Hortonworks Data Platform

Apache Storm Component Guide

(June 1, 2017)

docs.cloudera.com

Hortonworks Data Platform: Apache Storm Component Guide

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.

Table of Contents

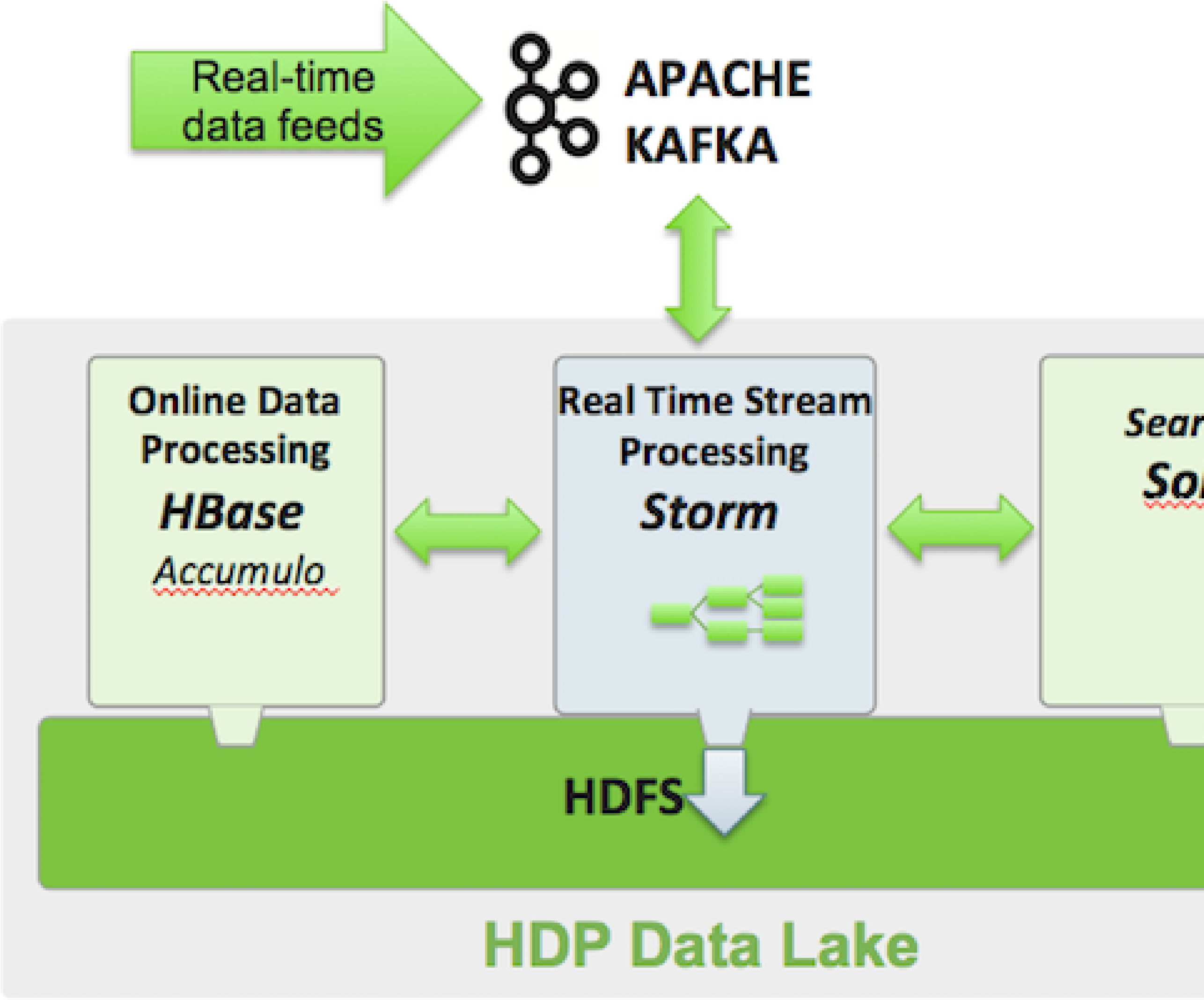
List of Tables

1. Analyzing Streams of Data with Apache Storm

The exponential increase in streams of data from real-time sources requires 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 smartphone. Apache Storm enables data-driven, automated activity by providing a realtime, scalable, fault-tolerant, highly available, distributed solution for streaming data.

Apache Storm is datatype-agnostic; it processes data streams of any data type. It can be used with any programming language, and guarantees that data streams are processed without data loss.

The following graphic illustrates a typical stream processing architecture:



2. Installing Apache Storm

Before installing Storm, ensure that your cluster meets the following prerequisites:

- HDP cluster stack version 2.5.0 or later.
- (Optional) Ambari version 2.4.0 or later.

Although you can install Apache Storm on a cluster not managed by Ambari (see Installing and Configuring Apache Storm in the Non-Ambari Cluster Installation Guide), this chapter describes how to install Storm on an Ambari-managed cluster.

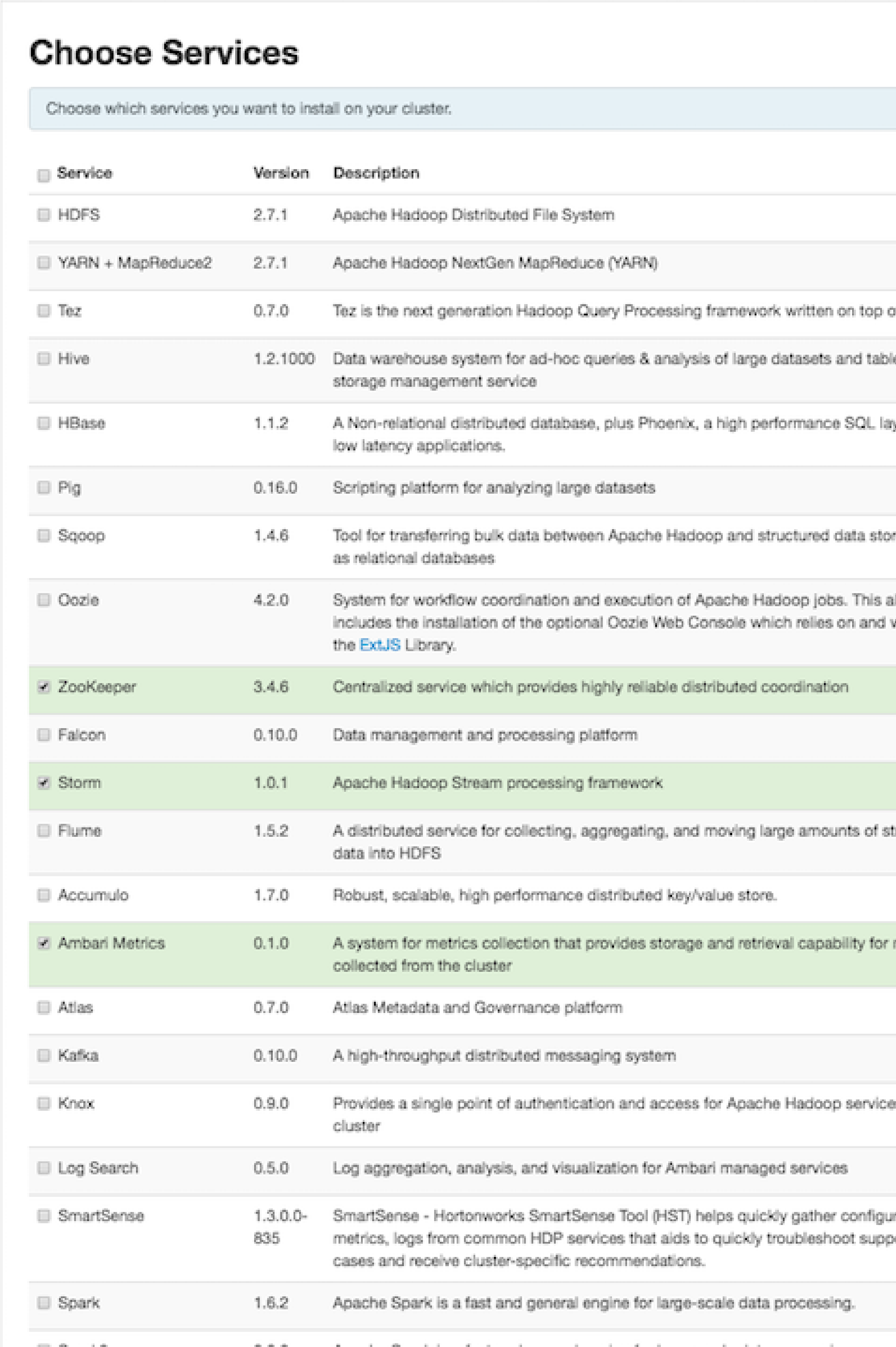
Note

Storm is not supported on the Windows operating system.

Before you install Storm using Ambari, refer to Adding a Service in the *Ambari Operations Guide* for background information about how to install HDP components using Ambari.

To install Storm using Ambari, complete the following steps.

1. Click the Ambari "Services" tab.
2. In the Ambari "Actions" menu, select "Add Service." This starts the Add Service Wizard, displaying the Choose Services screen. Some of the services are enabled by default.
3. Scroll down through the alphabetic list of components on the Choose Services page, select "Storm", and click "Next" to continue:



Choose Services

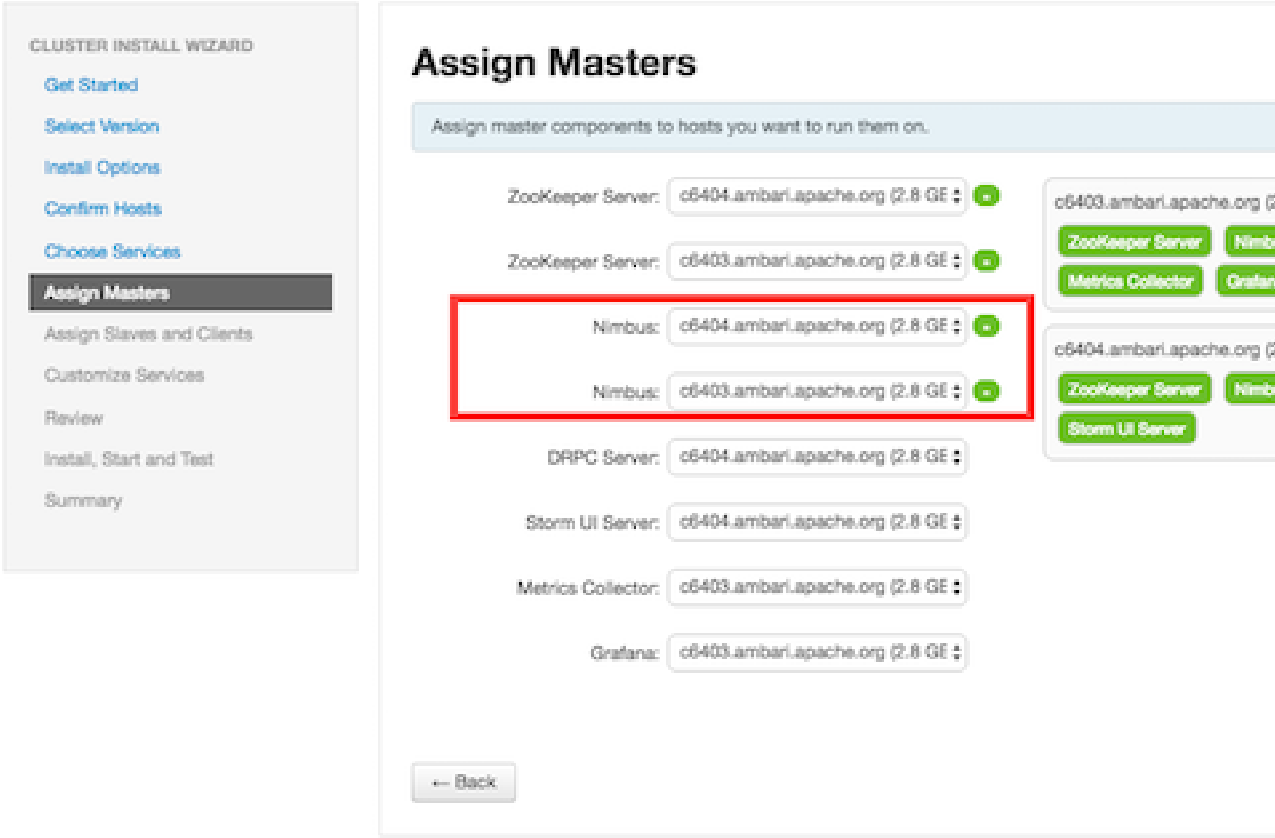
Choose which services you want to install on your cluster.

☐ Service	Version	Description
☐ HDFS	2.7.1	Apache Hadoop Distributed File System
☐ YARN + MapReduce2	2.7.1	Apache Hadoop NextGen MapReduce (YARN)
☐ Tez	0.7.0	Tez is the next generation Hadoop Query Processing framework written on top o
☐ Hive	1.2.1000	Data warehouse system for ad-hoc queries & analysis of large datasets and tabl storage management service
☐ HBase	1.1.2	A Non-relational distributed database, plus Phoenix, a high performance SQL la low latency applications.
☐ Pig	0.16.0	Scripting platform for analyzing large datasets
☐ Sqoop	1.4.6	Tool for transferring bulk data between Apache Hadoop and structured data sto as relational databases
☐ Oozie	4.2.0	System for workflow coordination and execution of Apache Hadoop jobs. This a includes the installation of the optional Oozie Web Console which relies on and the ExtJS Library.
☑ ZooKeeper	3.4.6	Centralized service which provides highly reliable distributed coordination
☐ Falcon	0.10.0	Data management and processing platform
☑ Storm	1.0.1	Apache Hadoop Stream processing framework
☐ Flume	1.5.2	A distributed service for collecting, aggregating, and moving large amounts of st data into HDFS
☐ Accumulo	1.7.0	Robust, scalable, high performance distributed key/value store.
☑ Ambari Metrics	0.1.0	A system for metrics collection that provides storage and retrieval capability for collected from the cluster
☐ Atlas	0.7.0	Atlas Metadata and Governance platform
☐ Kafka	0.10.0	A high-throughput distributed messaging system
☐ Knox	0.9.0	Provides a single point of authentication and access for Apache Hadoop service cluster
☐ Log Search	0.5.0	Log aggregation, analysis, and visualization for Ambari managed services
☐ SmartSense	1.3.0.0-835	SmartSense - Hortonworks SmartSense Tool (HST) helps quickly gather configu metrics, logs from common HDP services that aids to quickly troubleshoot supp cases and receive cluster-specific recommendations.
☐ Spark	1.6.2	Apache Spark is a fast and general engine for large-scale data processing.

4. On the Assign Masters page, review node assignments for Storm components.

   If you want to run Storm with high availability of nimbus nodes, select more than one nimbus node; the Nimbus daemon automatically starts in HA mode if you select more than one nimbus node.

   Modify additional node assignments if desired, and click "Next".



Assign Masters

Assign master components to hosts you want to run them on.

ZooKeeper Server: c6404.ambari.apache.org (2.8 GE

ZooKeeper Server: c6403.ambari.apache.org (2.8 GE

Nimbus: c6404.ambari.apache.org (2.8 GE

Nimbus: c6403.ambari.apache.org (2.8 GE

DRPC Server: c6404.ambari.apache.org (2.8 GE

Storm UI Server: c6404.ambari.apache.org (2.8 GE

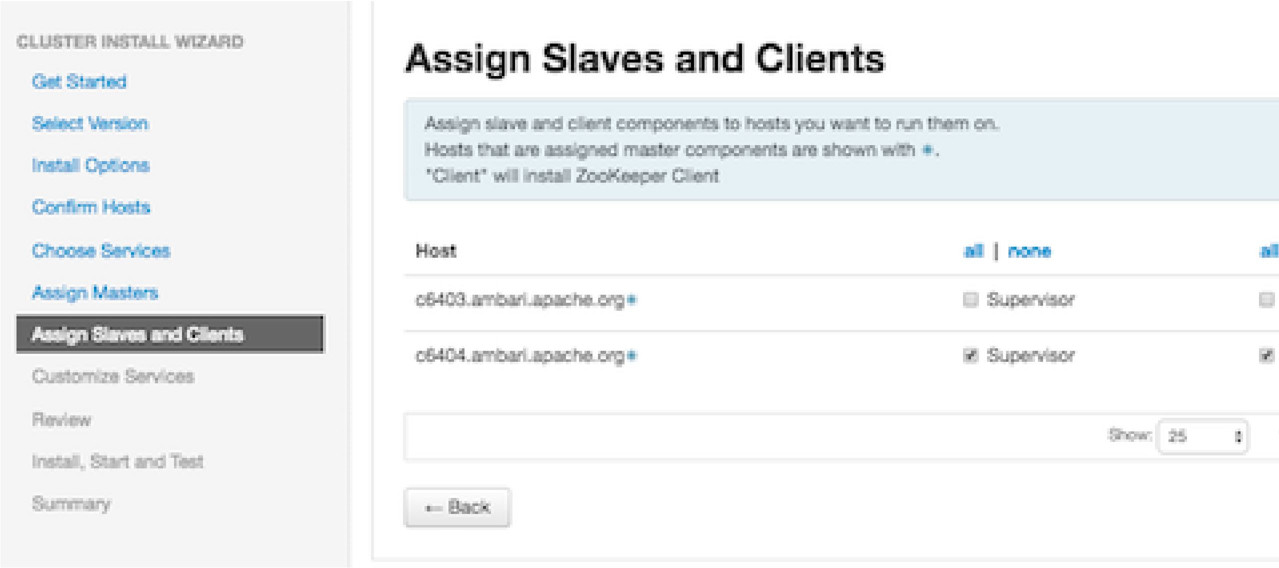
Metrics Collector: c6403.ambari.apache.org (2.8 GE

Grafana: c6403.ambari.apache.org (2.8 GE

c6403.ambari.apache.org (2

c6404.ambari.apache.org (2

5. On the Assign Slaves and Clients page, choose the nodes that you want to run Storm supervisors and clients:



Assign Slaves and Clients

Assign slave and client components to hosts you want to run them on.
Hosts that are assigned master components are shown with ✱.
"Client" will install ZooKeeper Client

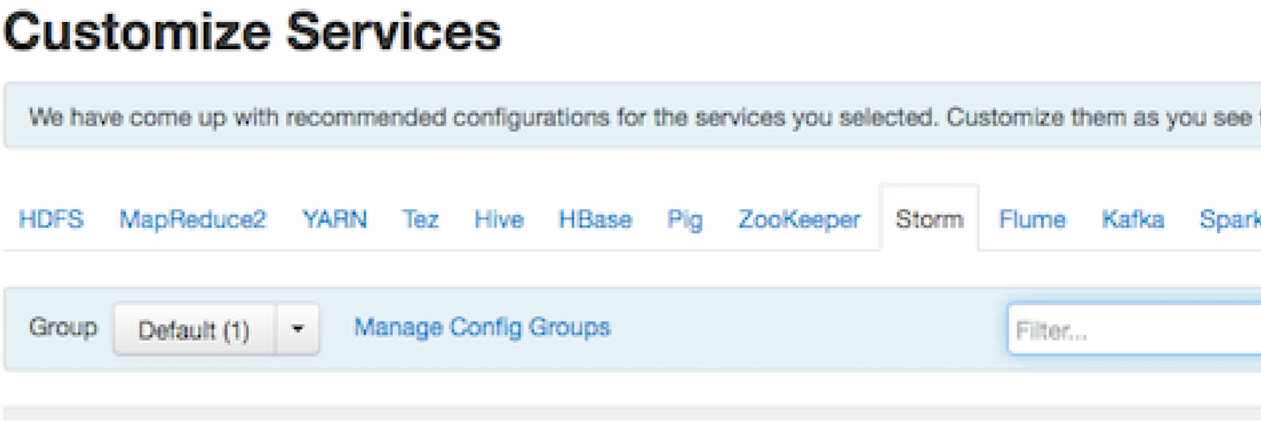
Host	all | none
c6403.ambari.apache.org✱	□ Supervisor
c6404.ambari.apache.org✱	☑ Supervisor

Storm supervisors are nodes from which the actual worker processes launch to execute spout and bolt tasks.

Storm clients are nodes from which you can run Storm commands (`jar`, `list`, and so on).

