**Why use Apache Storm?**

Apache Storm is a free and open source distributed real time computation system. Apache Storm makes it easy to reliably process unbounded streams of data, doing for real time processing what Hadoop did for batch processing. Apache Storm is simple, can be used with any programming language, and is a lot of fun to use!

Apache Storm has many use cases: real time analytics, online machine learning, continuous computation, distributed RPC, ETL, and more. Apache Storm is fast: a benchmark clocked it at over **a million tuples processed per second per node**. It is scalable, fault-tolerant, guarantees your data will be processed, and is easy to set up and operate.

Apache Storm integrates with the queueing and database technologies you already use. An Apache Storm topology consumes streams of data and processes those streams in arbitrarily complex ways, repartitioning the streams between each stage of the computation however needed. Read more in the tutorial.

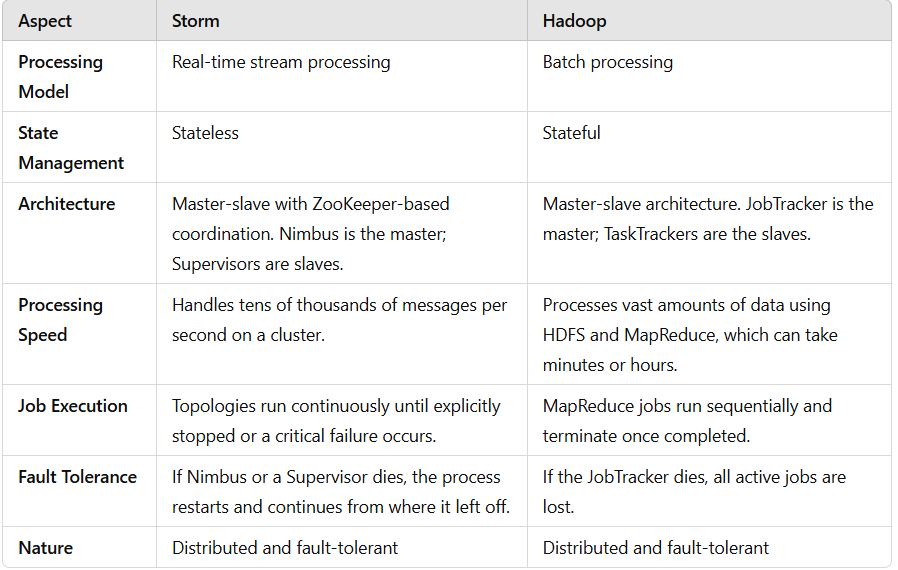
**What is Apache Storm?**

Apache Storm is a distributed real-time big data-processing system. Storm is designed to process vast amount of data in a fault-tolerant and horizontal scalable method. It is a streaming data framework that has the capability of highest ingestion rates. Though Storm is stateless, it manages distributed environment and cluster state via Apache ZooKeeper. It is simple and you can execute all kinds of manipulations on real-time data in parallel.

Apache Storm is continuing to be a leader in real-time data analytics. Storm is easy to setup, operate and it guarantees that every message will be processed through the topology at least once.

**Apache Storm vs Hadoop**

Basically, Hadoop and Storm frameworks are used for analysing big data. Both of them complement each other and differ in some aspects. Apache Storm does all the operations except persistency, while Hadoop is good at everything but lags in real-time computation. The following table compares the attributes of Storm and Hadoop.



**Use Cases of Apache Storm**

1. **Twitter**
   * Used for **Publisher Analytics Products** to process every tweet and click in real time.
   * Integrated deeply with Twitter's infrastructure.
2. **NaviSite**
   * Implements an **event log monitoring/auditing system**.
   * Logs are processed through Storm, matching messages against configured regular expressions. Matches are saved to a database.
3. **Wego**
   * A **travel metasearch engine** that processes real-time travel data from global sources.
   * Resolves concurrency issues and ensures the best data matches for end-users.

