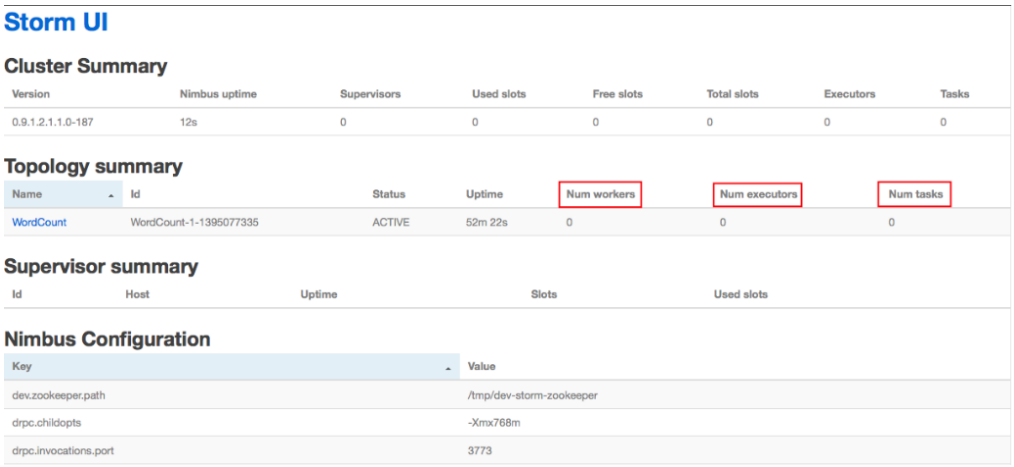
```
storm jar storm-starter-<starter_version>-storm-<storm_version>
 .jar storm.starter.WordCountTopology WordCount -c nimbus.host=sandbox.
hortonworks.com
```

Point a browser to the following URL to access the Storm UI and to manage deployed topologies.

`http://<storm-ui-server>:8080`

Note

You may need to configure Apache Storm to use a different port if Ambari is also running on the same host with Apache Storm. Both applications use port 8080 by default.



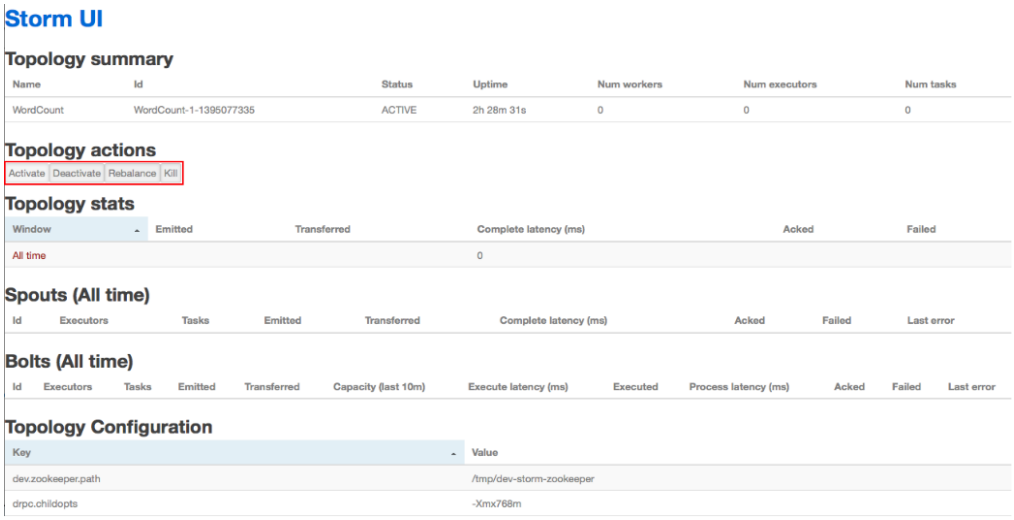
In the image above, no workers, executors, or tasks are running. However, the status of the topology remains active and the uptime continues to increase. Storm topologies, unlike traditional applications, remain active until an administrator deactivates or kills them. Storm administrators use the Storm user interface to perform the following administrative actions:

Table 9.1. Topology Administrative Actions

Topology Administrative Action	Description
Activate	Returns a topology to active status after it has been deactivated.
Deactivate	Sets the status of a topology to inactive. Topology uptime is not affected by deactivation.

Topology Administrative Action	Description
Rebalance	Dynamically increase or decrease the number of worker processes and/or executors. The administrator does not need to restart the cluster or the topology.
Kill	Stops the topology and removes it from Apache Storm. The topology no longer appears in the Storm UI, and the administrator must deploy the application again to activate it.

Click any topology in the Topology Summary section to launch the Topology Summary page. Administrators perform any of the topology actions in the table above by clicking the corresponding button, shown in the following image.



The Executors field in the Spouts and Bolts sections show all running Storm threads, including the host and port. If a bolt is experiencing latency issues, Storm developers should look here to determine which executor has reached capacity. Click the port number to display the log file for the corresponding executor.

10. Example: RollingTopWords Topology

The `RollingTopWords.java` is included with `storm-starter`.

```
package storm.starter;

import backtype.storm.Config;
import backtype.storm.testing.TestWordSpout;
import backtype.storm.topology.TopologyBuilder;
import backtype.storm.tuple.Fields;
import storm.starter.bolt.IntermediateRankingsBolt;
import storm.starter.bolt.RollingCountBolt;
import storm.starter.bolt.TotalRankingsBolt;
import storm.starter.util.StormRunner;

/**
* This topology does a continuous computation of the top N words that the
 topology has seen in terms of cardinality.
* The top N computation is done in a completely scalable way, and a similar
 approach could be used to compute things
* like trending topics or trending images on Twitter.
*/
public class RollingTopWords {

 private static final int DEFAULT_RUNTIME_IN_SECONDS = 60;
 private static final int TOP_N = 5;

 private final TopologyBuilder builder;
 private final String topologyName;
 private final Config topologyConfig;
 private final int runtimeInSeconds;

 public RollingTopWords() throws InterruptedException {
 builder = new TopologyBuilder();
 topologyName = "slidingWindowCounts";
 topologyConfig = createTopologyConfiguration();
 runtimeInSeconds = DEFAULT_RUNTIME_IN_SECONDS;

 wireTopology();
 }

 private static Config createTopologyConfiguration() {
 Config conf = new Config();
 conf.setDebug(true);
 return conf;
 }

 private void wireTopology() throws InterruptedException {
 String spoutId = "wordGenerator";
 String counterId = "counter";
 String intermediateRankerId = "intermediateRanker";
 String totalRankerId = "finalRanker";
 builder.setSpout(spoutId, new TestWordSpout(), 5);
 builder.setBolt(counterId, new RollingCountBolt(9, 3), 4).
fieldsGrouping(spoutId, new Fields("word"));
 builder.setBolt(intermediateRankerId, new IntermediateRankingsBolt(TOP_N),
 4).fieldsGrouping(counterId, new Fields( "obj"));
 builder.setBolt(totalRankerId, new TotalRankingsBolt(TOP_N)).
globalGrouping(intermediateRankerId);
```

```
 }

 public void run() throws InterruptedException {
 StormRunner.runTopologyLocally(builder.createTopology(), topologyName,
 topologyConfig, runtimeInSeconds);
 }

 public static void main(String[] args) throws Exception {
 new RollingTopWords().run();
 }
}
```