6. Click "Next" to continue.

7. Ambari displays the Customize Services page, which lists a series of services:



Customize Services

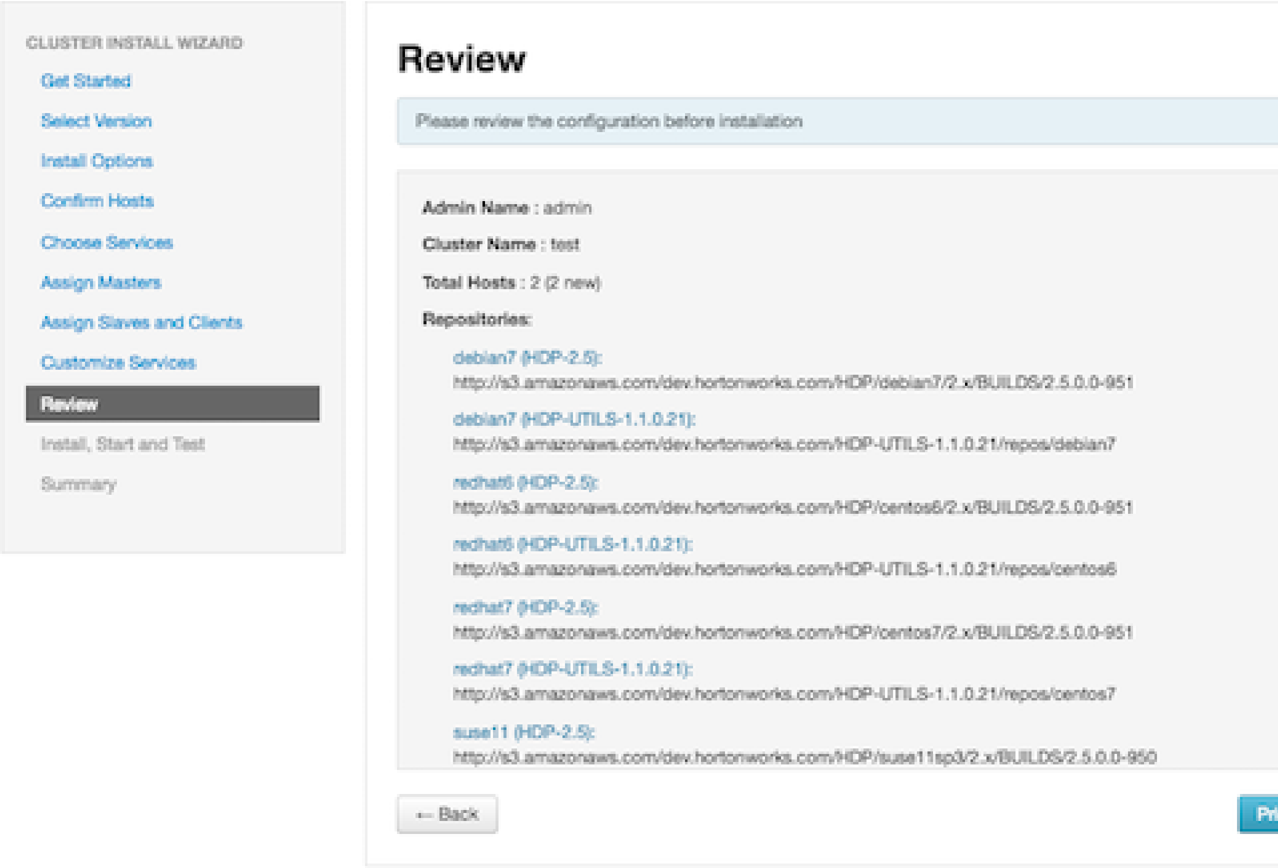
We have come up with recommended configurations for the services you selected. Customize them as you see

For your initial configuration you should use the default values set by Ambari. If Ambari prompts you with the message "Some configurations need your attention before you can proceed," review the list of properties and provide the required information.

For information about optional settings that are useful in production environments, see Configuring Apache Storm.

8. Click "Next" to continue.

9. When the wizard displays the Review page, ensure that all HDP components correspond to HDP 2.5.0 or later:



Review

Please review the configuration before installation

Admin Name : admin

Cluster Name : test

Total Hosts : 2 (2 new)

Repositories:

debian7 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/debian7/2.x/BUILDS/2.5.0.0-951

debian7 (HDP-UTILS-1.1.0.21):
http://s3.amazonaws.com/dev.hortonworks.com/HDP-UTILS-1.1.0.21/repos/debian7

redhat6 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/centos6/2.x/BUILDS/2.5.0.0-951

redhat6 (HDP-UTILS-1.1.0.21):
http://s3.amazonaws.com/dev.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos6

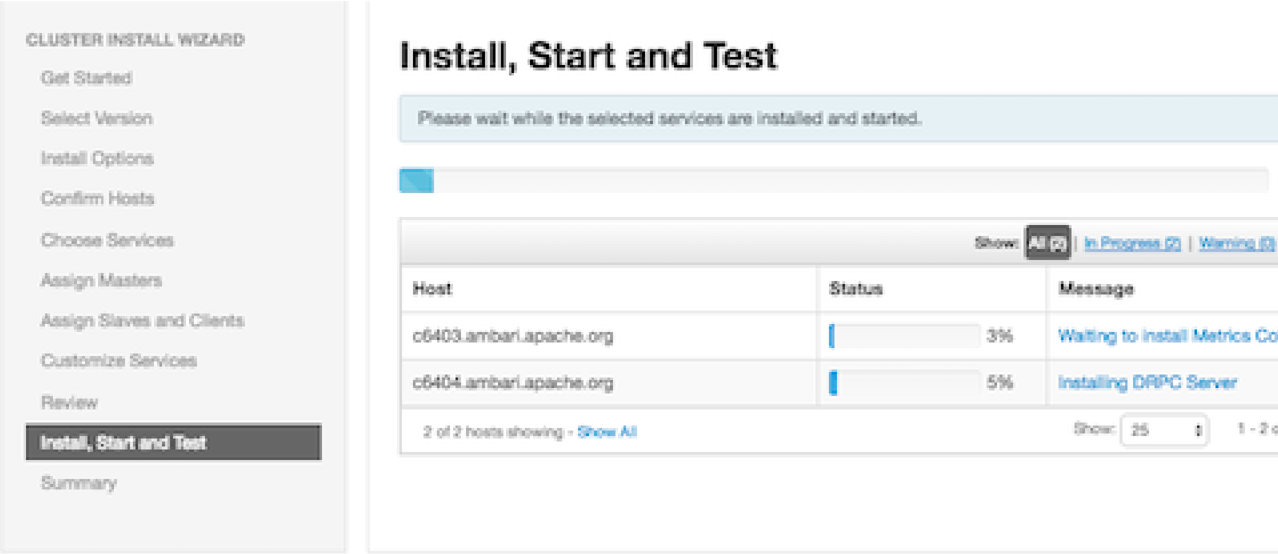
redhat7 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/centos7/2.x/BUILDS/2.5.0.0-951

redhat7 (HDP-UTILS-1.1.0.21):
http://s3.amazonaws.com/dev.hortonworks.com/HDP-UTILS-1.1.0.21/repos/centos7

suse11 (HDP-2.5):
http://s3.amazonaws.com/dev.hortonworks.com/HDP/suse11sp3/2.x/BUILDS/2.5.0.0-950

10.Click "Deploy" to begin installation.

11.Ambari displays the Install, Start and Test page. Review the status bar and messages for progress updates:



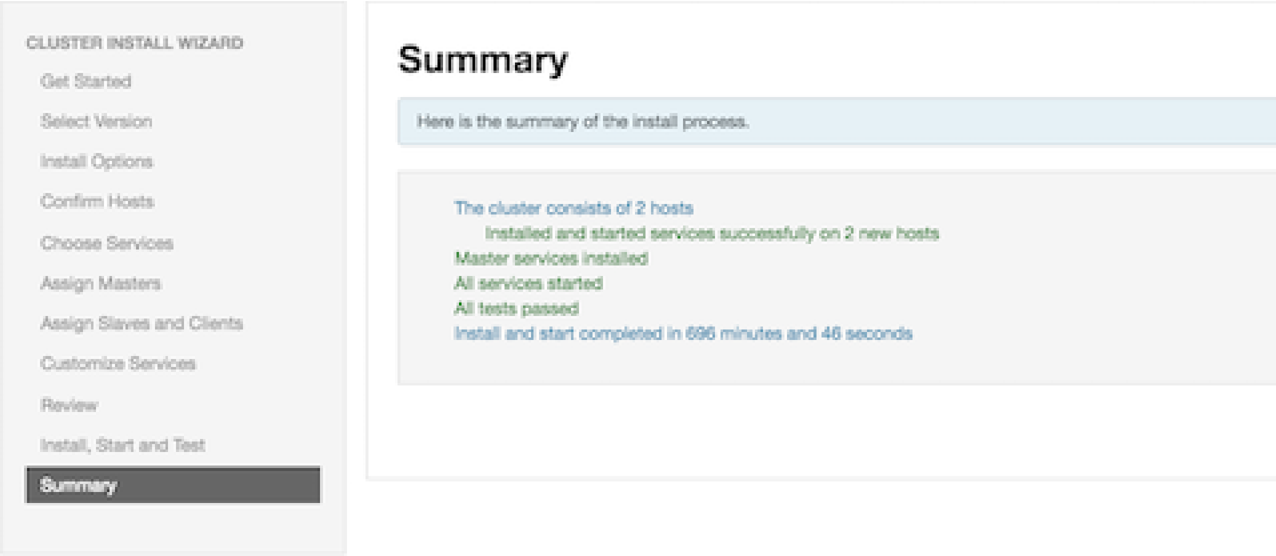
Install, Start and Test

Please wait while the selected services are installed and started.

Host	Status	Message
c6403.ambari.apache.org	3%	Waiting to install Metrics Co
c6404.ambari.apache.org	5%	Installing DRPC Server

2 of 2 hosts showing - Show All

12.When the wizard presents a summary of results, click "Complete" to finish installing Storm:



Summary

Here is the summary of the install process.

The cluster consists of 2 hosts
Installed and started services successfully on 2 new hosts
Master services installed
All services started
All tests passed
Install and start completed in 696 minutes and 46 seconds

To validate the Storm installation, complete the following steps:

1. Point your browser to the Storm UI URL for Ambari: `http://<storm-ui-server>:8744`

   You should see the Storm UI web page.

2. Submit the following command:

   ```
   storm jar /usr/hdp/current/storm-client/contrib/
   storm-starter/storm-starter-topologies-*.jar
   org.apache.storm.starter.WordCountTopology wordcount
   ```

3. The WordCount sample topology should run successfully.

3. Configuring Apache Storm for a Production Environment

This chapter covers topics related to Storm configuration:

- Configuring Storm to operate under supervision
- Properties to review when you place topologies into production use
- Enabling audit to HDFS for a secure cluster

Instructions are for Ambari-managed clusters.

3.1. Configuring Storm for Supervision

If you are deploying a production cluster with Storm, you should configure the Storm components to operate under supervision.

Follow these steps to configure Storm for supervision:

1. Stop all Storm components.

   a. Using Ambari Web, browse to `Services > Storm > Service Actions`.

   b. Choose Stop, and wait until the Storm service completes.

2. Stop Ambari Server:

   ```
   ambari-server stop
   ```

3. Change the Supervisor and Nimbus command scripts in the Stack definition.

   On Ambari Server host, run:

   ```
   sed -ir "s/scripts\/supervisor.py/scripts\/supervisor_prod.py/g" /var/lib/
   ambari-server/resources/common-services/STORM/0.9.1.2.1/metainfo.xml

   sed -ir "s/scripts\/nimbus.py/scripts\/nimbus_prod.py/g" /var/lib/ambari-
   server/resources/common-services/STORM/0.9.1.2.1/metainfo.xml
   ```

4. Install `supervisord` on all Nimbus and Supervisor hosts.

   - Install EPEL repository:

     ```
     yum install epel-release -y
     ```

   - Install supervisor package for `supervisord`:

     ```
     yum install supervisor -y
     ```

   - Enable `supervisord` on autostart:

```
chkconfig supervisord on
```

- Change `supervisord` configuration file permissions:

  ```
  chmod 600 /etc/supervisord.conf
  ```

5. Configure `supervisord` to supervise Nimbus Server and Supervisors by appending the following to `/etc/supervisord.conf` on all Supervisor host and Nimbus hosts:

```
[program:storm-nimbus]
command=env PATH=$PATH:/bin:/usr/bin/:/usr/jdk64/jdk1.7.0_67/bin/ JAVA_HOME=
/usr/jdk64/jdk1.7.0_67 /usr/hdp/current/storm-nimbus/bin/storm nimbus
user=storm
autostart=true
autorestart=true
startsecs=10
startretries=999
log_stdout=true
log_stderr=true
logfile=/var/log/storm/nimbus.out
logfile_maxbytes=20MB
logfile_backups=10

[program:storm-supervisor]
command=env PATH=$PATH:/bin:/usr/bin/:/usr/jdk64/jdk1.7.0_67/bin/ JAVA_HOME=
/usr/jdk64/jdk1.7.0_67 /usr/hdp/current/storm-supervisor/bin/storm
 supervisor
user=storm
autostart=true
autorestart=true
startsecs=10
startretries=999
log_stdout=true
log_stderr=true
logfile=/var/log/storm/supervisor.out
logfile_maxbytes=20MB
logfile_backups=10
```

Note

Change `/usr/jdk64/jdk1.7.0_67` to the location of the JDK being used by Ambari in your environment.

6. Start `supervisord` on all Supervisor and Nimbus hosts:

```
service supervisord start
```

7. Start Ambari Server:

```
ambari-server start
```

8. Start all other Storm components:

   a. Using Ambari Web, browse to `Services > Storm > Service Actions`.

   b. Choose `Start`.

3.2. Configuring Storm Resource Usage

The following settings can be useful for tuning Storm topologies in production environments.

Instructions are for a cluster managed by Ambari. For clusters that are not managed by Ambari, update the property in its configuration file; for example, update the value of `topology.message.timeout.secs` in the `storm.yaml` configuration file. (Do not update files manually if your cluster is managed by Ambari.)

Memory Allocation

Worker process max heap size: `worker.childopts -XmX` option

Maximum JVM heap size for the worker JVM. The default Ambari value is 768 MB. On a production system, this value should be based on workload and machine capacity. If you observe out-of-memory errors in the log, increase this value and fine tune it based on throughput; 1024 MB should be a reasonable value to start with.

To set maximum heap size for the worker JVM, navigate to the "Advanced storm-site" category and append the `-Xmx` option to worker.childopts setting. The following option sets maximum heap size to 1 GB: `-Xmx1024m`

Logviewer process max heap size: `logviewer.childopts -Xmx` option

Maximum JVM heap size for the logviewer process. The default is 128 MB. On production machines you should consider increasing the `logviewer.childopts -Xmx` option to 768 MB or more (1024 MB should be a sufficient for an upper-end value).

Message Throughput

`topology.max.spout.pending`

Maximum number of messages that can be pending in a spout at any time. The default is null (no limit).

The setting applies to all core Storm and Trident topologies in a cluster:

- For core Storm, this value specifies the maximum number of *tuples* that can be pending: tuples that have been emitted from a spout but have not been acked or failed yet.
- For Trident, which process batches in core, this property specifies the maximum number of *batches* that can be pending.

If you expect bolts to be slow in processing tuples (or batches) and you do not want internal buffers to fill up and temporarily stop emitting tuples to downstream bolts, you should set `topology.max.spout.pending` to a starting value of 1000 (for core Storm) or a value of 1 (for Trident), and increase the value depending on your throughput requirements.

You can override this value for a specific topology when you submit the topology. The following example restricts the number of pending tuples to 100 for a topology:

```
$ storm jar -c
topology.max.spout.pending=100 jar
args...
```

If you plan to use windowing functionality, set this value to null, or increase it to cover the estimated maximum number of active tuples in a single window. For example, if you define a sliding window with a duration of 10 minutes and a sliding interval of 1 minute, set `topology.max.spout.pending` to the maximum number of tuples that you expect to receive within an 11-minute interval.

This setting has no effect on spouts that do not anchor tuples while emitting.

`topology.message.timeout.secs`

Maximum amount of time given to the topology to fully process a tuple tree from the core-storm API, or a batch from the Trident API, emitted by a spout. If the message is not acked within this time frame, Storm fails the operation on the spout. The default is 30 seconds.

If you plan to use windowing functionality, set this value based on your windowing definitions. For example, if you define a 10 minute sliding window with a 1 minute sliding interval, you should set this value to at least 11 minutes.

You can also set this value at the topology level when you submit a topology; for example:

```
$ storm jar -c
topology.message.timeout.secs=660 jar
args...
```

Nimbus Node Resources

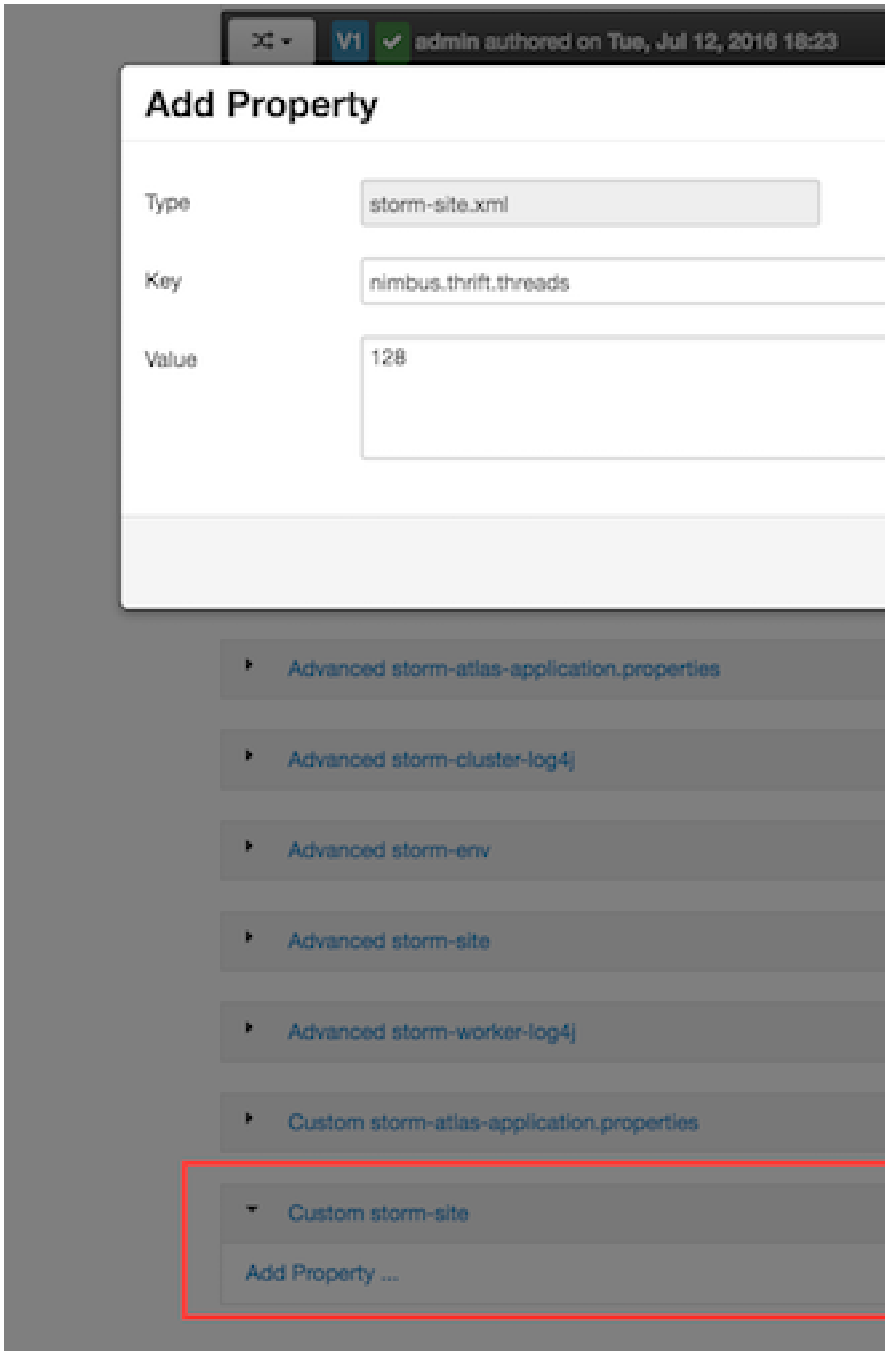
`nimbus.thrift.max_buffer_size`

Maximum buffer size that the Nimbus Thrift server allocates for servicing requests. The default is 1 MB. If you plan to submit topology files larger than 100 MB, consider increasing this value.

`nimbus.thrift.threads`

Number of threads to be used by the Nimbus Thrift server. The default is 64 threads. If you have more than ten hosts in your Storm cluster, consider increasing this value to a minimum of 196 threads, to handle the workload associated with multiple workers making multiple requests on each host.

You can set this value by adding the property and its value in the Custom storm-site category, as shown in the following graphic:


V1
admin authored on Tue, Jul 12, 2016 18:23
Add Property
Advanced storm-atlas-application.properties
Advanced storm-cluster-log4j
Advanced storm-env
Advanced storm-site
Advanced storm-worker-log4j
Custom storm-atlas-application.properties
Custom storm-site
Add Property ...

Type storm-site.xml

Key nimbus.thrift.threads

Value 128

Number of Workers on a Supervisor Node

`supervisor.slots.ports`

List of ports that can run workers on a supervisor node. The length of this list defines the number of workers that can be run on a supervisor node; there is one communication port per worker.

Use this configuration to tune how many workers to run on each machine. Adjust the value based on how many resources each worker will consume, based on the topologies you will submit (as opposed to machine capacity).

Number of Event Logger Tasks

`topology.eventlogger.executors`

Number of event logger tasks created for topology event logging. The default is 0; no event logger tasks are created.

If you enable topology event logging, you must set this value to a number greater than zero, or to `null`:

- `topology.eventlogger.executors: <n>` creates n event logger tasks for the topology. A value of 1 should be sufficient to handle most event logging use cases.
- `topology.eventlogger.executors: null` creates one event logger task per worker. This is only needed if you plan to use a high sampling percentage, such as logging all tuples from all spouts and bolts.