**Benefits of Apache Storm**

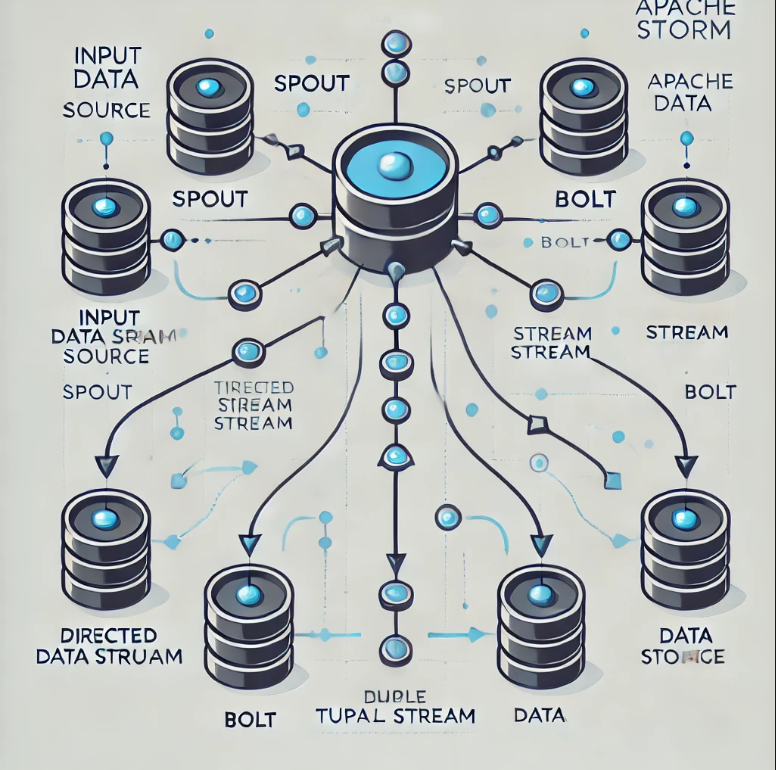
1. **Open Source and Versatile**
   * Robust, user-friendly, and suitable for both small and large organizations.
2. **Fault Tolerant and Reliable**
   * Maintains stability and guarantees data processing even with node failures.
3. **Real-Time Stream Processing**
   * Designed for low-latency data processing, completing tasks in seconds or minutes.
4. **High Speed and Scalability**
   * Processes enormous amounts of data quickly and scales linearly with increased resources.
5. **Flexibility**
   * Supports **multiple programming languages**, making it adaptable to various development environments.
6. **Low Latency**
   * Ensures rapid data refresh and delivery response.
7. **Operational Intelligence**
   * Provides insights into real-time data processing and decision-making capabilities.

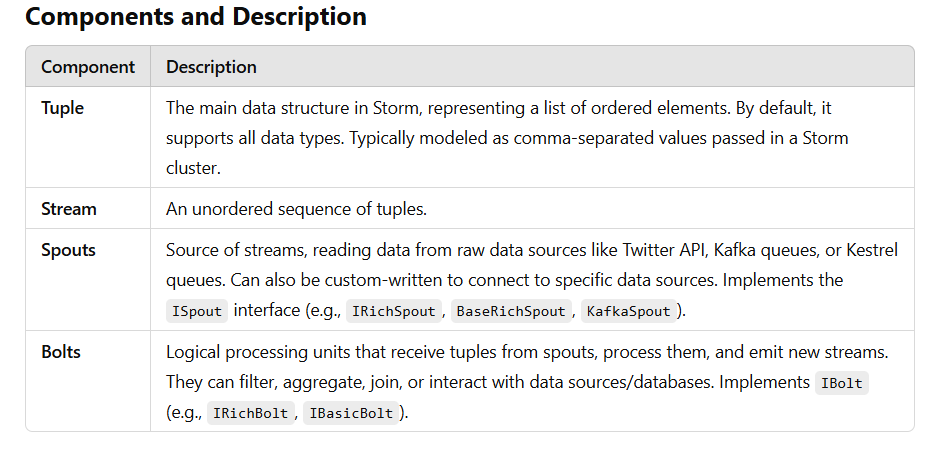
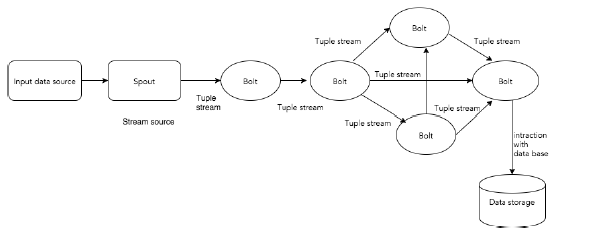
|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  | |
|  |  |

# Apache Storm - Core Concepts

Apache Storm reads raw streams of real-time data from one end, processes it through a sequence of small units, and outputs the useful information at the other end.

The core components of Apache Storm are explained below:





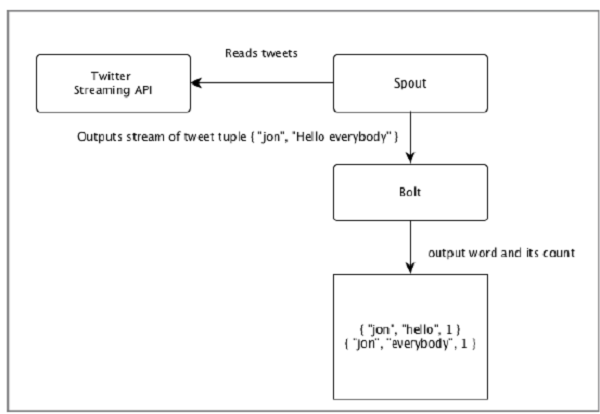
**Example: Twitter Analysis**

**Input**: Tweets are fetched from the Twitter Streaming API by a spout, which emits them as tuples, e.g., ["username", "tweet"].

**Processing**:

* Bolt 1 splits the tweet into words.
* Bolt 2 calculates the word count.
* Data is persisted to a configured database.

**Output**: Results can be queried from the database.



**Topology**

Spouts and bolts are connected to form a **topology**, which represents real-time application logic.

**Topology**:

A **directed graph**, where:

* **Vertices**: Computation units (spouts and bolts).
* **Edges**: Stream of data.

Starts with spouts that emit data to bolts.

Storm keeps the topology running continuously until manually terminated.

**Tasks**

* Execution of spouts and bolts is referred to as a **task**.
* Tasks run in parallel, with multiple instances of spouts and bolts executing in separate threads.

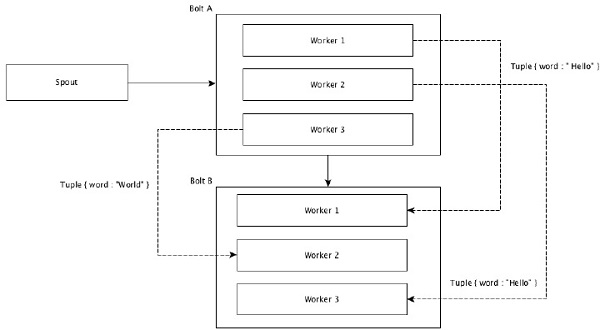
**Workers**

* Topologies run on multiple worker nodes in a distributed manner.
* Worker nodes listen for jobs and manage their execution.

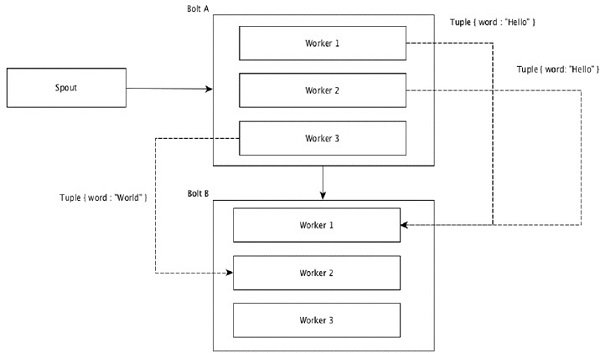
## **Stream Grouping**

* Stream of data flows from spouts to bolts or from one bolt to another bolt. Stream grouping controls how the tuples are routed in the topology and helps us to understand the tuples flow in the topology. There are four in-built groupings as explained below.

### **Shuffle Grouping**

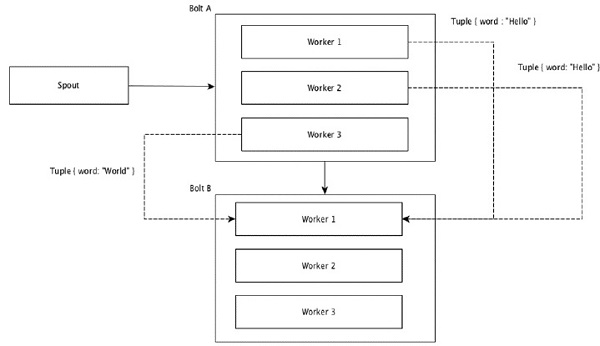
* In shuffle grouping, an equal number of tuples is distributed randomly across all of the workers executing the bolts. The following diagram depicts the structure.
* 

### **Field Grouping**

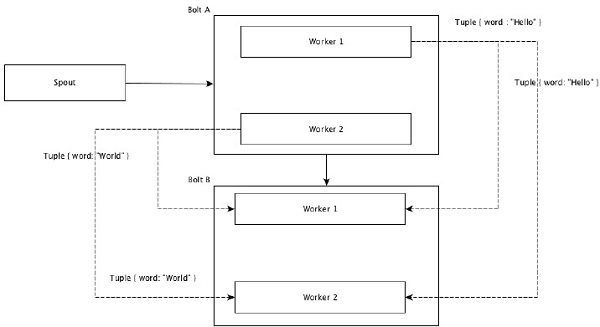
* The fields with same values in tuples are grouped together and the remaining tuples kept outside. Then, the tuples with the same field values are sent forward to the same worker executing the bolts. For example, if the stream is grouped by the field “word”, then the tuples with the same string, “Hello” will move to the same worker. The following diagram shows how Field Grouping works.
* 

### **Global Grouping**

* All the streams can be grouped and forward to one bolt. This grouping sends tuples generated by all instances of the source to a single target instance (specifically, pick the worker with lowest ID).