Storm Metadata Directory

`storm.local.dir`

Local directory where Storm daemons store topology metadata. You need not change the default value, but if you do change it, set it to a durable directory (not a directory such as /tmp).

3.3. Enabling Audit to HDFS for a Secure Cluster

To enable audit to HDFS when running Storm on a secure cluster, perform the steps listed at the bottom of Manually Updating Ambari HDFS Audit Settings in the *HDP Security Guide*.

4. Developing Apache Storm Applications

This chapter focuses on several aspects of Storm application development. Throughout this guide you will see references to core Storm and Trident. Trident is a layer of abstraction built on top of Apache Storm, with higher-level APIs. Both operate on unbounded streams of tuple-based data, and both address the same use cases: real-time computations on unbounded streams of data.

Here are some examples of differences between core Storm and Trident:

- The basic primitives in core storm are *bolts* and *spouts*. The core data abstraction in Trident is the *stream*.
- Core Storm processes events individually. Trident supports the concept of transactions, and processes data in micro-batches.
- Trident was designed to support stateful stream processing, although as of Apache Storm 1.0, core Storm also supports stateful stream processing.
- Core Storm supports a wider range of programming languages than Trident.
- Core Storm supports at-least-once processing very easily, but for exactly-once semantics, Trident is easier (from an implementation perspective) than using core Storm primitives.

A complete introduction to the Storm API is beyond the scope of this documentation. However, the following sections provide an overview of core Storm and Trident concepts. See Apache Storm documentation for an extensive description of Apache Storm concepts.

4.1. Core Storm Concepts

Developing a Storm application requires an understanding of the following basic concepts.

Table 4.1. Storm Concepts

Storm Concept	Description
Tuple	A named list of values of any data type. A tuple is 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.
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
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.
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 on slave nodes. The Nimbus node is a thrift service that is responsible for distributing code around the cluster, assigning 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 restarting failed worker processes.

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

4.1.1. 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"));
 }

}
```

4.1.2. 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 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) {
```

```
 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;
 }

}
```

4.1.3. 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 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.

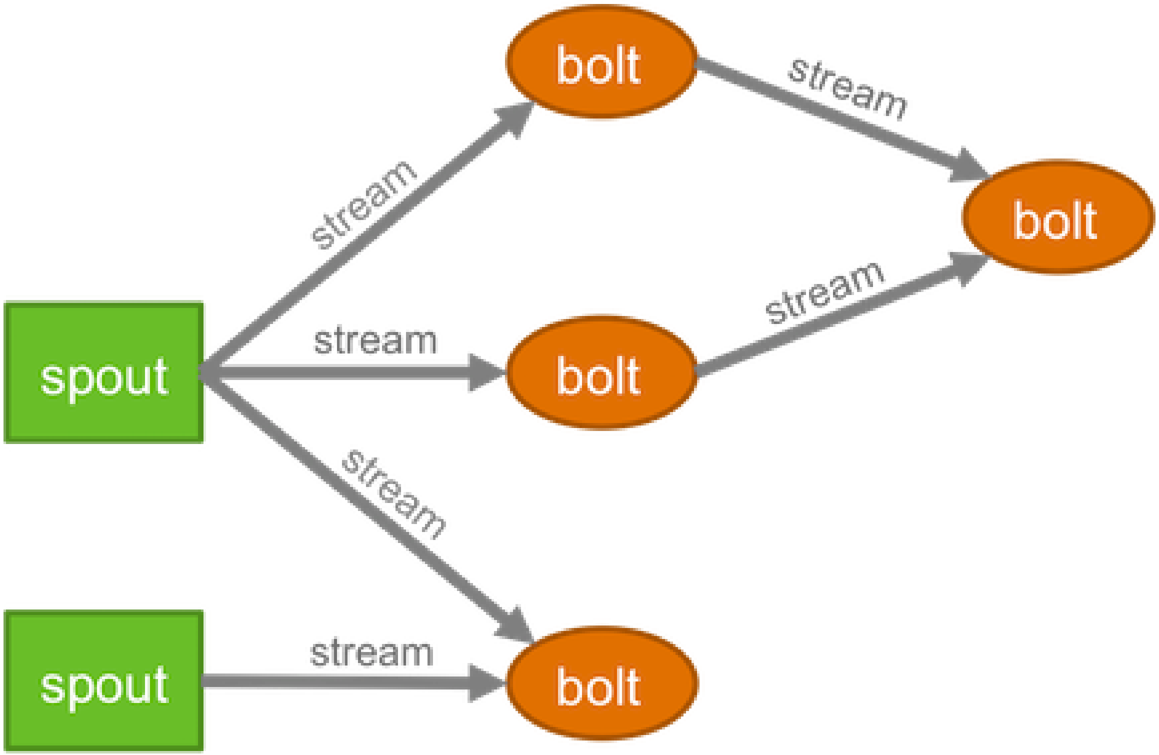
Storm developers specify the field grouping for each bolt using methods on the `TopologyBuilder.BoltGetter` inner class, as shown in the following excerpt from the the `WordCountTopology.java` example included with `storm-starter`.

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

4.1.4. 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());
...
```

4.1.5. Processing Reliability

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

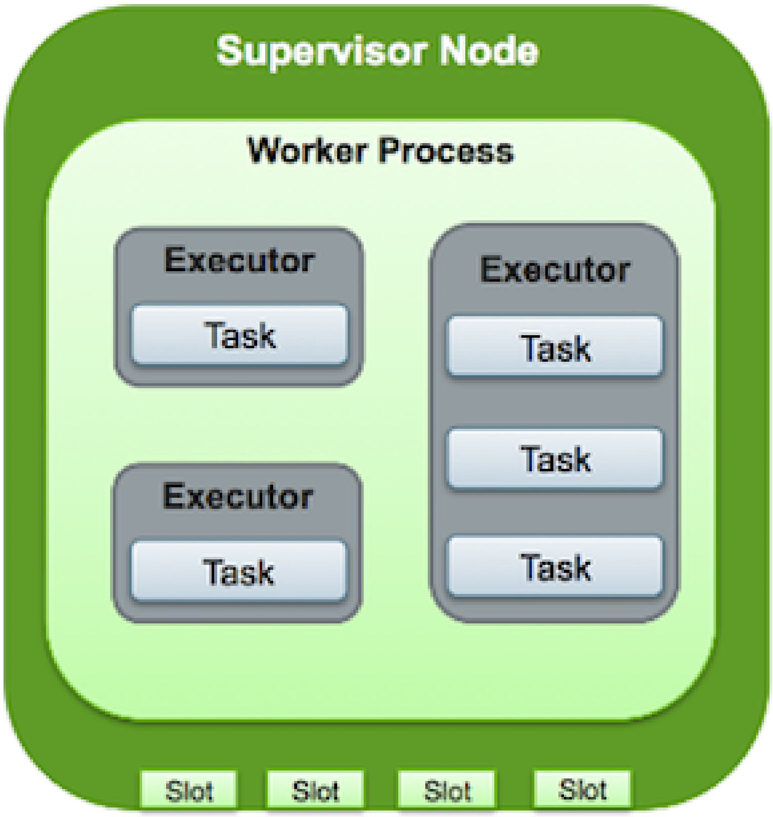
Table 4.3. Processing Guarantees

Guarantee	Description
At least once	Reliable; Tuples are processed at least once, but may be processed 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.)

4.1.6. 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.



4.1.7. Parallelism

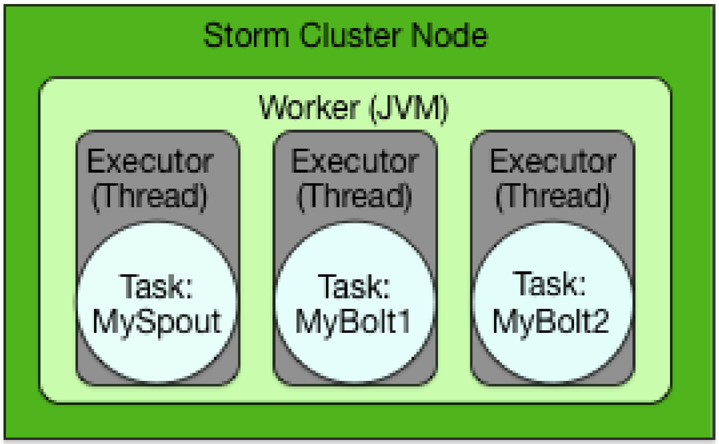
Distributed applications take advantage of horizontally-scaled clusters by dividing computation tasks across nodes in a cluster. Storm offers this and additional finer-grained ways to increase the parallelism of a Storm topology:

- 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

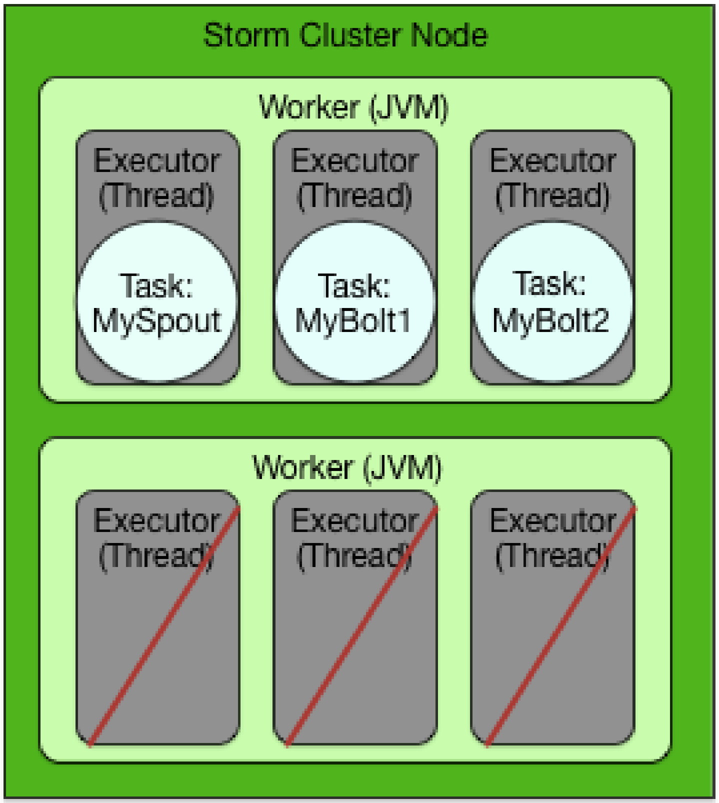
Hortonworks recommends that Storm developers store parallelism settings in a configuration file read by the topology at runtime rather than hard-coding the values passed to the Parallelism API. This topic describes and illustrates the use of the API, but developers can achieve the same effect by reading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

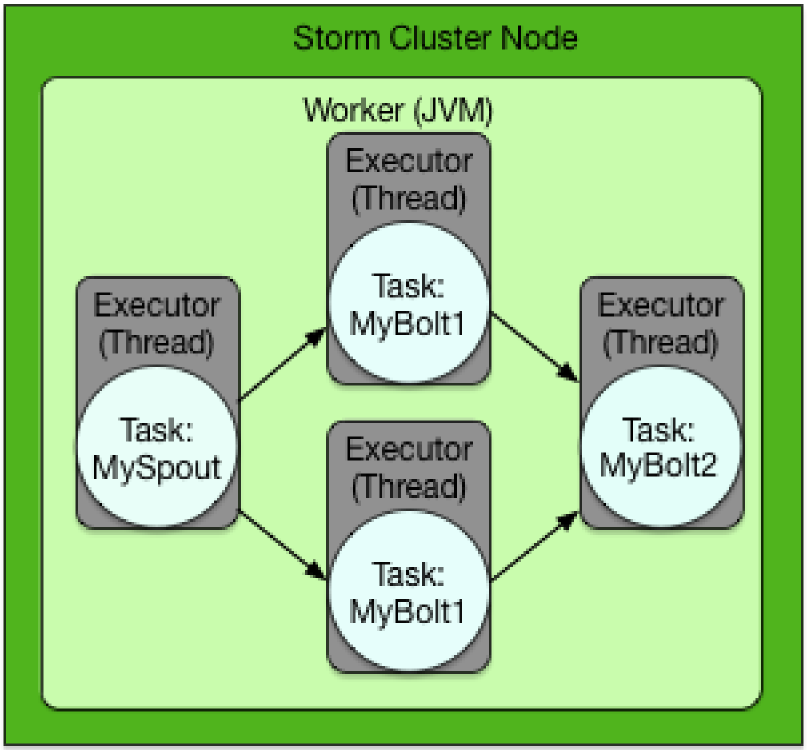
This example adds an additional worker without additional executors or tasks, but to take full advantage of this feature, Storm developers must add executors and tasks to the additional JVMs (described in the following examples).

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.



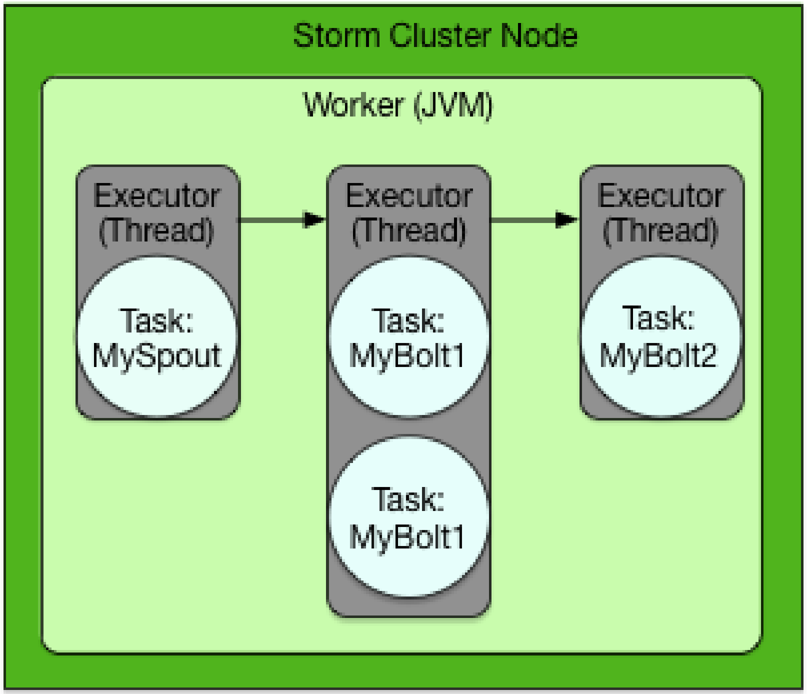
The number of executors is set at the level of individual topology components, so adding executors affects the code for the specified spouts and bolts. This differs from adding workers, which affects only the configuration of the topology.

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.

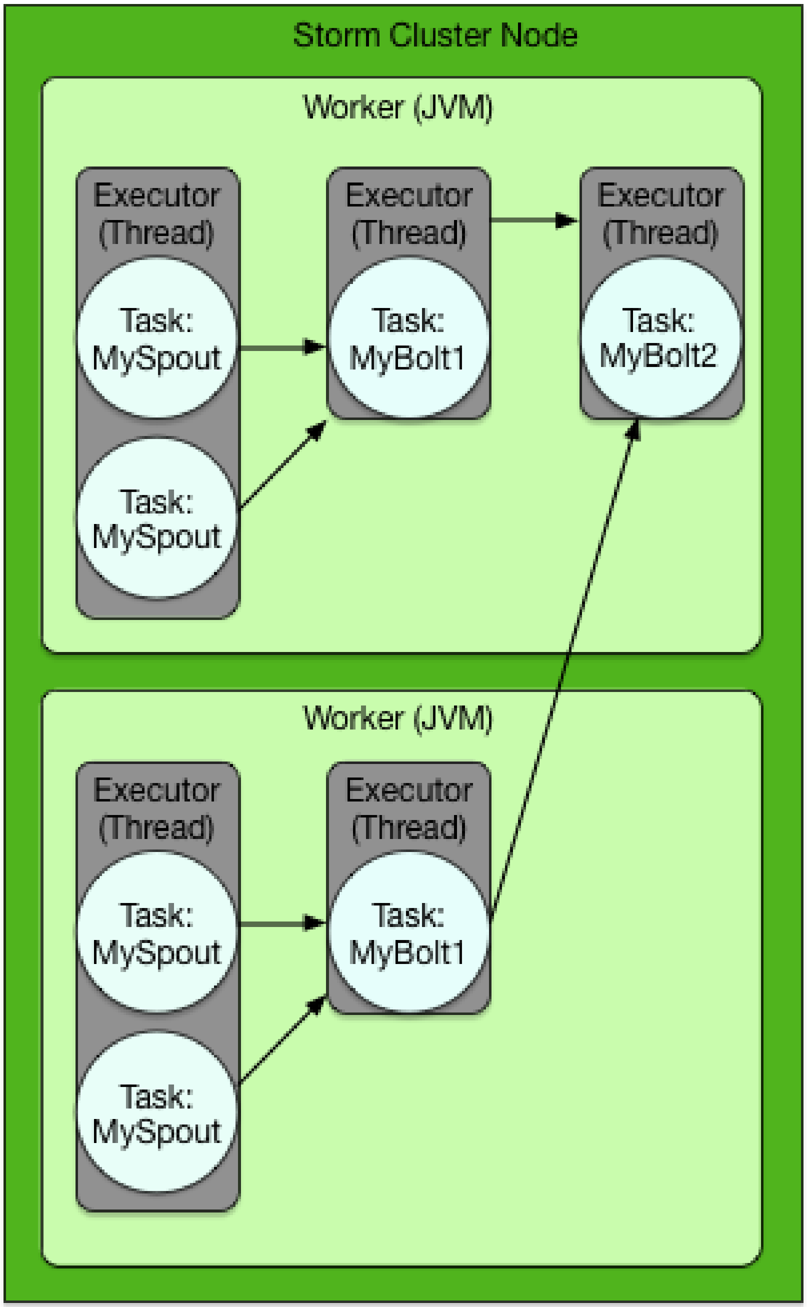


Putting it All Together

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

See the Storm javadocs at http://storm.apache.org/releases/1.0.1/javadocs/index.html for more information about the Storm API.

4.1.8. Core Storm Example: RollingTopWords Topology

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

```
package storm.starter;

import org.apache.storm.Config;
import org.apache.storm.testing.TestWordSpout;
import org.apache.storm.topology.TopologyBuilder;
import org.apache.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();
 }
}
```

4.2. Trident Concepts

Trident is a high-level API built on top of Storm core primitives (spouts and bolts). Trident provides join operations, aggregations, grouping, functions, and filters, as well as fault-tolerant state management. With Trident it is possible to achieve exactly-once processing semantics more easily than with the Storm core API.

In contrast to the Storm core API, Trident topologies process data in micro-batches. The micro-batch approach provides greater overall throughput at the cost of a slight increase in overall latency.

Because Trident APIs are built on top of Storm core API, Trident topologies compile to a graph of spouts and bolts.

The Trident API is built into Apache Storm, and does not require any additional configuration or dependencies.

4.2.1. Introductory Example: Trident Word Count

The following code sample illustrates how to implement a simple word count program using the Trident API:

```
TridentTopology topology = new TridentTopology();
    Stream wordCounts = topology.newStream("spout1", spout)
           .each(new Fields("sentence"), new Split(), new Fields("word"))
            .parallelismHint(16)
            .groupBy(new Fields("word"))
            .persistentAggregate(new MemoryMapState.Factory(), new Count(),
 new Fields("count"))
            .newValuesStream()
            .parallelismHint(16);
```

Here is detailed information about lines of code in the example:

- The first line creates the `TridentTopology` object that will be used to define the topology:

  ```
  TridentTopology topology = new TridentTopology();
  ```

- The second line creates a `Stream` object from a spout; it will be used to define subsequent operations to be performed on the stream of data:

  ```
  Stream wordCounts = topology.newStream("spout1", spout)
  ```

- The third line uses the `Stream.each()` method to apply the Split function on the "sentence" field, and specifies that the resulting output contains a new field named "word":

  `.each(new Fields("sentence"), new Split(), new Fields("word"))`

  The `Split` class is a simple Trident function that takes the first field of a tuple, tokenizes it on the space character, and emits resulting tokens:

```
public class Split extends BaseFunction {

  public void execute(TridentTuple tuple, TridentCollector collector) {
   String sentence = tuple.getString(0);
   for (String word : sentence.split(" ")) {
    collector.emit(new Values(word));
   }
  }
 }
```

- The next two lines set the parallelism of the `Split` function and apply a `groupBy()` operation to ensure that all tuples with the same "word" value are grouped together in subsequent operations.

  Calling `parallelismHint()` before a partitioning operation applies the specified parallelism value on the resulting bolt:

  `.parallelismHint(16)`

  The `groupBy()` operation is a partitioning operation; it forms the boundary between separate bolts in the resulting topology:

  `.groupBy(new Fields("word"))`

  The `groupBy()` operation results in batches of tuples being repartitioned by the value of the "word" field.

  For more information about stream operations that support partitioning, see the Stream JavaDoc.

- The remaining lines of code aggregate the running count for individual words, update a persistent state store, and emit the current count for each word.

  The `persistentAggregate()` method applies a Trident Aggregator to a stream, updates a persistent state store with the result of the aggregation, and emits the result:

  `.persistentAggregate(new MemoryMapState.Factory(), new Count(), new Fields("count"))`

  The sample code uses an in-memory state store (`MemoryMapState`); Storm comes with a number of state implementations for databases such as HBase.

  The `Count` class is a Trident `CombinerAggregator` implementation that sums all values in a batch partition of tuples:

```
public class Count implements CombinerAggregator<Long> {

    public Long init(TridentTuple tuple) {
        return 1L;
    }
    public Long combine(Long val1, Long val2) {
        return val1 + val2;
    }
    public Long zero() {
        return 0L;
    }
}
```

When applying the aggregator, Storm passes grouped partitions to the aggregator, calling `init()` for each tuple. It calls `combine()` repeatedly to process all tuples in the partition. When finished, the last value returned by `combine()` is used. If the partition is empty, the value of `zero()` is used.

The call to `newValuesStream()` tells Storm to emit the result of the persistent aggregation. This consists of a stream of individual word counts. The resulting stream can be reused in other parts of a topology.

4.2.2. Trident Operations

The Trident Stream class provides a number of methods that modify the content of a stream. The `Stream.each()` method is overloaded to allow the application of two types of operations: filters and functions.

For a complete list of methods in the Stream class, see the Trident JavaDoc.

4.2.2.1. Filters

Trident filters provide a way to exclude tuples from a Stream based on specific criteria. Implementing a Trident filter involves extending `BaseFilter` and implementing the `isKeep()` method of the Filter interface:

```
boolean isKeep(TridentTuple tuple);
```

The `isKeep()` method takes a TridentTuple as input and returns a boolean. If `isKeep()` returns `false`, the tuple is dropped from the stream; otherwise the tuple is kept.

For example, to exclude words with fewer than three characters from the word count, you could apply the following filter implementation to the stream:

```
public class ShortWordFilter extends BaseFilter {

          public boolean isKeep(TridentTuple tuple) {
              String word = tuple.getString(0);
              return word.length() > 3;
          }
      }
```

4.2.2.2. Functions

Trident functions are similar to Storm bolts, in that they consume individual tuples and optionally emit new tuples. An important difference is that tuples emitted by Trident functions are additive. Fields emitted by Trident functions are added to the tuple and existing fields are retained. The `Split` function in the word count example illustrates a function that emits additional tuples:

```
  public class Split extends BaseFunction {

    public void execute(TridentTuple tuple, TridentCollector collector) {
      String sentence = tuple.getString(0);
      for (String word : sentence.split(" ")) {
        collector.emit(new Values(word));
      }
    }
  }
```

Note that the `Split` function always processes the first (index 0) field in the tuple. It guarantees this because of the way that the function was applied using the `Stream.each()` method:

```
stream.each(new Fields("sentence"), new Split(), new Fields("word"))
```

The first argument to the `each()` method can be thought of as a field selector. Specifying “sentence” tells Trident to select only that field for processing, thus guaranteeing that the “sentence” field will be at index 0 in the tuple.

Similarly, the third argument names the fields emitted by the function. This behavior allows both filters and functions to be implemented in a more generic way, without depending on specific field naming conventions.

4.2.3. Trident Aggregations

In addition to functions and filters, Trident defines a number of aggregator interfaces that allow topologies to combine tuples.

There are three types of Trident aggregators:

- `CombinerAggregator`
- `ReducerAggregator`
- `Aggregator`

As with functions and filters, Trident aggregations are applied to streams via methods in the Stream class, namely `aggregate()`, `partitionAggregate()`, and `persistentAggregate()`.

4.2.3.1. CombinerAggregator

The `CombinerAggregator` interface is used to combine a set of tuples into a single field. In the word count example the Count class is an example of a CombinerAggregator that summed field values across a partition. The CombinerAggregator interface is as follows:

```
public interface CombinerAggregator<T> extends Serializable {
    T init(TridentTuple tuple);
    T combine(T val1, T val2);
    T zero();
}
```

When executing Aggregator, Storm calls init() for each tuple, and calls combine() repeatedly to process each tuple in the partition.

When complete, the last value returned by combine() is emitted. If the partition is empty, the value of zero() will be emitted.

4.2.3.2. ReducerAggregator

The ReducerAggregator interface has the following interface definition:

```
public interface ReducerAggregator<T> extends Serializable {
    T init();
    T reduce(T curr, TridentTuple tuple);
}
```

When applying a ReducerAggregator to a partition, Storm first calls the init() method to obtain an initial value. It then calls the reduce() method repeatedly, to process each tuple in the partition. The first argument to the reduce() method is the current cumulative aggregation, which the method returns after applying the tuple to the aggregation. When all tuples in the partition have been processed, Storm emits the last value returned by reduce().

4.2.3.3. Aggregator

The Aggregator interface represents the most general form of aggregation operations:

```
public interface Aggregator<T> extends Operation {
    T init(Object batchId, TridentCollector collector);
    void aggregate(T val, TridentTuple tuple, TridentCollector collector);
    void complete(T val, TridentCollector collector);
}
```

A key difference between Aggregator and other Trident aggregation interfaces is that an instance of TridentCollector is passed as a parameter to every method. This allows Aggregator implementations to emit tuples at any time during execution.

Storm executes Aggregator instances as follows:

1. Storm calls the init() method, which returns an object T representing the initial state of the aggregation.

   T is also passed to the aggregate() and complete() methods.

2. Storm calls the aggregate() method repeatedly, to process each tuple in the batch.

3. Storm calls complete() with the final value of the aggregation.

The word count example uses the built-in Count class that implements the CombinerAggregator interface. The Count class could also be implemented as an Aggregator:

```
public class Count extends BaseAggregator<CountState> {
    static class CountState {
        long count = 0;
    }

    public CountState init(Object batchId, TridentCollector collector) {
        return new CountState();
    }

    public void aggregate(CountState state, TridentTuple tuple,
 TridentCollector collector) {
        state.count+=1;
    }

    public void complete(CountState state, TridentCollector collector) {
        collector.emit(new Values(state.count));
    }
}
```

4.2.4. Trident State

Trident includes high-level abstractions for managing persistent state in a topology. State management is fault tolerant: updates are idempotent when failures and retries occur. These properties can be combined to achieve exactly-once processing semantics. Implementing persistent state with the Storm core API would be more difficult.

Trident groups tuples into batches, each of which is given a unique transaction ID. When a batch is replayed, the batch is given the same transaction ID. State updates in Trident are ordered such that a state update for a particular batch will not take place until the state update for the previous batch is fully processed. This is reflected in Tridents State interface at the center of the state management API:

```
public interface State {
    void beginCommit(Long txid);
    void commit(Long txid);
}
```

When updating state, Trident informs the `State` implementation that a transaction is about to begin by calling `beginCommit()`, indicating that state updates can proceed. At that point the `State` implementation updates state as a batch operation. Finally, when the state update is complete, Trident calls the `commit()` method, indicating that the state update is ending. The inclusion of transaction ID in both methods allows the underlying implementation to manage any necessary rollbacks if a failure occurs.

Implementing Trident states against various data stores is beyond the scope of this document, but more information can be found in the Trident State documentation(https://storm.apache.org/releases/1.0.1/Trident-state.html).

4.2.4.1. Trident Spouts

Trident defines three spout types that differ with respect to batch content, failure response, and support for exactly-once semantics:

Non-transactional spouts

Non-transactional spouts make no guarantees for the contents of each batch. As a result, processing may be at-most-once or at least once. It is not possible to achieve exactly-once processing when using non-transactional Trident spouts.

Transactional spouts

Transactional spouts support exactly-once processing in a Trident topology. They define success at the batch level, and have several important properties that allow them to accomplish this:

1. Batches with a given transaction ID are always identical in terms of tuple content, even when replayed.
2. Batch content never overlaps. A tuple can never be in more than one batch.
3. Tuples are never skipped.

With transactional spouts, idempotent state updates are relatively easy: because batch transaction IDs are strongly ordered, the ID can be used to track data that has already been persisted. For example, if the current transaction ID is 5 and the data store contains a value for ID 5, the update can be safely skipped.

Opaque transactional spouts

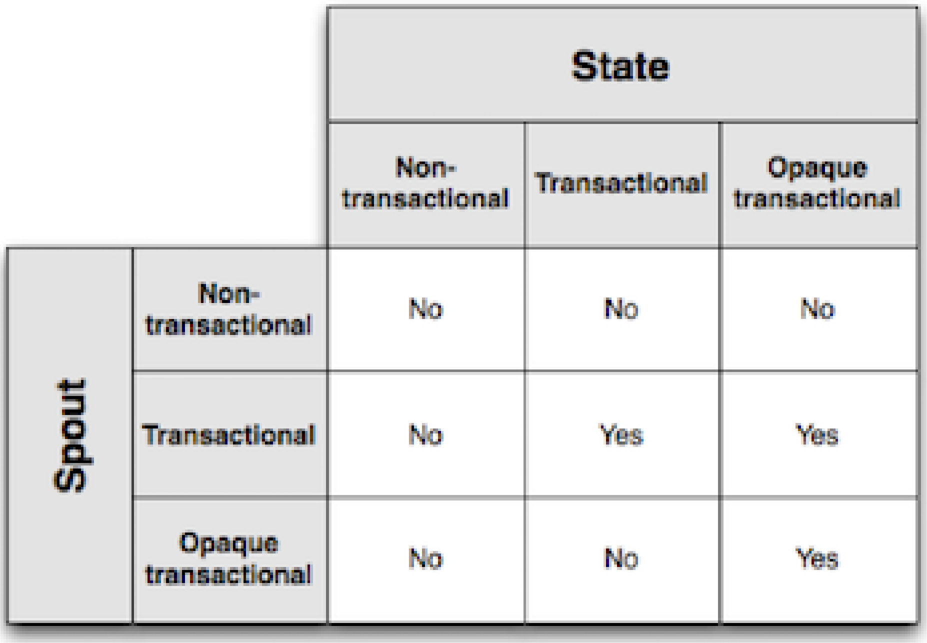
Opaque transactional spouts define success at the tuple level. Opaque transactional spouts have the following properties:

1. There is no guarantee that a batch for a particular transaction ID is always the same.
2. Each tuple is successfully processed in exactly one batch, though it is possible for a tuple to fail in one batch and succeed in another.

The difference in focus between transactional and opaque transactional spouts—success at the batch level versus the tuple level, respectively—has key implications in terms of achieving exactly-once semantics when combining different spouts with different state types.

4.2.4.2. Achieving Exactly-Once Messaging in Trident

As mentioned earlier, achieving exactly-once semantics in a Trident topology require certain combinations of spout and state types. It should also be clear why exactly-once guarantees are not possible with non-transactional spouts and states. The table below illustrates which combinations of spouts and states support exactly-once processing:



		State		
		Non-transactional	Transactional	Opaque transactional
Spout	Non-transactional	No	No	No
	Transactional	No	Yes	Yes
	Opaque transactional	No	No	Yes

4.2.5. Further Reading about Trident

For additional information about Trident, refer to the following documents:

Trident Tutorial

Trident API Overview

Trident State

Trident Spouts

4.3. Moving Data Into and Out of a Storm Topology

There are two approaches for moving data into and out of a Storm topology:

- Use a spout or bolt connector to ingest or write streaming data from or to a component such as Kafka, HDFS or HBase. For more information, see Moving Data Into and Out of Apache Storm Using Spouts and Bolts.

- Use the core Storm or Trident APIs to write a spout or bolt.

4.4. Implementing Windowing Computations on Data Streams

Windowing is one of the most frequently used processing methods for streams of data. An unbounded stream of data (events) is split into finite sets, or *windows*, based on specified criteria, such as time. A window can be conceptualized as an in-memory table in which events are added and removed based on a set of policies. Storm performs computations on each window of events. An example would be to compute the top trending Twitter topic every hour.

You can use high-level abstractions to define a window in a Storm topology, and you can use stateful computation in conjunction with windowing. For more information, see Implementing State Management.

This chapter includes examples that implement windowing features. For more information about interfaces and classes, refer to the Storm 1.1.0 javadocs.

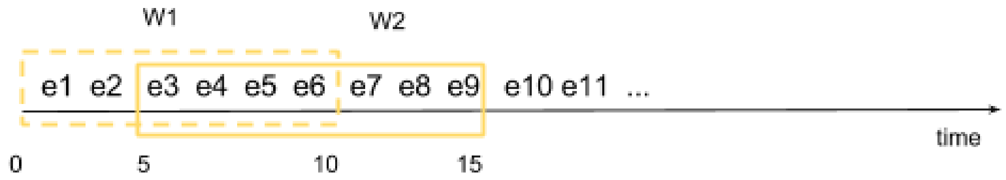
4.4.1. Understanding Sliding and Tumbling Windows

This subsection describes how sliding and tumbling windows work. Both types of windows move across continuous streaming data, splitting the data into finite sets. Finite windows are helpful for operations such as aggregations, joins, and pattern matching.

Sliding Windows

In a *sliding window*, tuples are grouped within a window that slides across the data stream according to a specified interval. A time-based sliding window with a length of ten seconds and a sliding interval of five seconds contains tuples that arrive within a ten-second window. The set of tuples within the window are evaluated every five seconds. Sliding windows can contain overlapping data; an event can belong to more than one sliding window.

In the following image, the first window (w1, in the box with dashed lines) contains events that arrived between the zeroth and tenth seconds. The second window (w2, in the box with solid lines) contains events that arrived between the fifth and fifteenth seconds. Note that events e3 through e6 are in both windows. When window w2 is evaluated at time t = 15 seconds, events e1 and e2 are dropped from the event queue.

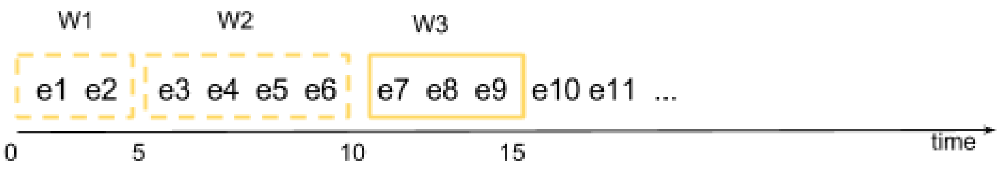


An example would be to compute the moving average of a stock price across the last five minutes, triggered every second.

Tumbling Windows

In a *tumbling window*, tuples are grouped in a single window based on time or count. A tuple belongs to only one window.

For example, consider a time-based tumbling window with a length of five seconds. The first window (w1) contains events that arrived between the zeroth and fifth seconds. The second window (w2) contains events that arrived between the fifth and tenth seconds, and the third window (w3) contains events that arrived between tenth and fifteenth seconds. The tumbling window is evaluated every five seconds, and none of the windows overlap; each segment represents a distinct time segment.



An example would be to compute the average price of a stock over the last five minutes, computed every five minutes.

4.4.2. Implementing Windowing in Core Storm

If you want to use windowing in a bolt, you can implement the bolt interface `IWindowedBolt`:

```
public interface IWindowedBolt extends IComponent {
   void prepare(Map stormConf, TopologyContext context, OutputCollector
 collector);
   /**
     * Process tuples falling within the window and optionally emit
     * new tuples based on the tuples in the input window.
     */
   void execute(TupleWindow inputWindow);
   void cleanup();
}
```

Every time the window slides (the sliding interval elapses), Storm invokes the `execute` method.

You can use the `TupleWindow` parameter to access current tuples in the window, expired tuples, and tuples added since the window was last computed. You can use this information to optimize the efficiency of windowing computations.

Bolts that need windowing support would typically extend `BaseWindowedBolt`, which has APIs for specifying type of window, window length, and sliding interval:

```
public class SlidingWindowBolt extends BaseWindowedBolt {
   private OutputCollector collector;
   @Override
   public void prepare(Map stormConf, TopologyContext context, OutputCollector
 collector){
       this.collector = collector;
   }
   @Override
   public void execute(TupleWindow inputWindow) {
     for(Tuple tuple: inputWindow.get()) {
       // do the windowing computation
          ...
     }
     collector.emit(new Values(computedValue));
   }
}
```

You can specify window length and sliding interval as a count of the number of tuples, a duration of time, or both. The following window configuration settings are supported:

```
/*
 * Tuple count based sliding window that slides after slidingInterval number
 of tuples
 */
withWindow(Count windowLength, Count slidingInterval)

/*
 * Tuple count based window that slides with every incoming tuple
 */
withWindow(Count windowLength)

/*
 * Tuple count based sliding window that slides after slidingInterval time
 duration
 */
withWindow(Count windowLength, Duration slidingInterval)

/*
 * Time duration based sliding window that slides after slidingInterval time
 duration
 */
withWindow(Duration windowLength, Duration slidingInterval)

/*
 * Time duration based window that slides with every incoming tuple
 */
withWindow(Duration windowLength)

/*
 * Time duration based sliding window that slides after slidingInterval number
 of tuples
 */
withWindow(Duration windowLength, Count slidingInterval)

/*
 * Count based tumbling window that tumbles after the specified count of
 tuples
 */
withTumblingWindow(BaseWindowedBolt.Count count)

/*
 * Time duration based tumbling window that tumbles after the specified time
 duration
 */
withTumblingWindow(BaseWindowedBolt.Duration duration)
```

To add windowed bolts to the topology, use the TopologyBuilder (as you would with non-windowed bolts):

```
TopologyBuilder builder = new TopologyBuilder();
/*
 * A windowed bolt that computes sum over a sliding window with window length
 of
 * 30 events that slides after every 10 events.
 */
builder.setBolt("sum", new WindowSumBolt().withWindow(Count.of(30), Count.
of(10)), 1)
       .shuffleGrouping("spout");
```

For a sample topology that shows how to use the APIs to compute a sliding window sum and a tumbling window average, see the `SlidingWindowTopology.java` file in the `storm-starter` GitHub directory.

For examples of tumbling and sliding windows, see the Apache document Windowing Support in Core Storm.

The following subsections describe additional aspects of windowing calculations: timestamps, watermarks, guarantees, and state management.

4.4.2.1. Understanding Tuple Timestamps and Out-of-Order Tuples

By default, window calculations are performed based on the processing timestamp. The timestamp tracked in each window is the time when the tuple is processed by the bolt.

Storm can also track windows by source-generated timestamp. This can be useful for processing events based on the time that an event occurs, such as log entries with timestamps.

The following example specifies a source-generated timestamp field. The value for `fieldName` is retrieved from the incoming tuple, and then considered for use in windowing calculations.

When this option is specified, all tuples are expected to contain the timestamp field.

```
/**
 * Specify the tuple field that represents the timestamp as a long value. If
 this field
 * is not present in the incoming tuple, an {@link IllegalArgumentException}
 will be thrown.
 *
 * @param fieldName the name of the field that contains the timestamp
 */
public BaseWindowedBolt withTimestampField(String fieldName)
```

Note: If the timestamp field is not present in the tuple, an exception is thrown and the topology terminates. To resolve this issue, remove the erroneous tuple manually from the source (such as Kafka), and then restart the topology.

In addition to using the timestamp field to trigger calculations, you can specify a time lag parameter that indicates the maximum time limit for tuples with out-of-order timestamps:

```
/**
 * Specify the maximum time lag of the tuple timestamp in millis. The tuple
 timestamps
 * cannot be out of order by more than this amount.
 *
 * @param duration the max lag duration
 */
public BaseWindowedBolt withLag(Duration duration)
```

For example, if the lag is five seconds and tuple t1 arrives with timestamp 06:00:05, no tuples can arrive with tuple timestamps earlier than 06:00:00. If a tuple arrives with timestamp 05:59:59 after t1 and the window has moved past t1, the tuple is considered late and is not processed; late tuples are ignored and are logged in the worker log files at the *INFO* level.

4.4.2.2. Understanding Watermarks

When processing tuples using a timestamp field, Storm computes watermarks based on the timestamp of an incoming tuple. Each watermark is the minimum of the latest tuple timestamps (minus the lag) across all the input streams. At a higher level, this is similar to the watermark concept used by Google's MillWheel for tracking event-based timestamps.

Periodically (by default, every second), Storm emits watermark timestamps, which are used as the “clock tick” for the window calculation when tuple-based timestamps are in use. You can change the interval at which watermarks are emitted by using the following API:

```
/**
 * Specify the watermark event generation interval. Watermark events
 * are used to track the progress of time
 *
 * @param interval the interval at which watermark events are generated
 */
public BaseWindowedBolt withWatermarkInterval(Duration interval)
```

When a watermark is received, all windows up to that timestamp are evaluated.

For example, consider tuple timestamp-based processing with the following window parameters:

- Window length equals 20 seconds, sliding interval equals 10 seconds, watermark emit frequency equals 1 second, max lag equals 5 seconds.
- Current timestamp equals 09:00:00.
- Tuples e1(6:00:03), e2(6:00:05), e3(6:00:07), e4(6:00:18), e5(6:00:26), e6(6:00:36) arrive between 9:00:00 and 9:00:01.

At time t equals 09:00:01, the following actions occur:

1. Storm emits watermark w1 at 6:00:31, because no tuples earlier than 6:00:31 can arrive.
2. Three windows are evaluated.

   The first window ending timestamp (06:00:10) is computed by taking the earliest event timestamp (06:00:03) and computing the duration based on the sliding interval (10 seconds):

   - 5:59:50 to 06:00:10 with tuples e1, e2, e3
   - 6:00:00 to 06:00:20 with tuples e1, e2, e3, e4
   - 6:00:10 to 06:00:30 with tuples e4, e5
3. Tuple e6 is not evaluated, because watermark timestamp 6:00:31 is less than tuple timestamp 6:00:36.
4. Tuples e7(8:00:25), e8(8:00:26), e9(8:00:27), e10(8:00:39) arrive between 9:00:01 and 9:00:02.

At time t equals 09:00:02, the following actions occur:

1. Storm emits watermark w2 at 08:00:34, because no tuples earlier than 8:00:34 can arrive.
2. Three windows are evaluated:

- 6:00:20 to 06:00:40, with tuples e5 and e6 (from an earlier batch)
- 6:00:30 to 06:00:50, with tuple e6 (from an earlier batch)
- 8:00:10 to 08:00:30, with tuples e7, e8, and e9

3. Tuple e10 is not evaluated, because the tuple timestamp 8:00:39 is beyond the watermark time 8:00:34.

The window calculation considers the time gaps and computes the windows based on the tuple timestamp.

4.4.2.3. Understanding the “at-least-once” Guarantee

The windowing functionality in Storm core provides an “at-least-once” guarantee. Values emitted from a bolt’s `execute(TupleWindow inputWindow)` method are automatically anchored to all tuples in `inputWindow`. Downstream bolts are expected to acknowledge the received tuple (the tuple emitted from the windowed bolt) to complete the tuple tree. If not acknowledged, the tuples are replayed and the windowing computation is reevaluated.

Tuples in a window are automatically acknowledged when they exit the window after `windowLength + slidingInterval`. Note that the configuration `topology.message.timeout.secs` should be more than `windowLength + slidingInterval` for time-based windows; otherwise, the tuples expire and are replayed, which can result in duplicate evaluations. For count-based windows, you should adjust the configuration so that `windowLength + slidingInterval` tuples can be received within the timeout period.

4.4.2.4. Saving the Window State

One issue with windowing is that tuples cannot be acknowledged until they exit the window.

For example, consider a one-hour window that slides every minute. The tuples in the window are evaluated (passed to the `bolt` execute method) every minute, but tuples that arrived during the first minute are acknowledged only after one hour and one minute. If there is a system outage after one hour, Storm replays all tuples from the starting point through the sixtieth minute. The bolt’s execute method is invoked with the same set of tuples 60 times; every window is reevaluated. One way to avoid this is to track tuples that have already been evaluated, save this information in an external durable location, and use this information to trim duplicate window evaluation during recovery.

For more information about state management and how it can be used to avoid duplicate window evaluations, see Implementing State Management.

4.4.3. Implementing Windowing in Trident

Trident processes a stream in batches of tuples for a defined topology. As with core Storm, Trident supports tumbling and sliding windows. Either type of window can be based on processing time, tuple count, or both.

Windowing API for Trident

The common windowing API takes `WindowConfig` for any supported windowing configuration. It returns a stream of aggregated results based on the given window configuration.