### **All Grouping**

* All Grouping sends a single copy of each tuple to all instances of the receiving bolt. This kind of grouping is used to send signals to bolts. All grouping is useful for join operations.
* 

**Apache Storm - Cluster Architecture**

One of the main highlights of the Apache Storm is that it is a fault-tolerant, fast with no “Single Point of Failure” (SPOF) distributed application. We can install Apache Storm in as many systems as needed to increase the capacity of the application.

Let’s have a look at how the Apache Storm cluster is designed and its internal architecture. The following diagram depicts the cluster design.

**Apache Storm - Workflow**

A working Storm cluster should have one nimbus and one or more supervisors. Another important node is Apache ZooKeeper, which will be used for the coordination between the nimbus and the supervisors.

Let us now take a close look at the workflow of Apache Storm −

* Initially, the nimbus will wait for the “Storm Topology” to be submitted to it.
* Once a topology is submitted, it will process the topology and gather all the tasks that are to be carried out and the order in which the task is to be executed.
* Then, the nimbus will evenly distribute the tasks to all the available supervisors.
* At a particular time interval, all supervisors will send heartbeats to the nimbus to inform that they are still alive.
* When a supervisor dies and doesn’t send a heartbeat to the nimbus, then the nimbus assigns the tasks to another supervisor.
* When the nimbus itself dies, supervisors will work on the already assigned task without any issue.
* Once all the tasks are completed, the supervisor will wait for a new task to come in.
* In the meantime, the dead nimbus will be restarted automatically by service monitoring tools.
* The restarted nimbus will continue from where it stopped. Similarly, the dead supervisor can also be restarted automatically. Since both the nimbus and the supervisor can be restarted automatically and both will continue as before, Storm is guaranteed to process all the task at least once.
* Once all the topologies are processed, the nimbus waits for a new topology to arrive and similarly the supervisor waits for new tasks.

By default, there are two modes in a Storm cluster −

* **Local mode** − This mode is used for development, testing, and debugging because it is the easiest way to see all the topology components working together. In this mode, we can adjust parameters that enable us to see how our topology runs in different Storm configuration environments. In Local mode, storm topologies run on the local machine in a single JVM.
* **Production mode** − In this mode, we submit our topology to the working storm cluster, which is composed of many processes, usually running on different machines. As discussed in the workflow of storm, a working cluster will run indefinitely until it is shut down.

# Storm - Distributed Messaging System

Apache Storm processes real-time data and the input normally comes from a message queuing system. An external distributed messaging system will provide the input necessary for the real time computation. Spout will read the data from the messaging system and convert it into tuples and input into the Apache Storm. The interesting fact is that Apache Storm uses its own distributed messaging system internally for the communication between its nimbus and supervisor.

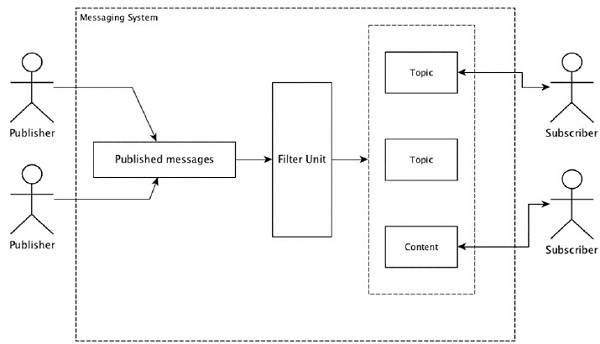
What is Distributed Messaging System?

Distributed messaging is based on the concept of reliable message queuing. Messages are queued asynchronously between client applications and messaging systems. A distributed messaging system provides the benefits of reliability, scalability, and persistence.

Most of the messaging patterns follow the **publish-subscribe** model (simply **Pub-Sub**) where the senders of the messages are called **publishers** and those who want to receive the messages are called **subscribers**.

Once the message has been published by the sender, the subscribers can receive the selected message with the help of a filtering option. Usually we have two types of filtering, one is **topic-based filtering** and another one is **content-based filtering**.

Note that the pub-sub model can communicate only via messages. It is a very loosely coupled architecture; even the senders don’t know who their subscribers are. Many of the message patterns enable with message broker to exchange publish messages for timely access by many subscribers. A real-life example is Dish TV, which publishes different channels like sports, movies, music, etc., and anyone can subscribe to their own set of channels and get them whenever their subscribed channels are available.



The following table describes some of the popular high throughput messaging systems −

|  |  |
| --- | --- |
| **Distributed messaging system** | **Description** |
| Apache Kafka | Kafka was developed at LinkedIn corporation and later it became a sub-project of Apache. Apache Kafka is based on brokerenabled, persistent, distributed publish-subscribe model. Kafka is fast, scalable, and highly efficient. |
| RabbitMQ | RabbitMQ is an open source distributed robust messaging application. It is easy to use and runs on all platforms. |
| JMS(Java Message Service) | JMS is an open source API that supports creating, reading, and sending messages from one application to another. It provides guaranteed message delivery and follows publish-subscribe model. |
| ActiveMQ | ActiveMQ messaging system is an open source API of JMS. |
| ZeroMQ | ZeroMQ is broker-less peer-peer message processing. It provides push-pull, router-dealer message patterns. |
| Kestrel | Kestrel is a fast, reliable, and simple distributed message queue. |

Thrift Protocol

Thrift was built at Facebook for cross-language services development and remote procedure call (RPC). Later, it became an open source Apache project. Apache Thrift is an **Interface Definition Language** and allows to define new data types and services implementation on top of the defined data types in an easy manner.

Apache Thrift is also a communication framework that supports embedded systems, mobile applications, web applications, and many other programming languages. Some of the key features associated with Apache Thrift are its modularity, flexibility, and high performance. In addition, it can perform streaming, messaging, and RPC in distributed applications.

Storm extensively uses Thrift Protocol for its internal communication and data definition. Storm topology is simply **Thrift Structs**. Storm Nimbus that runs the topology in Apache Storm is a **Thrift service**.

**Apache Storm - Working Example**

We have gone through the core technical details of the Apache Storm and now it is time to code some simple scenarios.

Scenario – Mobile Call Log Analyzer

Mobile call and its duration will be given as input to Apache Storm and the Storm will process and group the call between the same caller and receiver and their total number of calls.

Spout Creation

Spout is a component which is used for data generation. Basically, a spout will implement an IRichSpout interface. “IRichSpout” interface has the following important methods −

* **open** − Provides the spout with an environment to execute. The executors will run this method to initialize the spout.
* **nextTuple** − Emits the generated data through the collector.
* **close** − This method is called when a spout is going to shutdown.
* **declareOutputFields** − Declares the output schema of the tuple.
* **ack** − Acknowledges that a specific tuple is processed
* **fail** − Specifies that a specific tuple is not processed and not to be reprocessed.

Open

The signature of the **open** method is as follows −

open(Map conf, TopologyContext context, SpoutOutputCollector collector)

* **conf** − Provides storm configuration for this spout.
* **context** − Provides complete information about the spout place within the topology, its task id, input and output information.
* **collector** − Enables us to emit the tuple that will be processed by the bolts.

nextTuple

The signature of the **nextTuple** method is as follows −

nextTuple()

nextTuple() is called periodically from the same loop as the ack() and fail() methods. It must release control of the thread when there is no work to do, so that the other methods have a chance to be called. So the first line of nextTuple checks to see if processing has finished. If so, it should sleep for at least one millisecond to reduce load on the processor before returning.

close

The signature of the **close** method is as follows −

close()

declareOutputFields

The signature of the **declareOutputFields** method is as follows −

declareOutputFields(OutputFieldsDeclarer declarer)

**declarer** − It is used to declare output stream ids, output fields, etc.

This method is used to specify the output schema of the tuple.

ack

The signature of the **ack** method is as follows −

ack(Object msgId)

This method acknowledges that a specific tuple has been processed.

fail

The signature of the **nextTuple** method is as follows −

ack(Object msgId)

This method informs that a specific tuple has not been fully processed. Storm will reprocess the specific tuple.

FakeCallLogReaderSpout

In our scenario, we need to collect the call log details. The information of the call log contains.

* caller number
* receiver number
* duration

Since, we don’t have real-time information of call logs, we will generate fake call logs. The fake information will be created using Random class. The complete program code is given below.

Coding − FakeCallLogReaderSpout.java

import java.util.\*;

//import storm tuple packages

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;

//import Spout interface packages

import backtype.storm.topology.IRichSpout;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.spout.SpoutOutputCollector;

import backtype.storm.task.TopologyContext;

//Create a class FakeLogReaderSpout which implement IRichSpout interface

to access functionalities

public class FakeCallLogReaderSpout implements IRichSpout {

//Create instance for SpoutOutputCollector which passes tuples to bolt.

private SpoutOutputCollector collector;

private boolean completed = false;

//Create instance for TopologyContext which contains topology data.

private TopologyContext context;

//Create instance for Random class.

private Random randomGenerator = new Random();

private Integer idx = 0;