```
public Stream window(WindowConfig windowConfig,
                   Fields inputFields,
                   Aggregator aggregator,
                   Fields functionFields)
```

`windowConfig` can be any of the following:

- `SlidingCountWindow of(int windowCount, int slidingCount)`
- `SlidingDurationWindow of(BaseWindowedBolt.Duration windowDuration,`

  `BaseWindowedBolt.Duration slidingDuration)`
- `TumblingCountWindow of(int windowLength)`
- `TumblingDurationWindow of(BaseWindowedBolt.Duration windowLength)`

Trident windowing APIs also need to implement `WindowsStoreFactory`, to store received tuples and aggregated values.

Implementing a Tumbling Window

For a tumbling window implementation, tuples are grouped in a single window based on processing time or count. Any tuple belongs to only one window. Here is the API for a tumbling window:

```
    /**
     * Returns a stream of tuples which are aggregated results of a tumbling
 window with
            every {@code windowCount} of tuples.
     */
public Stream tumblingWindow(int windowCount,
                          WindowsStoreFactory windowStoreFactory,
                          Fields inputFields,
                          Aggregator aggregator,
                          Fields functionFields)
    /**
     * Returns a stream of tuples which are aggregated results of a window
 that tumbles at
            duration of {@code windowDuration}
     */

public Stream tumblingWindow(BaseWindowedBolt.Duration windowDuration,
                          WindowsStoreFactory windowStoreFactory,
                          Fields inputFields,
                          Aggregator aggregator,
                          Fields functionFields)
```

Implementing a Sliding Window

For a sliding window implementation, tuples are grouped in windows that slide for every sliding interval. A tuple can belong to more than one window. Here is the API for a sliding window:

```
    /**
```

```
     * Returns a stream of tuples which are aggregated results of a sliding
 window with
            every {@code windowCount} of tuples and slides the window after
 {@code slideCount}.
     */
public Stream slidingWindow(int windowCount,
                          int slideCount,
                          WindowsStoreFactory windowStoreFactory,
                          Fields inputFields,
                          Aggregator aggregator,
                          Fields functionFields)
/**
* Returns a stream of tuples which are aggregated results of a window which
 slides at
            duration of {@code slidingInterval}
* and completes a window at {@code windowDuration}
*/
                          public Stream slidingWindow( BaseWindowedBolt.
Duration windowDuration,
                          BaseWindowedBolt.Duration slidingInterval,
                          WindowsStoreFactory windowStoreFactory,
                          Fields inputFields,
                          Aggregator aggregator,
                          Fields functionFields)
```

4.4.3.1. Trident Windowing Implementation Details

For information about `org.apache.storm.trident.Stream`, see the Apache javadoc for Trident streams.

The following example shows a basic implementation of `WindowStoreFactory` for HBase, using `HBaseWindowsStoreFactory` and `HBaseWindowsStore`. It can be extended to address other use cases.

```
/**
 * Factory to create instances of {@code WindowsStore}.
 */
public interface WindowsStoreFactory extends Serializable {
       public WindowsStore create();
}

    /**
     * Store for storing window related entities like windowed tuples,
 triggers etc.
     *
     */
public interface WindowsStore extends Serializable {

public Object get(String key);

public Iterable<Object> get(List<String> keys);

public Iterable<String> getAllKeys();

public void put(String key, Object value);

public void putAll(Collection<Entry> entries);

public void remove(String key);
public void removeAll(Collection<String> keys);

public void shutdown();

    /**
     * This class wraps key and value objects which can be passed to {@code
 putAll} method.
     */
public static class Entry implements Serializable {
    public final String key;
    public final Object value;
...
}
```

A windowing operation in a Trident stream is a `TridentProcessor` implementation with the following lifecycle for each batch of tuples received:

```
// This is invoked when a new batch of tuples is received.
void startBatch(ProcessorContext processorContext);

// This is invoked for each tuple of a batch.
void execute(ProcessorContext processorContext, String streamId, TridentTuple
 tuple);

// This is invoked for a batch to make it complete. All the tuples of this
 batch
would have been already invoked with #execute(ProcessorContext
 processorContext, String streamId, TridentTuple tuple)
void finishBatch(ProcessorContext processorContext);
```

Each tuple is received in window operation through `WindowTridentProcessor#execute ( ProcessorContext processorContext, String streamId, TridentTuple tuple)`. These tuples are accumulated for each batch.

When a batch is finished, associated tuple information is added to the window, and tuples are saved in the configured `WindowsStore`. Bolts for respective window operations fire a trigger according to the specified windowing configuration (like tumbling/sliding count or time). These triggers compute the aggregated result according to the given Aggregator. Results are emitted as part of the current batch, if it exists.

When a trigger is fired outside `WindowTridentProcessor#finishBatch` invocation, those triggers are stored in the given `WindowsStore`, and are emitted as part of the next immediate batch from that window's processor.

4.4.3.2. Sample Trident Application with Windowing

Here is an example that uses `HBaseWindowStoreFactory` for windowing:

```
// define arguments
Map<String, Object> config = new HashMap<>();
String tableName = "window-state";
byte[] columnFamily = “cf”.getBytes(“UTF-8”);
byte[] columnQualifier = “tuples”.getBytes(“UTF-8”);

// window-state table should already be created with cf:tuples column
    HBaseWindowsStoreFactory windowStoreFactory = new
 HBaseWindowsStoreFactory(config, tablename, columnFamily, columnQualifier);
```

```
    FixedBatchSpout spout = new FixedBatchSpout(new Fields("sentence"), 3, new
 Values("the cow jumped over the moon"),
            new Values("the man went to the store and bought some candy"), new
 Values("four score and seven years ago"),
            new Values("how many apples can you eat"), new Values("to be or
 not to be the person"));

    spout.setCycle(true);

    TridentTopology topology = new TridentTopology();

    Stream stream = topology.newStream("spout1", spout).parallelismHint(16).
each(new Fields("sentence"),
            new Split(), new Fields("word"))
            .tumblingWindow(1000, windowStoreFactory, new Fields("word"), new
 CountAsAggregator(), new Fields("count"))
            .peek(new Consumer() {
                @Override
                public void accept(TridentTuple input) {
                    LOG.info("Received tuple: [{}]", input);
                }
            });
    StormTopology stormTopology =  topology.build();
```

For additional examples that use Trident windowing APIs, see TridentHBaseWindowingStoreTopology and TridentWindowingInmemoryStoreTopology.

4.5. Implementing State Management

This subsection describes state management APIs and architecture for core Storm.

Stateful abstractions allow Storm bolts to store and retrieve the state of their computations. The state management framework automatically, periodically snapshots the state of bolts across a topology. There is a default in-memory-based state implementation, as well as a Redis-backed implementation that provides state persistence.

Bolts that require state to be managed and persisted by the framework should implement the `IStatefulBolt` interface or extend `BaseStatefulBolt`, and implement the `void initState(T state)` method. The `initState` method is invoked by the framework during bolt initialization. It contains the previously saved state of the bolt. Invoke `initState` after `prepare`, but before the bolt starts processing any tuples.

Currently the only supported State implementation is `KeyValueState`, which provides key-value mapping.

The following example describes how to implement a word count bolt that uses the key-value state abstraction for word counts:

```
public class WordCountBolt
  extends BaseStatefulBolt<KeyValueState<String, Integer>> {
    private KeyValueState<String,Integer> wordCounts;
    ...
    @Override
    public void initState(KeyValueState<String,Integer> state) {
      wordCounts = state;
    }
    @Override
    public void execute(Tuple tuple) {
      String word = tuple.getString(0);
      Integer count = wordCounts.get(word, 0);
      count++;
      wordCounts.put(word, count);
      collector.emit(tuple, new Values(word, count));
      collector.ack(tuple);
    }
  ...
  }
```

1. Extend the `BaseStatefulBolt` and type parameterize it with `KeyValueState`, to store the mapping of word to count.
2. In the `init` method, initialize the bolt with its previously saved state: the word count last committed by the framework during the previous run.
3. In the `execute` method, update the word count.

The framework periodically checkpoints the state of the bolt (default every second). The frequency can be changed by setting the storm config `topology.state.checkpoint.interval.ms`.

For state persistence, use a state provider that supports persistence by setting the topology.state.provider in the storm config. For example, for Redis based key-value state implementation, you can set `topology.state.provider` to `org.apache.storm.redis.state.RedisKeyValueStateProvider` in `storm.yaml`. The provider implementation .jar should be in the class path, which in this case means placing the `storm-redis-*`.jar in the `extlib` directory.

You can override state provider properties by setting `topology.state.provider.config`. For Redis state this is a JSON configuration with the following properties:

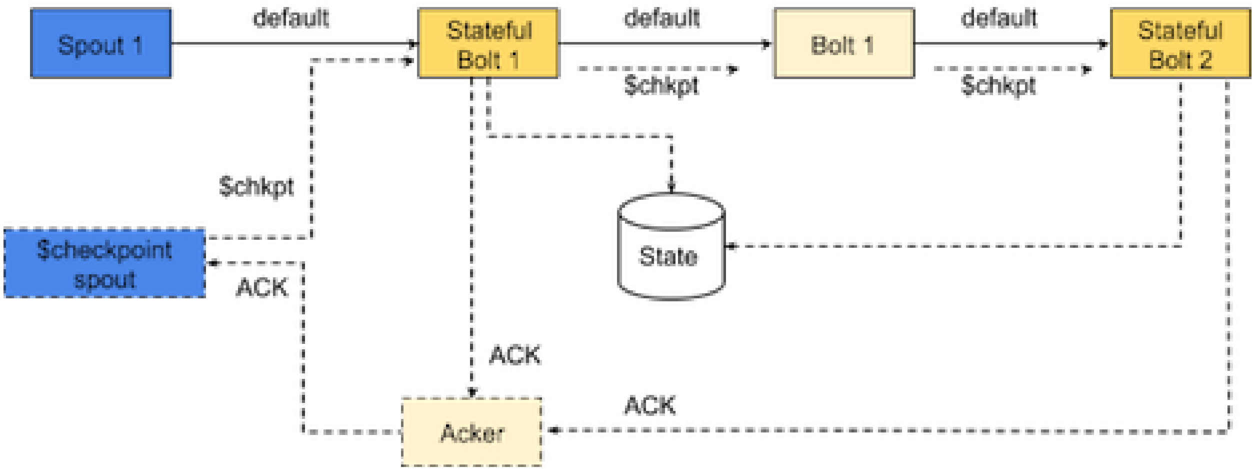
```
{
   "keyClass": "Optional fully qualified class name of the Key type.",
   "valueClass": "Optional fully qualified class name of the Value type.",
   "keySerializerClass": "Optional Key serializer implementation class.",
   "valueSerializerClass": "Optional Value Serializer implementation class.",
   "jedisPoolConfig": {
     "host": "localhost",
     "port": 6379,
     "timeout": 2000,
     "database": 0,
     "password": "xyz"
   }
}
```

4.5.1. Checkpointing

Checkpointing is triggered by an internal checkpoint spout at the interval specified by `topology.state.checkpoint.interval.ms`. If there is at least one IStatefulBolt in the topology, the checkpoint spout is automatically added by the topology builder .

For stateful topologies, the topology builder wraps the `IStatefulBolt` in a `StatefulBoltExecutor`, which handles the state commits on receiving the checkpoint tuples. Non-stateful bolts are wrapped in a `CheckpointTupleForwarder`, which simply forwards the checkpoint tuples so that the checkpoint tuples can flow through the topology directed acyclic graph (DAG).

Checkpoint tuples flow through a separate internal stream called `$checkpoint`. The topology builder wires the checkpoint stream across the whole topology, with the checkpoint spout at the root.



At specified checkpoint intervals, the checkpoint spout emits checkpoint tuples. Upon receiving a checkpoint tuple, the state of the bolt is saved and the checkpoint tuple is forwarded to the next component. Each bolt waits for the checkpoint to arrive on all of its input streams before it saves its state, so state is consistent across the topology. Once the checkpoint spout receives an ack from all bolts, the state commit is complete and the transaction is recorded as committed by the checkpoint spout.

This checkpoint mechanism builds on Storm's existing acking mechanism to replay the tuples. It uses concepts from the asynchronous snapshot algorithm used by Flink, and from the Chandy-Lamport algorithm for distributed snapshots. Internally, checkpointing uses a three-phase commit protocol with a prepare and commit phase, so that the state across the topology is saved in a consistent and atomic manner.

4.5.2. Recovery

The recovery phase is triggered for the following conditions:

- When a topology is started for the first time.
    - If the previous transaction was not prepared successfully, a rollback message is sent across the topology to indicate that if a bolt has some prepared transactions it can be discarded.
    - If the previous transaction was prepared successfully but not committed, a commit message is sent across the topology so that the prepared transactions can be committed.

  After these steps finish, bolts are initialized with the state.
- When a bolt fails to acknowledge the checkpoint message; for example, if a worker crashes during a transaction.

When the worker is restarted by the supervisor, the checkpoint mechanism ensures that the bolt is initialized with its previous state. Checkpointing continues from the point where it left off.

4.5.3. Guarantees

Storm relies on the acking mechanism to replay tuples in case of failures. It is possible that the state is committed but the worker crashes before acking the tuples. In this case the tuples are replayed causing duplicate state updates. Also currently the `StatefulBoltExecutor` continues to process the tuples from a stream after it has received a checkpoint tuple on one stream while waiting for checkpoint to arrive on other input streams for saving the state. This can also cause duplicate state updates during recovery.

The state abstraction does not eliminate duplicate evaluations and currently provides only at-least once guarantee.

To provide the at-least-once guarantee, all bolts in a stateful topology are expected to anchor the tuples while emitting and ack the input tuples once it is processed. For non-stateful bolts, the anchoring and acking can be automatically managed by extending the `BaseBasicBolt`. Stateful bolts are expected to anchor tuples while emitting and ack the tuple after processing like in the `WordCountBolt` example in the State management subsection.

4.5.4. Implementing Custom Actions: IStateful Bolt Hooks

The IStateful bolt interface provides hook methods through which stateful bolts can implement custom actions. This feature is optional; stateful bolts are not expected to provide an implementation. The feature is provided so that other system-level components can be built on top of stateful abstractions; for example, to implement actions before the state of the stateful bolt is prepared, committed or rolled back.

```
/**
* This is a hook for the component to perform some actions just before the
* framework commits its state.
*/
void preCommit(long txid);

/**
* This is a hook for the component to perform some actions just before the
* framework prepares its state.
*/
void prePrepare(long txid);

/**
* This is a hook for the component to perform some actions just before the
* framework rolls back the prepared state.
*/
void preRollback();
```

4.5.5. Implementing Custom States

Currently the only kind of State implementation supported is `KeyValueState`, which provides key-value mapping.

Custom state implementations should provide implementations for the methods defined in the `State` interface. These are the `void prepareCommit(long txid)`, `void commit(long txid)`, and `rollback()` methods. The `commit()` method is optional; it is useful if the bolt manages state on its own. This is currently used only by internal system bolts (such as `CheckpointSpout`).

`KeyValueState` implementations should also implement the methods defined in the `KeyValueState` interface.

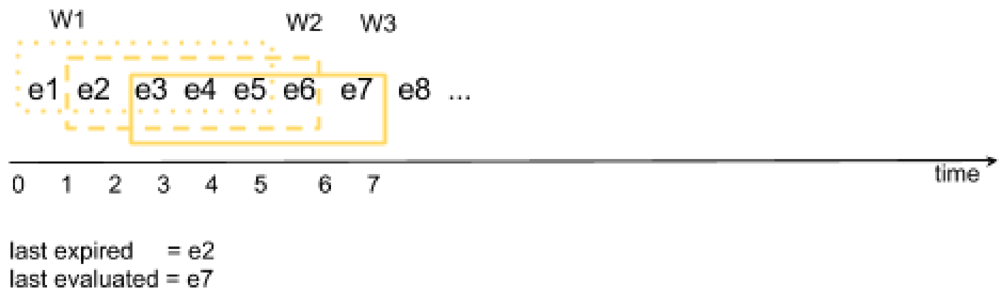
The framework instantiates state through the corresponding `StateProvider` implementation. A custom state should also provide a `StateProvider` implementation that can load and return the state based on the namespace.

Each state belongs to a unique namespace. The namespace is typically unique to a task, so that each task can have its own state. The StateProvider and corresponding State implementation should be available in the class path of Storm, by placing them in the `extlib` directory.

4.5.6. Implementing Stateful Windowing

The windowing implementation in core Storm acknowledges tuples in a window only when they fall out of the window.

For example, consider a window configuration with a window length of 5 minutes and a sliding interval of 1 minute. The tuples that arrived between 0 and 1 minutes are acked only when the window slides past one minute (for example, at the 6th minute).



If the system crashes, tuples e1 to e8 are replayed, assuming that the ack for e1 and e2 did not reach the acker. Tuples w1, w2 and w3 will be reevaluated.

Stateful windowing tries to minimize duplicate window evaluations by saving the last evaluated state and the last expired state of the window. Stateful windowing expects a monotonically increasing message ID to be part of the tuple, and uses the stateful abstractions discussed previously to save the last expired and last evaluated message IDs.

During recovery, Storm uses the last expired and last evaluated message IDs to avoid duplicate window evaluations:

- Tuples with message IDs lower than the last expired ID are discarded.
- Tuples with message IDs between the last expired and last evaluated message IDs are fed into the system without activating any triggers.

- Tuples beyond the last evaluated message ids are processed as usual.

State support in windowing is provided by `IStatefulWindowedBolt`. User bolts should typically extend `BaseStatefulWindowedBolt` for windowings operation that use the Storm framework to automatically manage the state of the window.

4.5.7. Sample Topology with Saved State

A sample topology in `storm-starter`, `StatefulWindowingTopology`, demonstrates the use of `IStatefulWindowedBolt` to save the state of a windowing operation and avoid recomputation in case of failures. The framework manages window boundaries internally; it does not invoke `execute(TupleWindow inputWindow)` for already-evaluated windows if there is a restart after a failure.

4.6. Performance Guidelines for Developing a Storm Topology

The following table lists several general performance-related guidelines for developing Storm topologies.

Table 4.4. Storm Topology Development 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.

5. Moving Data Into and Out of Apache Storm Using Spouts and Bolts

This chapter focuses on moving data into and out of Apache Storm through the use of spouts and bolts. Spouts read data from external sources to ingest data into a topology. Bolts consume input streams and process the data, emit new streams, or send results to persistent storage. This chapter focuses on bolts that move data from Storm to external sources.

The following spouts are available in HDP 2.5:

- Kafka spout based on Kafka 0.7.x/0.8.x, plus a new Kafka consumer spout available as a technical preview (not for production use)
- HDFS
- EventHubs
- Kinesis (technical preview)

The following bolts are available in HDP 2.5:

- Kafka
- HDFS
- EventHubs
- HBase
- Hive
- JDBC (supports Phoenix)
- Solr
- Cassandra
- MongoDB
- ElasticSearch
- Redis
- OpenTSDB (technical preview)

Supported 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 the Kafka spout, HDFS spout, Kafka bolt, Storm-HDFS connector, and Storm-HBase connector APIs. For information about connecting to components on a Kerberos-enabled cluster, see Configuring Connectors for a Secure Cluster.

5.1. Ingesting Data from Kafka

KafkaSpout reads from Kafka topics. To do so, it needs to connect to the Kafka broker, locate the topic from which it will read, and store consumer offset information (using the ZooKeeper root and consumer group ID). If a failure occurs, KafkaSpout can use the offset to continue reading messages from the point where the operation failed.

The `storm-kafka` components include a core Storm spout and 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

You should use the Trident API unless your application requires sub-second latency.

5.1.1. KafkaSpout Integration: Core Storm APIs

The core-storm API represents a Kafka spout with the `KafkaSpout` class.

To initialize `KafkaSpout`, define a `SpoutConfig` subclass instance of the `KafkaConfig` class, representing configuration information needed to ingest data from a Kafka cluster. `KafkaSpout` requires an instance of the `BrokerHosts` interface.

`BrokerHosts` Interface

The `BrokerHost` interface maps Kafka brokers to topic partitions. Constructors for `KafkaSpout` (and, for the Trident API, `TridentKafkaConfig`) require an implementation of the `BrokerHosts` interface.

The `storm-kafka` component provides two implementations of `BrokerHosts`, `ZkHosts` and `StaticHosts`:

- Use `ZkHosts` if you want to track broker-to-partition mapping dynamically.This class uses Kafka's ZooKeeper entries to track mapping.

  You can instantiate an object as follows:

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

  ```
  public ZkHosts(String brokerZkStr)
  ```

  where:

  - `brokerZkStr` is the `IP:port` address for the ZooKeeper host; for example, `localhost:2181`.
  - `brokerZkPath` is the root directory under which topics and partition information are stored. By default this is `/brokers`, which is the default used by Kafka.

  By default, broker-partition mapping refreshes every 60 seconds. If you want to change the refresh frequency, set `host.refreshFreqSecs` to your chosen value.

- Use `StaticHosts` for static broker-to-partition mapping. To construct an instance of this class, you must first construct an instance of `GlobalPartitionInformation`; for example:

```
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 and `SpoutConfig` Subclass

Next, define a `SpoutConfig` subclass instance of the `KafkaConfig` class.