@Override

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

this.context = context;

this.collector = collector;

}

@Override

public void nextTuple() {

if(this.idx <= 1000) {

List<String> mobileNumbers = new ArrayList<String>();

mobileNumbers.add("1234123401");

mobileNumbers.add("1234123402");

mobileNumbers.add("1234123403");

mobileNumbers.add("1234123404");

Integer localIdx = 0;

while(localIdx++ < 100 && this.idx++ < 1000) {

String fromMobileNumber = mobileNumbers.get(randomGenerator.nextInt(4));

String toMobileNumber = mobileNumbers.get(randomGenerator.nextInt(4));

while(fromMobileNumber == toMobileNumber) {

toMobileNumber = mobileNumbers.get(randomGenerator.nextInt(4));

}

Integer duration = randomGenerator.nextInt(60);

this.collector.emit(new Values(fromMobileNumber, toMobileNumber, duration));

}

}

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("from", "to", "duration"));

}

//Override all the interface methods

@Override

public void close() {}

public boolean isDistributed() {

return false;

}

@Override

public void activate() {}

@Override

public void deactivate() {}

@Override

public void ack(Object msgId) {}

@Override

public void fail(Object msgId) {}

@Override

public Map<String, Object> getComponentConfiguration() {

return null;

}

}

[Explore our](https://www.tutorialspoint.com/latest/courses?utm_source=tutorialspoint&utm_medium=tutorials_3p&utm_campaign=internal" \t "_blank)**[latest online courses](https://www.tutorialspoint.com/latest/courses?utm_source=tutorialspoint&utm_medium=tutorials_3p&utm_campaign=internal" \t "_blank)**[and learn new skills at your own pace. Enroll and become a certified expert to boost your career.](https://www.tutorialspoint.com/latest/courses?utm_source=tutorialspoint&utm_medium=tutorials_3p&utm_campaign=internal" \t "_blank)

Bolt Creation

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

IRichBolt interface has the following methods −

* **prepare** − Provides the bolt with an environment to execute. The executors will run this method to initialize the spout.
* **execute** − Process a single tuple of input.
* **cleanup** − Called when a bolt is going to shutdown.
* **declareOutputFields** − Declares the output schema of the tuple.

Prepare

The signature of the **prepare** method is as follows −

prepare(Map conf, TopologyContext context, OutputCollector collector)

* **conf** − Provides Storm configuration for this bolt.
* **context** − Provides complete information about the bolt place within the topology, its task id, input and output information, etc.
* **collector** − Enables us to emit the processed tuple.

execute

The signature of the **execute** method is as follows −

execute(Tuple tuple)

Here **tuple** is the input tuple to be processed.

The **execute** method processes a single tuple at a time. The tuple data can be accessed by getValue method of Tuple class. It is not necessary to process the input tuple immediately. Multiple tuple can be processed and output as a single output tuple. The processed tuple can be emitted by using the OutputCollector class.

cleanup

The signature of the **cleanup** method is as follows −

cleanup()

declareOutputFields

The signature of the **declareOutputFields** method is as follows −

declareOutputFields(OutputFieldsDeclarer declarer)

Here the parameter **declarer** is used to declare output stream ids, output fields, etc.

This method is used to specify the output schema of the tuple

Call log Creator Bolt

Call log creator bolt receives the call log tuple. The call log tuple has caller number, receiver number, and call duration. This bolt simply creates a new value by combining the caller number and the receiver number. The format of the new value is "Caller number – Receiver number" and it is named as new field, "call". The complete code is given below.

Coding − CallLogCreatorBolt.java

//import util packages

import java.util.HashMap;

import java.util.Map;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

//import Storm IRichBolt package

import backtype.storm.topology.IRichBolt;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Tuple;

//Create a class CallLogCreatorBolt which implement IRichBolt interface

public class CallLogCreatorBolt implements IRichBolt {

//Create instance for OutputCollector which collects and emits tuples to produce output

private OutputCollector collector;

@Override

public void prepare(Map conf, TopologyContext context, OutputCollector collector) {

this.collector = collector;

}

@Override

public void execute(Tuple tuple) {

String from = tuple.getString(0);

String to = tuple.getString(1);

Integer duration = tuple.getInteger(2);

collector.emit(new Values(from + " - " + to, duration));

}

@Override

public void cleanup() {}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("call", "duration"));

}

@Override

public Map<String, Object> getComponentConfiguration() {

return null;

}

}

Call log Counter Bolt

Call log counter bolt receives call and its duration as a tuple. This bolt initializes a dictionary (Map) object in the prepare method. In **execute** method, it checks the tuple and creates a new entry in the dictionary object for every new “call” value in the tuple and sets a value 1 in the dictionary object. For the already available entry in the dictionary, it just increment its value. In simple terms, this bolt saves the call and its count in the dictionary object. Instead of saving the call and its count in the dictionary, we can also save it to a datasource. The complete program code is as follows −