`KafkaConfig` contains several fields used to configure the behavior of a Kafka spout in a Storm topology; `Spoutconfig` extends `KafkaConfig`, supporting additional fields for ZooKeeper connection info and for controlling behavior specific to `KafkaSpout`.

`KafkaConfig` implements 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)
```

`KafkaConfig` Parameters

`hosts` One or more hosts that are Kafka ZooKeeper broker nodes (see "`BrokerHosts` Interface").

`topic` Name of the Kafka topic that KafkaSpout will consume from.

`clientId` Optional parameter used as part of the ZooKeeper path, specifying where the spout's current offset is stored.

`KafkaConfig` Fields

`fetchSizeBytes` Number of bytes to attempt to fetch in one request to a Kafka server. The default is 1MB.

`socketTimeoutMs` Number of milliseconds to wait before a socket fails an operation with a timeout. The default value is 10 seconds.

`bufferSizeBytes` Buffer size (in bytes) for network requests. The default is 1MB.

`scheme` The interface that specifies how a `ByteBuffer` from a Kafka topic is transformed into a Storm tuple.

The default, `MultiScheme`, returns a tuple and no additional processing.

The API provides many implementations of the `Scheme` class, including:

- `storm.kafka.StringScheme`
- `storm.kafka.KeyValueSchemeAsMultiScheme`
- `storm.kafka.StringKeyValueScheme`
- `storm.kafka.KeyValueSchemeAsMultiScheme`

Important

In Apache Storm versions prior to 1.0, `MultiScheme` methods accepted a `byte[]` parameter instead of a `ByteBuffer`. In Storm version 1.0, `MultiScheme` and related scheme APIs changed; they now accept a `ByteBuffer` instead of a `byte[]`.

As a result, Kafka spouts built with Storm versions earlier than 1.0 do not work with Storm versions 1.0 and later. When running topologies with Storm version 1.0 and later, ensure that your version of `storm-kafka` is at least 1.0. Rebuild pre-1.0 shaded topology .jar files that bundle `storm-kafka` classes with `storm-kafka` version 1.0 before running them in clusters with Storm 1.0 and later.

`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 than `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.

Instantiate `SpoutConfig` as follows:

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

`SpoutConfig` Parameters

`hosts`
One or more hosts that are Kafka ZooKeeper broker nodes (see "`BrokerHosts` Interface").

`topic`
Name of the Kafka topic that KafkaSpout will consume from.

`zkroot`
Root directory in ZooKeeper under which KafkaSpout consumer offsets are stored. The default is `/brokers`.

`nodeId`
ZooKeeper node under which KafkaSpout stores offsets for each topic-partition. The node ID must be unique for each Topology. The topology uses this path to recover in failure scenarios, or when there is maintenance that requires killing the topology.

`zkroot` and `nodeId` are used to construct the ZooKeeper path where Storm stores the Kafka offset. You can find offsets at `zkroot+"/"+nodeId`.

To start processing messages from where the last operation left off, use the same `zkroot` and `nodeId`. To start from the beginning of the Kafka topic, set `KafkaConfig.ignoreZKOffsets` to `true`.

Example

The following example illustrates the use of the `KafkaSpout` class and related interfaces:

```
BrokerHosts hosts = new ZkHosts(zkConnString);
SpoutConfig spoutConfig = new SpoutConfig(hosts, topicName, "/" + zkrootDir,
 node);
spoutConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
```

5.1.2. KafkaSpout Integration: Trident APIs

The Trident API represents a Kafka spout with the `OpaqueTridentKafkaSpout` class.

To initialize `OpaqueTridentKafkaSpout`, define a `TridentKafkaConfig` subclass instance of the `KafkaConfig` class, representing configuration information needed to ingest data from a Kafka cluster.

KafkaConfig Class and TridentKafkaConfig Subclass

Both the core-storm and Trident APIs use KafkaConfig, which contains several parameters and fields used to configure the behavior of a Kafka spout in a Storm topology. For more information, see "KafkaConfig Class" in KafkaSpout Configuration Settings: Core Storm API.

Instantiate a TridentKafkaConfig subclass instance of the KafkaConfig class. Use one of the following constructors, each of which requires an implementation of the BrokerHosts interface. For more information about BrokerHosts, see "BrokerHosts Interface" in KafkaSpout Configuration Settings: Core Storm APIs.

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

TridentKafkaConfig Parameters

hosts	One or more hosts that are Kafka ZooKeeper broker nodes (see "BrokerHosts Interface").
topic	Name of the Kafka topic.
clientid	Unique identifier for this spout.

Example

The following example illustrates the use of the OpaqueTridentKafkaSpout class and related interfaces:

```
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);
```

Important

In Apache Storm versions prior to 1.0, MultiScheme methods accepted a byte[] parameter instead of a ByteBuffer. In Storm version 1.0, MultiScheme and related scheme APIs changed; they now accept a ByteBuffer instead of a byte[].

As a result, Kafka spouts built with Storm versions earlier than 1.0 do not work with Storm versions 1.0 and later. When running topologies with Storm version 1.0 and later, ensure that your version of storm-kafka is at least 1.0. Rebuild pre-1.0 shaded topology .jar files that bundle storm-kafka classes with storm-kafka version 1.0 before running them in clusters with Storm 1.0 and later.

5.1.3. Tuning KafkaSpout Performance

KafkaSpout provides two internal parameters to control performance:

- offset.commit.period.ms specifies the period of time (in milliseconds) after which the spout commits to Kafka. To set this parameter, use the KafkaSpoutConfig set method setOffsetCommitPeriodMs.

- `max.uncommitted.offsets` defines the maximum number of polled offsets (records) that can be pending commit before another poll can take place. When this limit is reached, no more offsets can be polled until the next succesful commit sets the number of pending offsets below the threshold. To set this parameter, use the KafkaSpoutConfig set method setMaxUncommittedOffsets.

Note that these two parameters trade off memory versus time:

- When `offset.commit.period.ms` is set to a low value, the spout commits to Kafka more often. When the spout is committing to Kafka, it is not fetching new records nor processing new tuples.
- When `max.uncommitted.offsets` increases, the memory footprint increases. Each offset uses eight bytes of memory, which means that a value of 10000000 (10MB) uses about 80MB of memory.

It is possible to achieve good performance with a low commit period and small memory footprint (a small value for `max.uncommitted.offsets`), as well as with a larger commit period and larger memory footprint. However, you should avoid using large values for `offset.commit.period.ms` with a low value for `max.uncommitted.offsets`.

Kafka consumer configuration parameters can also have an impact on the KafkaSpout performance. The following Kafka parameters are most likely to have the strongest impact on KafkaSpout performance:

- The Kafka Consumer poll timeout specifies the time (in milliseconds) spent polling if data is not available. To set this parameter, use the KafkaSpoutConfig set method setPollTimeoutMs.
- Kafka consumer parameter `fetch.min.bytes` specifies the minimum amount of data the server returns for a fetch request. If the minimum amount is not available, the request waits until the minimum amount accumulates before answering the request.
- Kafka consumer parameter `fetch.max.wait.ms` specifies the maximum amount of time the server will wait before answering a fetch request, when there is not sufficient data to satisfy `fetch.min.bytes`.

Important

For HDP 2.5.0 clusters in production use, you should override the default values of KafkaSpout parameters `offset.commit.period` and `max.uncommitted.offsets`, and Kafka consumer parameter `poll.timeout.ms`, as follows:

- Set `poll.timeout.ms` to `200`.
- Set `offset.commit.period.ms` to `30000` (30 seconds).
- Set `max.uncommitted.offsets` to `10000000` (ten million).

Performance also depends on the structure of your Kafka cluster, the distribution of the data, and the availability of data to poll.

Log Level Performance Impact

Storm supports several logging levels, including Trace, Debug, Info, Warn, and Error. Trace-level logging has a significant impact on performance, and should be avoided in production. The amount of log messages is proportional to the number of records fetched from Kafka, so a lot of messages are printed when Trace-level logging is enabled.

Trace-level logging is most useful for debugging pre-production environments under mild load. For debugging, if necessary, you can throttle how many messages are polled from Kafka by setting the `max.partition.fetch.bytes` parameter to a low number that is larger than than the largest single message stored in Kafka.

Logs with Debug level will have slightly less performance impact than Trace-level logs, but still generate a lot of messages. This setting can be useful for assessing whether the Kafka spout is properly tuned.

For general information about Apache Storm logging features, see Monitoring and Debugging an Apache Storm Topology.

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

Before using the `storm-kafka` connector, you must modify your Apache Kafka configuration: add a `zookeeper.connect` property, with hostnames and port numbers of HDP ZooKeeper nodes, to the Kafka `server.properties` file.

5.1.5. Configuring KafkaSpout to Connect to HBase or Hive

Before connecting to HBase or Hive, add the following exclusions to your POM file for the curator framework:

```
<exclusion>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-client</artifactId>
</exclusion>
<exclusion>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
</exclusion>
<exclusion>
  <groupId>org.apache.curator</groupId>
  <artifactId>apache-curator</artifactId>
</exclusion>
```

5.2. 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.

5.2.1. 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.

5.2.2. 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.

5.3. Streaming Data to Kafka

Storm provides a Kafka Bolt for both the core-storm and Trident APIs that publish data to Kafka topics. 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 Kafka Producer properties.

The following code samples illustrate the construction of a simple Kafka bolt.

5.3.1. KafkaBolt Integration: Core Storm APIs

To use `KafkaBolt`, create an instance of `org.apache.storm.kafka.bolt.KafkaBolt` and attach it as a component to your topology. The following example shows construction of a Kafka bolt using core Storm APIs, followed by details about the code:

```
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);
//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "1");
props.put("key.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
```

```
props.put("value.serializer", "org.apache.kafka.common.serialization.
StringSerializer");

KafkaBolt bolt = new KafkaBolt()
.withProducerProperties(props)
.withTopicSelector(new DefaultTopicSelector("test"))
.withTupleToKafkaMapper(new FieldNameBasedTupleToKafkaMapper());
builder.setBolt("forwardToKafka", bolt, 8).shuffleGrouping("spout");

Config conf = new Config();

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

1. Instantiate a `KafkaBolt`.

   The core-storm API uses the `storm.kafka.bolt.KafkaBolt` class to instantiate a Kafka Bolt:

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

2. Configure the KafkaBolt with a Tuple-to-Message Mapper.

   The `KafkaBolt` maps 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.

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

3. Configure the Kafka Bolt with a Kafka Topic Selector.

   Note

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

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

   If you need to write to multiple Kafka topics, you can write your own implementation of the `KafkaTopicSelector` interface .

4. Configure the Kafka Bolt with Kafka Producer properties.

   You can specify producer properties in your Storm topology by calling `KafkaBolt.withProducerProperties()`. See the Apache Producer Configs documentation for more information.

5.3.2. KafkaBolt Integration: Trident APIs

To use `KafkaBolt`, create an instance of `org.apache.storm.kafka.trident.TridentState` and `org.apache.storm.kafka.trident.TridentStateFactory`, and attach them to your topology. The following example shows construction of a Kafka bolt using Trident APIs, followed by details about the code:

```
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);

//set producer properties.
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("acks", "1");
props.put("key.serializer", "org.apache.kafka.common.serialization.
StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.
StringSerializer");

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

Config conf = new Config();
StormSubmitter.submitTopology("kafkaTridentTest", conf, topology.build());
```

1. Instantiate a KafkaBolt.

   The Trident API uses a combination of the `storm.kafka.trident.TridentStateFactory` and `storm.kafka.trident.TridentKafkaStateFactory` classes.

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

2. 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.

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

You 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. You can write your own implementation of the TupleToKafkaMapper and TridentTupleToKafkaMapper interfaces to customize the mapping of Storm tuples to Kafka messages. Both interfaces define two methods:

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

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

3. Configure the KafkaBolt with a Kafka Topic Selector.

Note

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

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

If you need to write to multiple Kafka topics, you can write your own implementation of the KafkaTopicSelector interface; for example:

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

4. Configure the KafkaBolt with Kafka Producer properties.

You can specify producer properties in your Storm topology by calling TridentKafkaStateFactory.withProducerProperties(). See the Apache Producer Configs documentation for more information.

5.4. 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.

5.4.1. 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.

For more information about the HdfsBolt class, refer to the Apache Storm HdfsBolt documentation.

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>
```

5.4.2. 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.

5.5. Writing Data to HBase

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.

`SimpleHBaseMapper` Methods

`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");
```

5.6. Writing Data to Hive

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. You can 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.

5.6.1. Core-storm APIs

The following example constructs a Kafka bolt using core Storm APIs:

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

1. 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 5.1. HiveMapper Arguments

Argument	Data Type	Description
withColumnFields	org.apache.storm.tuple.Fields	The name of the tuple fields that you want to map to table column names.
withPartitionFields	org.apache.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");
...
```

2. Instantiate a `HiveOptions` Class with the `HiveMapper` Implementation. The `HiveOptions` class configures transactions used by Hive to ingest the streaming data:

```
...
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 5.2. 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, 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.

HiveOptions Configuration Property	Data Type	Description
`withMaxOpenConnections`	`Integer`	Specifies the maximum number of open connections. Each connection is to a single Hive table partition. 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.

3. Instantiate the HiveBolt with the HiveOptions class:

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

4. Before building your topology code, add the following dependency to your topology `pom.xml` file:

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.3.3</version>
</dependency>
```

5.6.2. Trident APIs

The following example shows construction of a Kafka bolt using core Storm APIs, followed by details about the code:

```
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());
```

1. 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 5.3. HiveMapper Arguments

Argument	Data Type	Description
withColumnFields	org.apache.storm.tuple.Fields	The name of the tuple fields that you want to map to table column names.
withPartitionFields	org.apache.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");
   ...
   ```

2. 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);
   ...
   ```

   See "HiveOptions Class Configuration Properties" for a list of configuration properties for the `HiveOptions` class.

3. Instantiate the `HiveBolt` with the `HiveOptions` class:

   ```
   ...
   StateFactory factory = new HiveStateFactory().withOptions(hiveOptions);
   ```

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

4. Before building your topology code, add the following dependency to your topology `pom.xml` file:

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId>
  <version>4.3.3</version>
</dependency>
```

5.7. Configuring Connectors for a Secure Cluster

If your topology uses KafkaSpout, Storm-HDFS, Storm-HBase, or Storm-Hive to access components on a Kerberos-enabled cluster, complete the associated configuration steps listed in this subsection.

5.7.1. Configuring KafkaSpout for 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
       useKeyTab=true
       keyTab="/etc/security/keytabs/stormusr.service.keytab"
       storeKey=true
       useTicketCache=false
       serviceName="kafka"
       principal="stormusr/host.name@EXAMPLE.COM";
       };
```

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

```
bin/kafka-acls.sh --authorizer
kafka.security.auth.SimpleAclAuthorizer --authorizer-properties
zookeeper.connect=localhost:2181 --add --allow-principal
user:stormusr --allow-hosts * --operations Read --topic TEST
```

5.7.2. Configuring Storm-HDFS for a Secure Cluster

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.

   Note

   For more information about the `HdfsBolt` class, refer to the Apache Storm HdfsBolt API documentation.

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 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.

5.7.3. Configuring Storm-HBase for a Secure Cluster

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.

Note

For more information about the `HBaseBolt` class, refer to the Apache Storm HBaseBolt API documentation.

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.

5.7.4. Configuring Storm-Hive for a Secure Cluster

The Storm-Hive connector accepts configuration settings as part of the HiveOptions class. For more information about the HiveBolt and HiveOptions classes, see the Apache Storm HiveOptions and HiveBolt API documentation.

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)
  ```

6. 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 6.1. 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

7. Deploying and Managing Apache Storm Topologies

Use the command line interface to deploy a Storm topology after packaging it in a .jar file.