Coding − CallLogCounterBolt.java

import java.util.HashMap;

import java.util.Map;

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;

import backtype.storm.task.OutputCollector;

import backtype.storm.task.TopologyContext;

import backtype.storm.topology.IRichBolt;

import backtype.storm.topology.OutputFieldsDeclarer;

import backtype.storm.tuple.Tuple;

public class CallLogCounterBolt implements IRichBolt {

Map<String, Integer> counterMap;

private OutputCollector collector;

@Override

public void prepare(Map conf, TopologyContext context, OutputCollector collector) {

this.counterMap = new HashMap<String, Integer>();

this.collector = collector;

}

@Override

public void execute(Tuple tuple) {

String call = tuple.getString(0);

Integer duration = tuple.getInteger(1);

if(!counterMap.containsKey(call)){

counterMap.put(call, 1);

}else{

Integer c = counterMap.get(call) + 1;

counterMap.put(call, c);

}

collector.ack(tuple);

}

@Override

public void cleanup() {

for(Map.Entry<String, Integer> entry:counterMap.entrySet()){

System.out.println(entry.getKey()+" : " + entry.getValue());

}

}

@Override

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("call"));

}

@Override

public Map<String, Object> getComponentConfiguration() {

return null;

}

}

Creating Topology

The Storm topology is basically a Thrift structure. TopologyBuilder class provides simple and easy methods to create complex topologies. The TopologyBuilder class has methods to set spout **(setSpout)** and to set bolt **(setBolt)**. Finally, TopologyBuilder has createTopology to create topology. Use the following code snippet to create a topology −

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("call-log-reader-spout", new FakeCallLogReaderSpout());

builder.setBolt("call-log-creator-bolt", new CallLogCreatorBolt())

.shuffleGrouping("call-log-reader-spout");

builder.setBolt("call-log-counter-bolt", new CallLogCounterBolt())

.fieldsGrouping("call-log-creator-bolt", new Fields("call"));

**shuffleGrouping** and **fieldsGrouping** methods help to set stream grouping for spout and bolts.

Local Cluster

For development purpose, we can create a local cluster using "LocalCluster" object and then submit the topology using "submitTopology" method of "LocalCluster" class. One of the arguments for "submitTopology" is an instance of "Config" class. The "Config" class is used to set configuration options before submitting the topology. This configuration option will be merged with the cluster configuration at run time and sent to all task (spout and bolt) with the prepare method. Once topology is submitted to the cluster, we will wait 10 seconds for the cluster to compute the submitted topology and then shutdown the cluster using “shutdown” method of "LocalCluster". The complete program code is as follows −

Coding − LogAnalyserStorm.java

import backtype.storm.tuple.Fields;

import backtype.storm.tuple.Values;

//import storm configuration packages

import backtype.storm.Config;

import backtype.storm.LocalCluster;

import backtype.storm.topology.TopologyBuilder;

//Create main class LogAnalyserStorm submit topology.

public class LogAnalyserStorm {

public static void main(String[] args) throws Exception{

//Create Config instance for cluster configuration

Config config = new Config();

config.setDebug(true);

//

TopologyBuilder builder = new TopologyBuilder();

builder.setSpout("call-log-reader-spout", new FakeCallLogReaderSpout());

builder.setBolt("call-log-creator-bolt", new CallLogCreatorBolt())

.shuffleGrouping("call-log-reader-spout");

builder.setBolt("call-log-counter-bolt", new CallLogCounterBolt())

.fieldsGrouping("call-log-creator-bolt", new Fields("call"));

LocalCluster cluster = new LocalCluster();

cluster.submitTopology("LogAnalyserStorm", config, builder.createTopology());

Thread.sleep(10000);

//Stop the topology

cluster.shutdown();

}

}

Building and Running the Application

The complete application has four Java codes. They are −

* FakeCallLogReaderSpout.java
* CallLogCreaterBolt.java
* CallLogCounterBolt.java
* LogAnalyerStorm.java

The application can be built using the following command −

javac -cp “/path/to/storm/apache-storm-0.9.5/lib/\*” \*.java

The application can be run using the following command −

java -cp “/path/to/storm/apache-storm-0.9.5/lib/\*”:. LogAnalyserStorm

Output

Once the application is started, it will output the complete details about the cluster startup process, spout and bolt processing, and finally, the cluster shutdown process. In "CallLogCounterBolt", we have printed the call and its count details. This information will be displayed on the console as follows −

1234123402 - 1234123401 : 78

1234123402 - 1234123404 : 88

1234123402 - 1234123403 : 105

1234123401 - 1234123404 : 74

1234123401 - 1234123403 : 81

1234123401 - 1234123402 : 81

1234123403 - 1234123404 : 86

1234123404 - 1234123401 : 63

1234123404 - 1234123402 : 82

1234123403 - 1234123402 : 83

1234123404 - 1234123403 : 86

1234123403 - 1234123401 : 93

Non-JVM languages

Storm topologies are implemented by Thrift interfaces which makes it easy to submit topologies in any language. Storm supports Ruby, Python and many other languages. Let’s take a look at python binding.

Python Binding

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. Storm supports Python to implement its topology. Python supports emitting, anchoring, acking, and logging operations.

As you know, bolts can be defined in any language. Bolts written in another language are executed as sub-processes, and Storm communicates with those sub-processes with JSON messages over stdin/stdout. First take a sample bolt WordCount that supports python binding.

public static class WordCount implements IRichBolt {

public WordSplit() {

super("python", "splitword.py");

}

public void declareOutputFields(OutputFieldsDeclarer declarer) {

declarer.declare(new Fields("word"));

}

}

Here the class **WordCount** implements the **IRichBolt** interface and running with python implementation specified super method argument "splitword.py". Now create a python implementation named "splitword.py".

import storm

class WordCountBolt(storm.BasicBolt):

def process(self, tup):

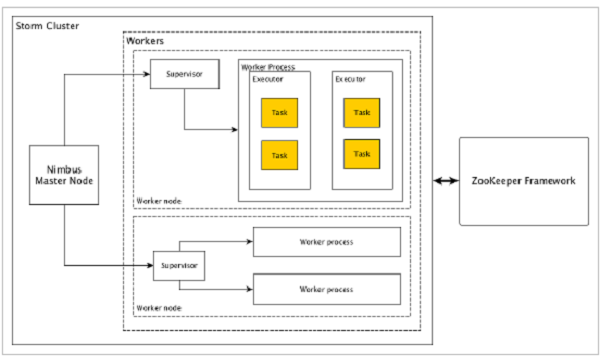
words = tup.values[0].split(" ")

for word in words:

storm.emit([word])

WordCountBolt().run()

This is the sample implementation for Python that counts the words in a given sentence. Similarly you can bind with other supporting languages as well.



Apache Storm has two type of nodes, **Nimbus** (master node) and **Supervisor** (worker node). Nimbus is the central component of Apache Storm. The main job of Nimbus is to run the Storm topology. Nimbus analyzes the topology and gathers the task to be executed. Then, it will distributes the task to an available supervisor.

A supervisor will have one or more worker process. Supervisor will delegate the tasks to worker processes. Worker process will spawn as many executors as needed and run the task. Apache Storm uses an internal distributed messaging system for the communication between nimbus and supervisors.

|  |  |
| --- | --- |
| **Components** | **Description** |
| Nimbus | Nimbus is a master node of Storm cluster. All other nodes in the cluster are called as **worker nodes**. Master node is responsible for distributing data among all the worker nodes, assign tasks to worker nodes and monitoring failures. |
| Supervisor | The nodes that follow instructions given by the nimbus are called as Supervisors. A **supervisor** has multiple worker processes and it governs worker processes to complete the tasks assigned by the nimbus. |
| Worker process | A worker process will execute tasks related to a specific topology. A worker process will not run a task by itself, instead it creates **executors** and asks them to perform a particular task. A worker process will have multiple executors. |
| Executor | An executor is nothing but a single thread spawn by a worker process. An executor runs one or more tasks but only for a specific spout or bolt. |
| Task | A task performs actual data processing. So, it is either a spout or a bolt. |
| ZooKeeper framework | Apache ZooKeeper is a service used by a cluster (group of nodes) to coordinate between themselves and maintaining shared data with robust synchronization techniques. Nimbus is stateless, so it depends on ZooKeeper to monitor the working node status.  ZooKeeper helps the supervisor to interact with the nimbus. It is responsible to maintain the state of nimbus and supervisor. |

Storm is stateless in nature. Even though stateless nature has its own disadvantages, it actually helps Storm to process real-time data in the best possible and quickest way.

Storm is *not entirely* stateless though. It stores its state in Apache ZooKeeper. Since the state is available in Apache ZooKeeper, a failed nimbus can be restarted and made to work from where it left. Usually, service monitoring tools like **monit** will monitor Nimbus and restart it if there is any failure.

Apache Storm also have an advanced topology called **Trident Topology** with state maintenance and it also provides a high-level API like Pig. We will discuss all these features in the coming chapters.