For example, you can 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
```

The remainder of this chapter describes the Storm UI, which shows diagnostics for a cluster and topologies, allowing you to monitor and manage deployed topologies.

7.1. Configuring the Storm UI

If Ambari is running on the same node as Apache Storm, ensure that Ambari and Storm use different ports. Both processes use port 8080 by default, so you might need to configure Apache Storm to use a different port.

If you want to use the Storm UI on a Kerberos-enabled cluster, ensure that your browser is configured to use authentication for the Storm UI. For example, complete the following steps to configure Firefox:

1. Go to the `about:config` configuration page.

2. Search for the `network.negotiate-auth.trusted-uris` configuration option.

3. Double-click on the option.

4. An "Enter string value" dialog box opens.

5. In this box, enter the value `http://storm-ui-hostname:8080`.

6. Click OK to finish.

7. Close and relaunch the browser.

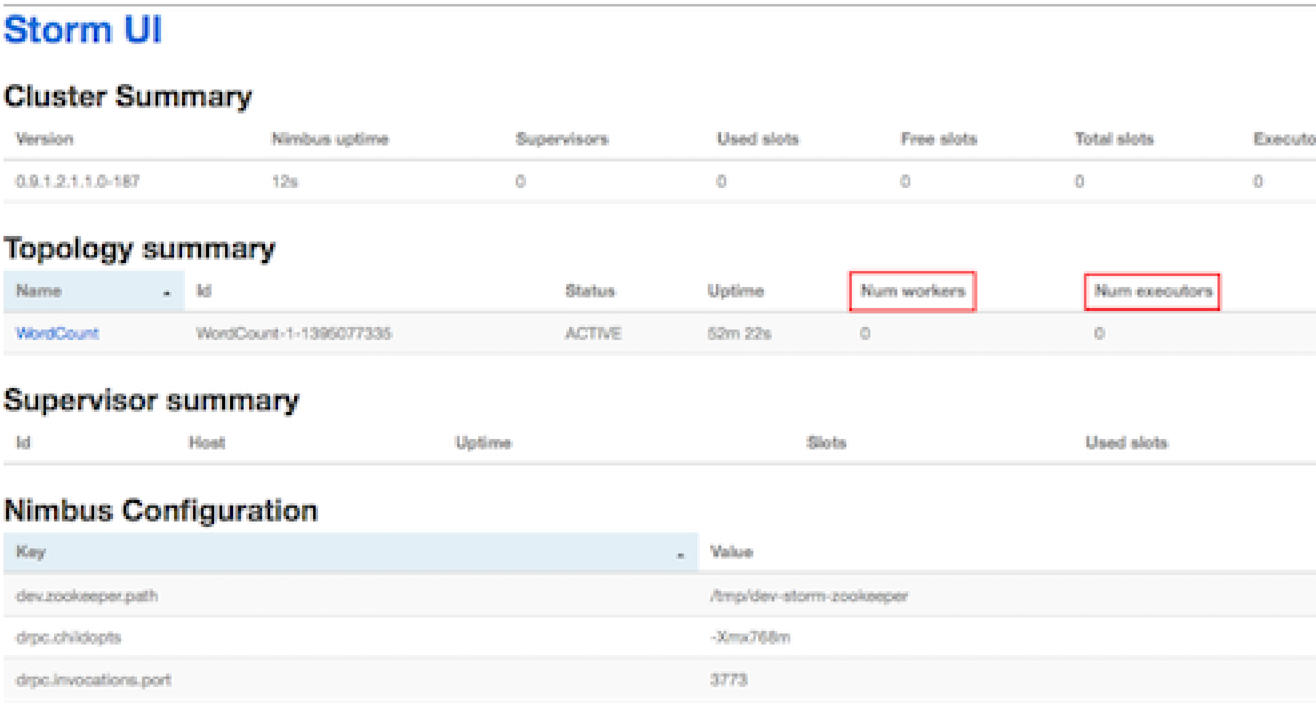
If your cluster is not managed by Ambari, refer to the UI/Logviewer section of Apache Storm security documentation for additional configuration guidelines.

7.2. Using the Storm UI

To access the Storm UI, point a browser to the following URL:

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

In the following image, 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 UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executo
0.9.1.2.1.1.0-187	12s	0	0	0	0	0

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors
WordCount	WordCount-1-1395077335	ACTIVE	52m 22s	0	0

Supervisor summary

Id	Host	Uptime	Slots	Used slots

Nimbus Configuration

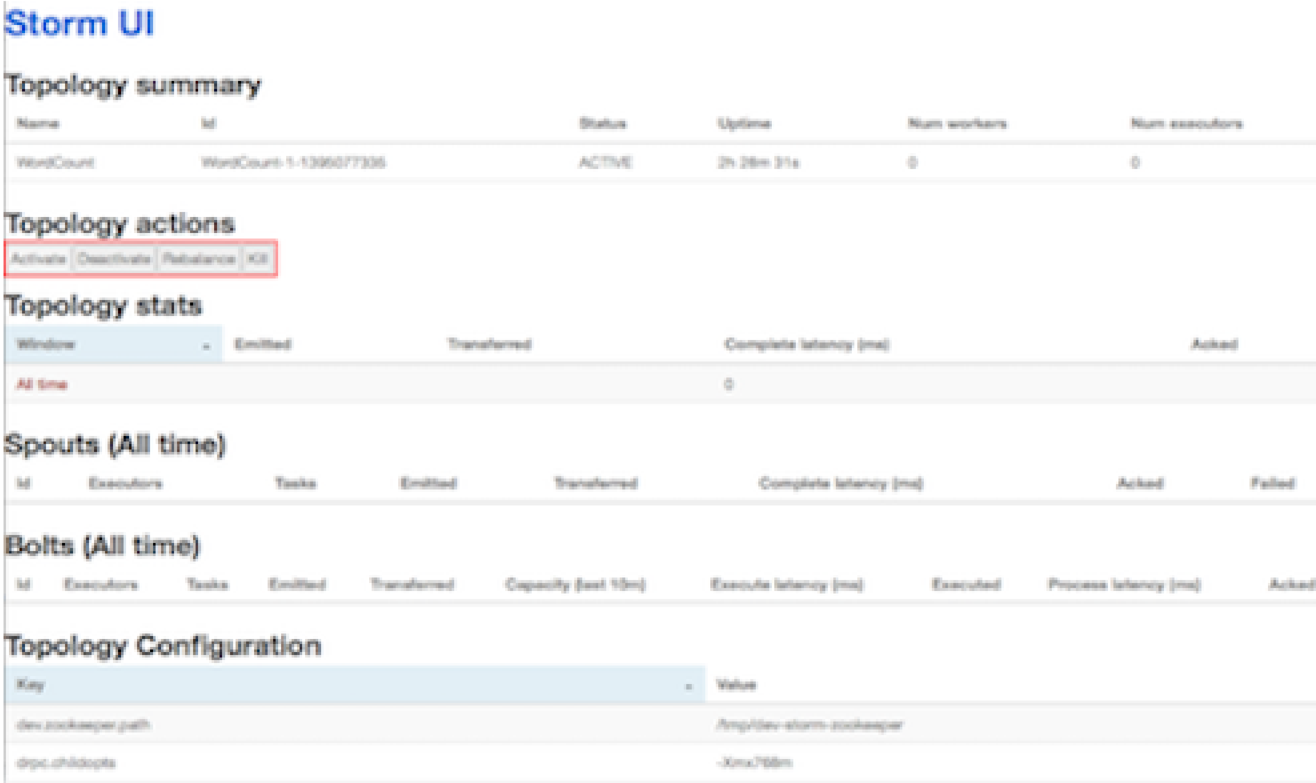
Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx768m
drpc.invocations.port	3773

Storm administrators use the Storm UI to perform the following administrative actions:

Table 7.1. Topology Administrative Actions

Topology Administrative Action	Description
Activate	Return a topology to active status after it has been deactivated.
Deactivate	Set the status of a topology to inactive. Topology uptime is not affected by deactivation.
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	Stop the topology and remove 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. To perform any of the topology actions in the preceding table, you can click the corresponding button (shown in the following image):



Storm UI

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors
WordCount	WordCount-1-1396077306	ACTIVE	2h 28m 31s	0	0

Topology actions

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked
All time			0	

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked

Topology Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx768m

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

8. Monitoring and Debugging an Apache Storm Topology

Debugging Storm applications can be challenging due to the number of moving parts across a large number of nodes in a cluster. Tracing failures to a particular component or a node in the system requires collection and analysis of log files and analysis of debug/trace processes running in the cluster. The following subsections describe features designed to facilitate the process of debugging a storm topology.

8.1. Enabling Dynamic Log Levels

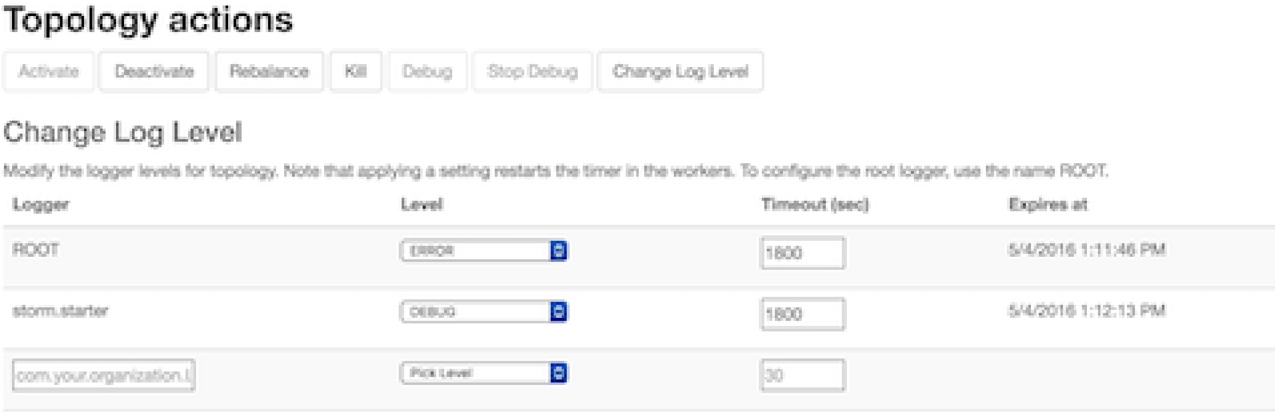
Storm allows users and administrators to dynamically change the log level settings of a running topology. You can change log level settings from either the Storm UI or the command line. No Storm processes need to be restarted for the settings to take effect. The resulting log files are searchable from the Storm UI and logviewer service.

Standard log4j levels include DEBUG, INFO, WARN, ERROR, and FATAL, specifying logging of coarse or finer-grained levels of informational messages and error events. Inheritance is consistent with log4j behavior. For example, if you set the log level of a parent logger, the child loggers start using that level (unless the children have a more restrictive level defined for them).

8.1.1. Setting and Clearing Log Levels Using the Storm UI

To set log level from the Storm UI:

1. Click on a running topology.
2. Click on “Change Log Level” in the Topology Actions section:



Topology actions

Change Log Level

Modify the logger levels for topology. Note that applying a setting restarts the timer in the workers. To configure the root logger, use the name ROOT.

Logger	Level	Timeout (sec)	Expires at
ROOT	ERROR	1800	5/4/2016 1:11:46 PM
storm.starter	DEBUG	1800	5/4/2016 1:12:13 PM
com.your.organization.L	Pick Level	30	

3. For an existing logger, select the desired log level for the logger. Alternately, add a logger and set the desired log level.
4. Optionally, specify a timeout value in seconds, after which changes will be reverted automatically. Specify 0 if no timeout is needed.

5. Click "Apply".

The preceding example sets the log4j log level to ERROR for the root logger, and to DEBUG for `storm.starter`. Logging for the root logger will be limited to error events, and finer-grained informational events (useful for debugging the application) will be recorded for `storm.starter` packages.

To clear (reset) a log level setting using the Storm UI, click on the “Clear” button. This reverts the log level back to what it was before you added the setting. The log level line will disappear from the UI.

8.1.2. Setting and Clearing Log Levels Using the CLI

To set log level from the command line, use the following command:

```
./bin/storm set_log_level [topology name] -l [logger name]=[LEVEL]:[TIMEOUT]
```

The following example sets the ROOT logger to DEBUG for 30 seconds:

```
./bin/storm set_log_level my_topology -l ROOT=DEBUG:30
```

To clear (reset) the log level using the CLI, use the following command. This reverts the log level back to what it was before you added the setting.

```
./bin/storm set_log_level [topology name] -r [logger name]
```

The following example clears the ROOT logger dynamic log level, resetting it to its original value:

```
./bin/storm set_log_level my_topology -r ROOT
```

For more information, see Apache STORM-412.

8.2. Enabling Topology Event Logging

The topology event inspector lets you view tuples as they flow through different stages of a Storm topology. This tool is useful for inspecting tuples emitted from a spout or a bolt in the topology pipeline while the topology is running; you do not need to stop or redeploy the topology to use the event inspector. The normal flow of tuples from spouts to bolts is not affected by turning on event logging.

8.2.1. Configuring Topology Event Logging

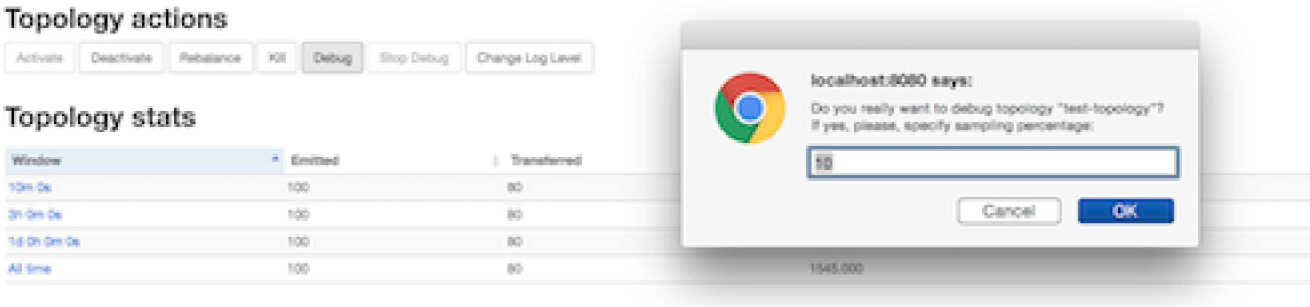
Event logging sends events (tuples) from each component to an internal eventlogger bolt.

Event logging is disabled by default, due to a slight performance degradation associated with eventlogger tasks.

To enable event logging, set the `topology.eventlogger.executors` property to a non-zero value when submitting your topology. You can set the property globally in the `storm.yaml` file, or use the command line. For more information about `topology.eventlogger.executors` and other property settings, see Configuring Apache Storm for Production Environments.

8.2.2. Enabling Event Logging

To log events for an entire topology, click the "Debug" button under "Topology actions" in the topology view. This setting logs tuples from all spouts and bolts in a topology at the specified sampling percentage.



Topology stats

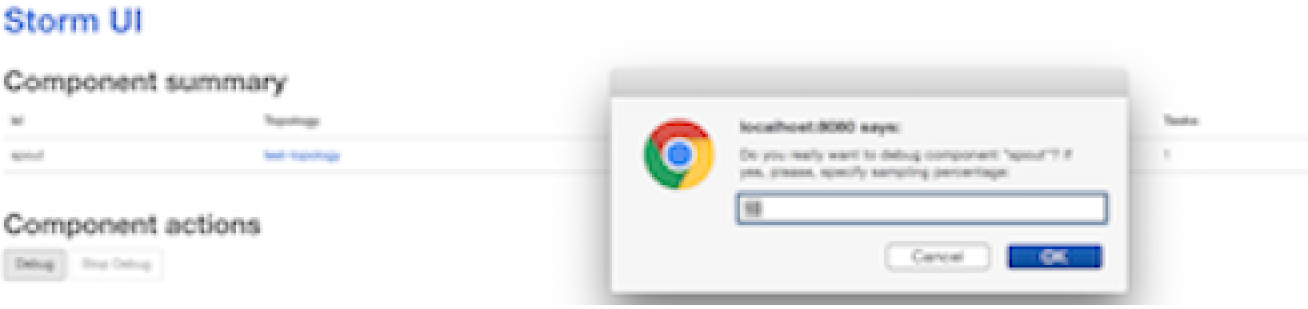
Window	Emitted	Transferred	
10m 0s	100	80	
3h 0m 0s	100	80	
1d 0h 0m 0s	100	80	
All time	100	80	1545.000

localhost:8080 says:

Do you really want to debug topology "test-topology"? If yes, please, specify sampling percentage:

10

To log events at a specific spout or bolt level, navigate to the corresponding component page and click "Debug" under component actions:



Component summary

Id	Topology	Tasks
spout	test-topology	1

localhost:8080 says:

Do you really want to debug component "spout"? If yes, please, specify sampling percentage:

10

8.2.3. Viewing Event Logs

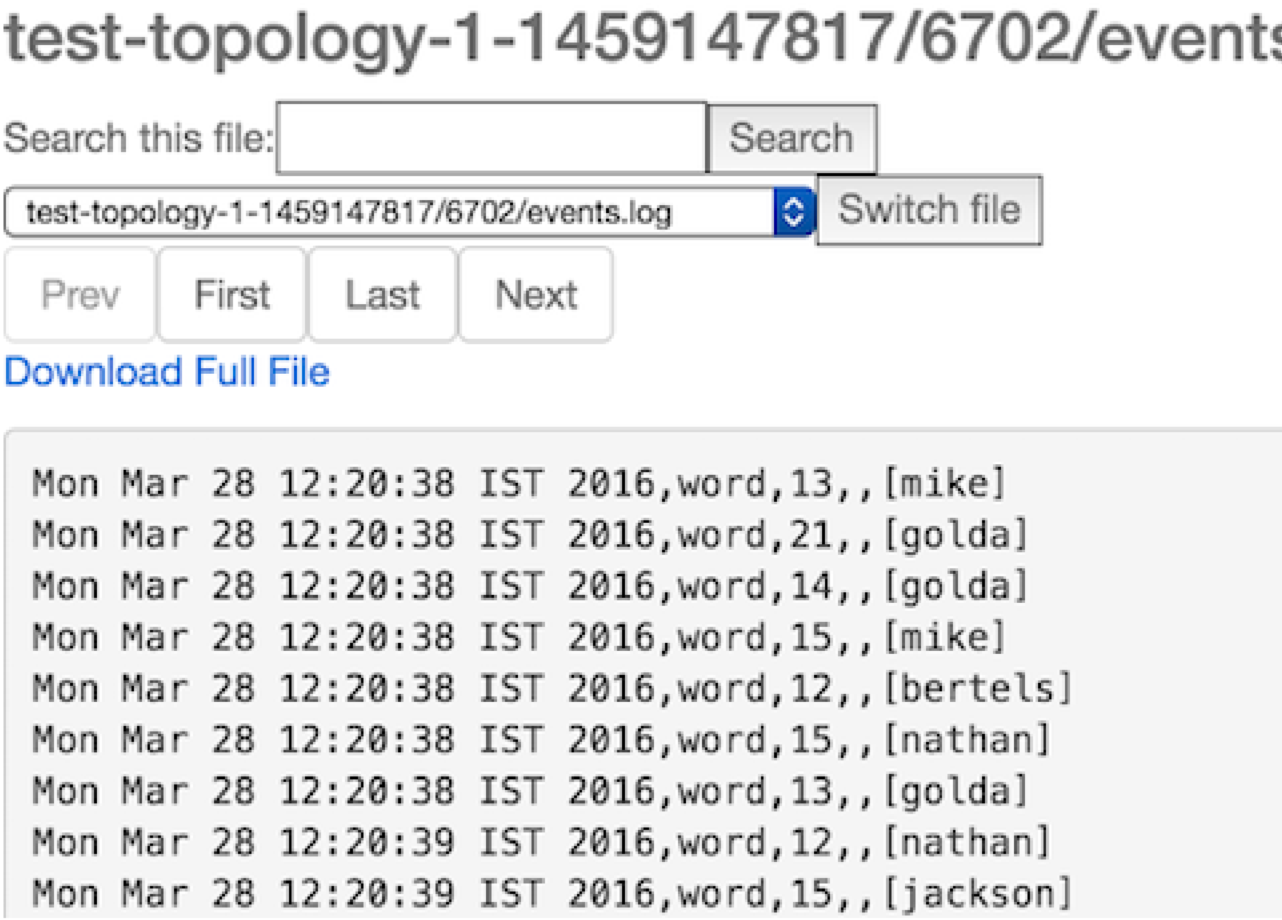
Prerequisite: The Storm "logviewer" process should be running so that you can view the logged tuples. If it is not already running, start the log viewer by running the following command from the storm installation directory:

```
bin/storm logviewer
```

To view tuples:

1. From the Storm UI, navigate to the specific spout or bolt component page.

2. Click on the "events" link in the Debug column of the component summary. This will open a view similar to the following:


test-topology-1-1459147817/6702/events
Search this file:
Search
test-topology-1-1459147817/6702/events.log
Switch file
Prev
First
Last
Next
Download Full File

```
Mon Mar 28 12:20:38 IST 2016,word,13,,[mike]
Mon Mar 28 12:20:38 IST 2016,word,21,,[golda]
Mon Mar 28 12:20:38 IST 2016,word,14,,[golda]
Mon Mar 28 12:20:38 IST 2016,word,15,,[mike]
Mon Mar 28 12:20:38 IST 2016,word,12,,[bertels]
Mon Mar 28 12:20:38 IST 2016,word,15,,[nathan]
Mon Mar 28 12:20:38 IST 2016,word,13,,[golda]
Mon Mar 28 12:20:39 IST 2016,word,12,,[nathan]
Mon Mar 28 12:20:39 IST 2016,word,15,,[jackson]
```

Each line in the event log contains an entry corresponding to a tuple emitted from a specific spout or bolt, presented in a comma-separated format:

```
Timestamp, Component name, Component task-id, MessageId (incase of
 anchoring), List of emitted values
```

3. Navigate between different pages to view logged events.

8.2.4. Accessing Event Logs on a Secure Cluster

If you want to view logs in secure mode, ensure that your browser is configured to use authentication with all supervisor nodes running logviewer. This process is similar to the process used to configure access to the Storm UI on a secure cluster (described in Configuring the Storm UI).

Add domains to the white list by setting `network.negotiate-auth.trusted-uris` to a comma-separated list containing one or more domain names and URLs. For example, the following steps configure Firefox to use authentication with two nodes:

1. Go to the `about:config` configuration page.
2. Search for the `network.negotiate-auth.trusted-uris` configuration option.
3. Double-click on the option.

4. An "Enter string value" dialog box opens.

5. In this box, enter the values `http://node1.example.com, http://node2.example.com`.

6. Click OK to finish.

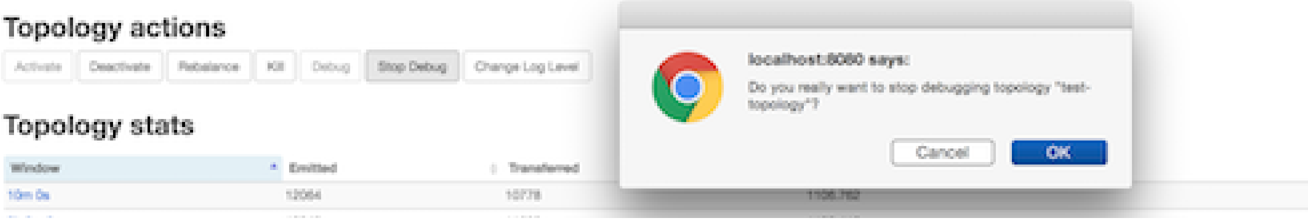
7. Close and relaunch the browser.

If your cluster is not managed by Ambari, refer to the UI/Logviewer section of Apache Storm security documentation for additional configuration guidelines.

8.2.5. Disabling Event Logs

To disable event logging for a component or for a topology, click "Stop Debug" under the Component actions or Topology actions page (respectively) in the Storm UI.

The following example disables topology "test-topology":



Topology actions

Topology stats

localhost:8080 says:

Do you really want to stop debugging topology "test-topology"?

Window	Emitted	Transferred	
10m 0s	12064	10778	1106.762

8.2.6. Extending Event Logging

The Storm eventlogger bolt uses the IEventLogger interface to log events. The default implementation is a`FileBasedEventLogger`, which logs events to a log file at `logs/workers-artifacts/<topology-id>/<worker-port>/events.log`.

To extend event logging functionality (for example, to build a search index or log events to a database), add an alternate implementation of the `IEventLogger` interface.

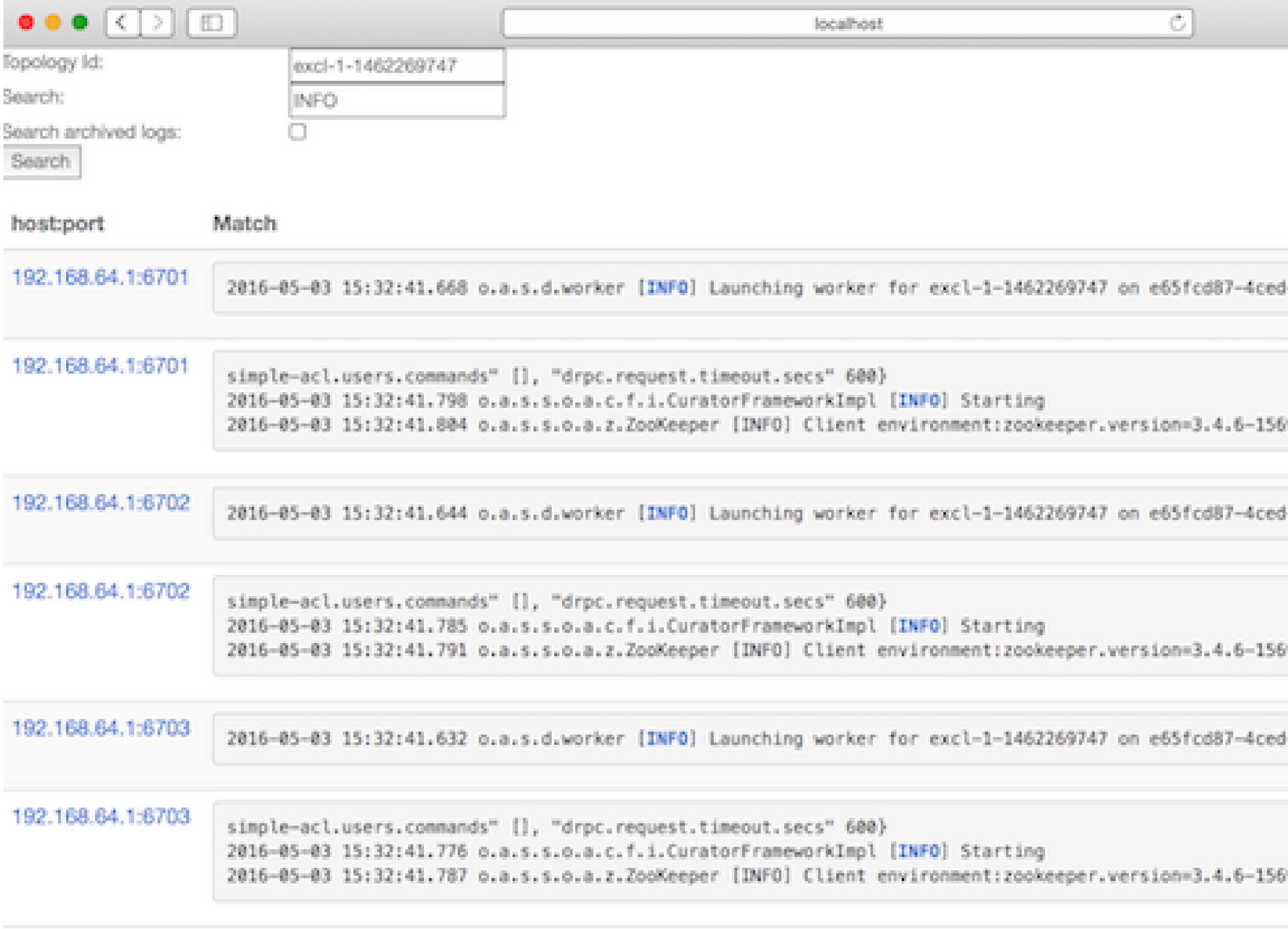
```
/**
      * EventLogger interface for logging the event info to a sink like log
 file or db
      * for inspecting the events via UI for debugging.
      */public interface IEventLogger {
          void prepare(Map stormConf, TopologyContext context);
          /**
          * Invoked when the {@link EventLoggerBolt} receives a tuple from the
 spouts or bolts that
          * have event logging enabled.
          *
          * @param e the event
          */
          void log(EventInfo e);
          /**
          * Invoked when the event logger bolt is cleaned up
          */
          void close();
          }
```

See JIRA STORM-954 for more details.

8.3. Enabling Distributed Log Search

The distributed log search capability allows users to search across log files (including archived logs) to find information and events for a specific topology. Results include matches from all supervisor nodes.

This feature is useful when searching for patterns across workers or supervisors of a topology. (A similar log search is supported within specific worker log files via the Storm UI.)



Topology Id: excl-1-1462269747

Search: INFO

Search archived logs: ☐

host:port	Match
192.168.64.1:6701	2016-05-03 15:32:41.668 o.a.s.d.worker [INFO] Launching worker for excl-1-1462269747 on e65fcd87-4ced
192.168.64.1:6701	simple-acl.users.commands" [], "drpc.request.timeout.secs" 600} 2016-05-03 15:32:41.798 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl [INFO] Starting 2016-05-03 15:32:41.804 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:zookeeper.version=3.4.6-156
192.168.64.1:6702	2016-05-03 15:32:41.644 o.a.s.d.worker [INFO] Launching worker for excl-1-1462269747 on e65fcd87-4ced
192.168.64.1:6702	simple-acl.users.commands" [], "drpc.request.timeout.secs" 600} 2016-05-03 15:32:41.785 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl [INFO] Starting 2016-05-03 15:32:41.791 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:zookeeper.version=3.4.6-156
192.168.64.1:6703	2016-05-03 15:32:41.632 o.a.s.d.worker [INFO] Launching worker for excl-1-1462269747 on e65fcd87-4ced
192.168.64.1:6703	simple-acl.users.commands" [], "drpc.request.timeout.secs" 600} 2016-05-03 15:32:41.776 o.a.s.s.o.a.c.f.i.CuratorFrameworkImpl [INFO] Starting 2016-05-03 15:32:41.787 o.a.s.s.o.a.z.ZooKeeper [INFO] Client environment:zookeeper.version=3.4.6-156

For more information about log search capabilities, see Apache JIRA STORM-902.

8.4. Dynamic Worker Profiling

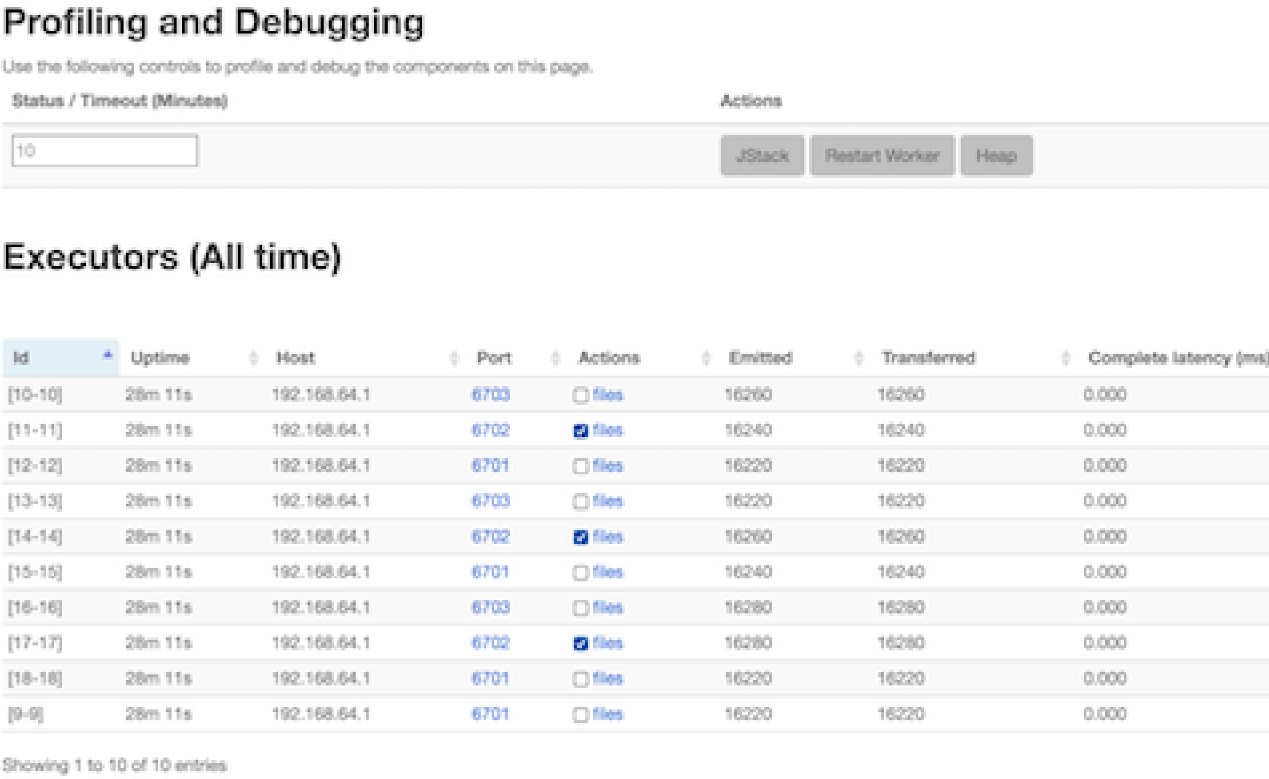
You can request the following types of worker profile data directly from the Storm UI , without restarting the topologies:

- Heap dumps
- JStack output

- JProfile recordings

To access this feature:

1. Navigate to the “Profiling and Debugging” section on the Spout/Bolt component summary page. There you will see buttons to request JStack output or generate a Heap dump:



Profiling and Debugging

Use the following controls to profile and debug the components on this page.

Status / Timeout (Minutes)	Actions
10	JStack Restart Worker Heap

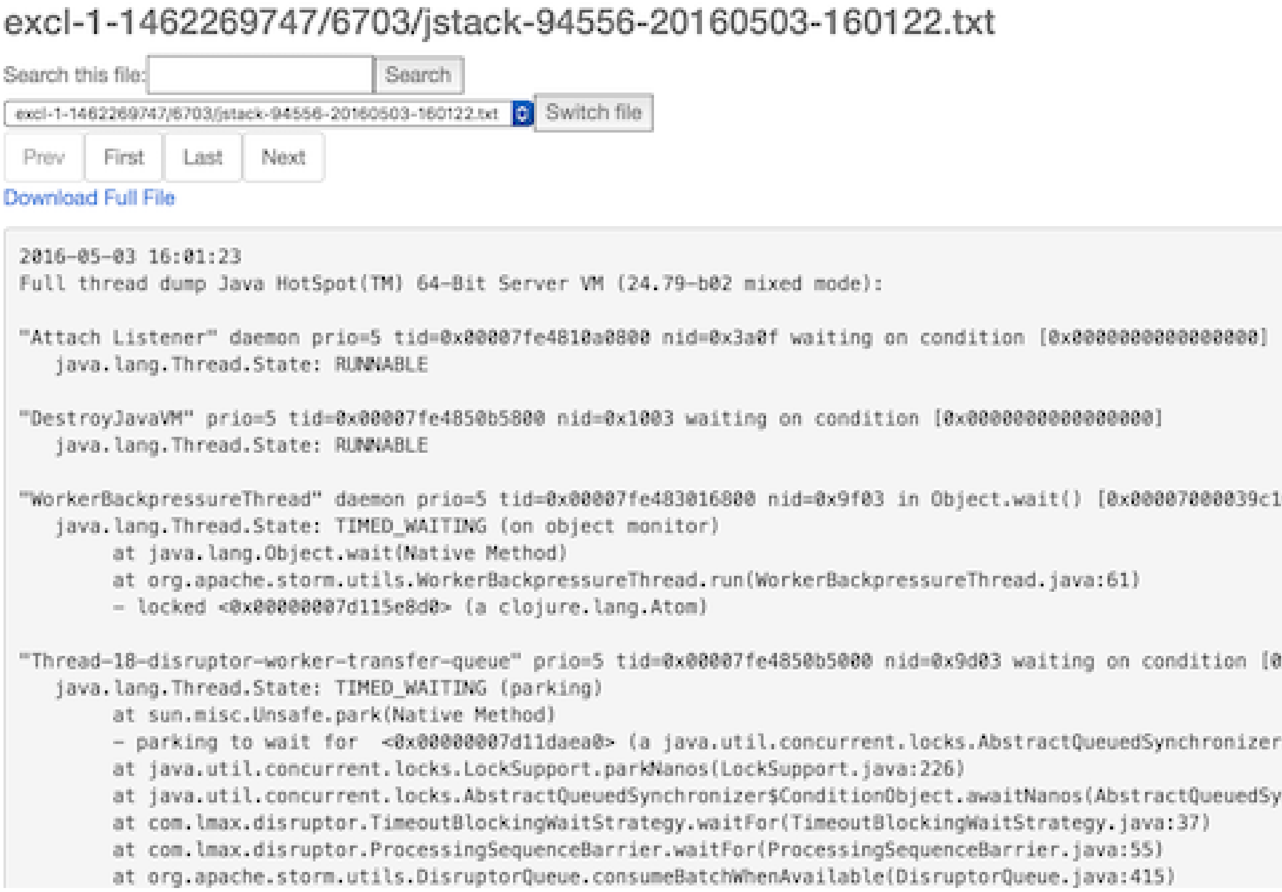
Executors (All time)

Id	Uptime	Host	Port	Actions	Emitted	Transferred	Complete latency (ms)
[10-10]	28m 11s	192.168.64.1	6703	□ files	16260	16260	0.000
[11-11]	28m 11s	192.168.64.1	6702	☑ files	16240	16240	0.000
[12-12]	28m 11s	192.168.64.1	6701	□ files	16220	16220	0.000
[13-13]	28m 11s	192.168.64.1	6703	□ files	16220	16220	0.000
[14-14]	28m 11s	192.168.64.1	6702	☑ files	16260	16260	0.000
[15-15]	28m 11s	192.168.64.1	6701	□ files	16240	16240	0.000
[16-16]	28m 11s	192.168.64.1	6703	□ files	16280	16280	0.000
[17-17]	28m 11s	192.168.64.1	6702	☑ files	16280	16280	0.000
[18-18]	28m 11s	192.168.64.1	6701	□ files	16220	16220	0.000
[9-9]	28m 11s	192.168.64.1	6701	□ files	16220	16220	0.000

Showing 1 to 10 of 10 entries

Note that you can also restart worker processes from this page.

2. To view output, click the “files” link under “Actions”.



excl-1-1462269747/6703/jstack-94556-20160503-160122.txt

```
2016-05-03 16:01:23
Full thread dump Java HotSpot(TM) 64-Bit Server VM (24.79-b02 mixed mode):

"Attach Listener" daemon prio=5 tid=0x00007fe4810a0800 nid=0x3a0f waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"DestroyJavaVM" prio=5 tid=0x00007fe4850b5800 nid=0x1003 waiting on condition [0x0000000000000000]
   java.lang.Thread.State: RUNNABLE

"WorkerBackpressureThread" daemon prio=5 tid=0x00007fe483016800 nid=0x9f03 in Object.wait() [0x00007000039c1
   java.lang.Thread.State: TIMED_WAITING (on object monitor)
        at java.lang.Object.wait(Native Method)
        at org.apache.storm.utils.WorkerBackpressureThread.run(WorkerBackpressureThread.java:61)
        - locked <0x00000007d115e8d0> (a clojure.lang.Atom)

"Thread-18-disruptor-worker-transfer-queue" prio=5 tid=0x00007fe4850b5000 nid=0x9d03 waiting on condition [0
   java.lang.Thread.State: TIMED_WAITING (parking)
        at sun.misc.Unsafe.park(Native Method)
        - parking to wait for  <0x00000007d11daea0> (a java.util.concurrent.locks.AbstractQueuedSynchronizer
        at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
        at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.awaitNanos(AbstractQueuedSy
        at com.lmax.disruptor.TimeoutBlockingWaitStrategy.waitFor(TimeoutBlockingWaitStrategy.java:37)
        at com.lmax.disruptor.ProcessingSequenceBarrier.waitFor(ProcessingSequenceBarrier.java:55)
        at org.apache.storm.utils.DisruptorQueue.consumeBatchWhenAvailable(DisruptorQueue.java:415)
```

3. To download output for offline analysis, click the associated link under the "Actions" column on the Profiling and Debugging page, or click "Download Full File" on the output page.

See Apache JIRA STORM-1157 for more information.

9. Tuning an Apache Storm Topology

Because Storm topologies operate on streaming data (rather than data at rest, as in HDFS) they are sensitive to data sources. When tuning 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?

In a Storm cluster, most of the computational burden typically falls on the Supervisor and Worker nodes. 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.

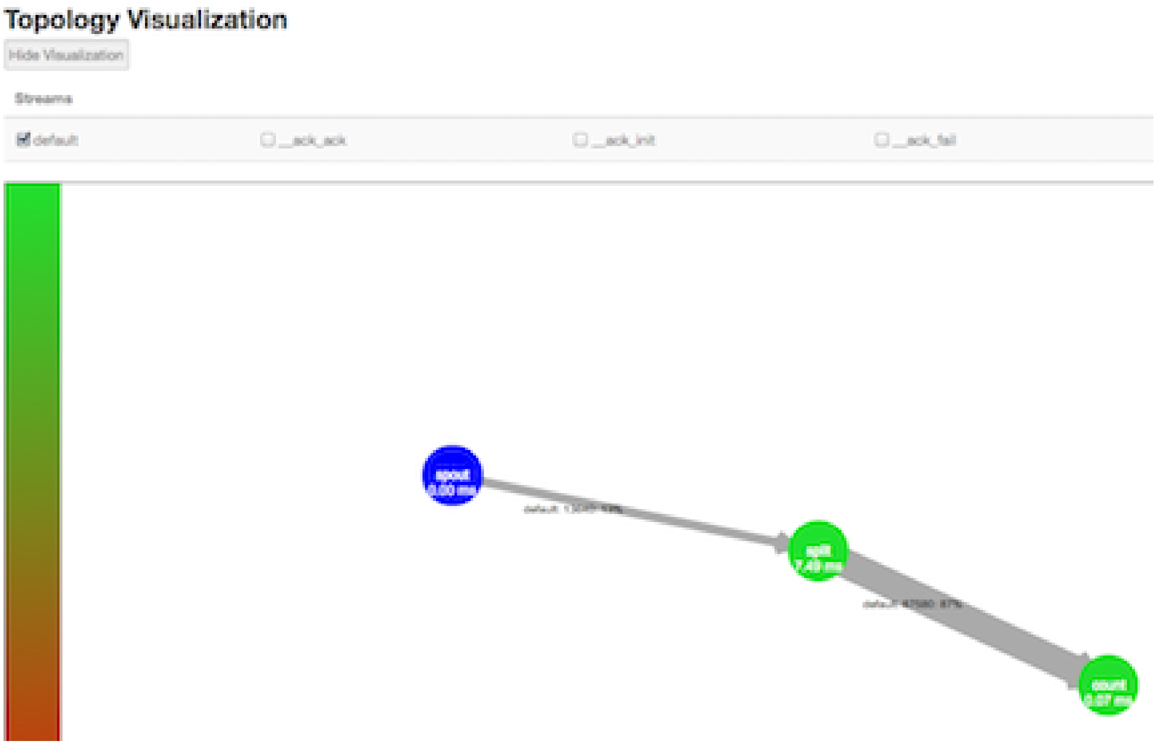
The performance of a Storm topology degrades when it cannot ingest data fast enough to keep up with the data source. The velocity of incoming streaming data changes over time. When the data flow of the source exceeds what the topology can process, memory buffers fill up. The topology suffers frequent timeouts and must replay tuples to process them.

Use the following techniques to identify and overcome poor topology performance due to mismatched data flow rates between source and application:

1. Identify the bottleneck.

   a. In the Storm UI, click Show Visualization to display a visual representation of your topology 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.



Topology Visualization

Hide Visualization

Streams

☑ default ☐ __ack_ack ☐ __ack_init ☐ __ack_fail

Note

In addition to bolts defined 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. The names of these "acker" bolts are prefixed with an underscore in the visualization, but they do not appear in the default view.

To display component-specific data about successful acknowledgements, select the `_ack_ack` checkbox. To display component-specific data about failed acknowledgements, select the `_ack_fail` checkbox.

b. To verify that you have found the topology bottleneck, rewrite the `execute()` method of the target bolt or spout so that it performs no operations. If the performance of the topology improves, you have found the bottleneck.

   Alternately, turn off each topology component, one at a time, to find the component responsible for the bottleneck.

2. Refer to "Performance Guidelines for Developing a Storm Topology" for several performance-related development guidelines.

3. Adjust topology configuration settings. For more information, see Configuring Storm Resource Usage.

4. Increase the parallelism for the target spout or bolt. Parallelism units are a useful conceptual tool for determining how to distribute processing tasks across a distributed application.

Hortonworks recommends using the following calculation to determine the total number of parallelism units for a topology.

```
(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.