Learning Storm

Create real-time stream processing applications with Apache Storm

Ankit Jain   **Anand Nalya**

Learning Storm

Credits

Authors

Ankit Jain

Anand Nalya

Reviewers

Vinoth Kannan

Sonal Raj

Danijel Schiavuzzi

Commissioning Editor

Usha Iyer

Acquisition Editor

Llewellyn Rozario

Content Development Editor

Sankalp Pawar

Technical Editors

Menza Mathew

Siddhi Rane

Copy Editors

Sarang Chari

Mradula Hegde

Project Coordinator

Harshal Ved

Proofreaders

Simran Bhogal

Ameesha Green

Paul Hindle

Indexers

Hemangini Bari

Tejal Soni

Priya Subramani

Graphics

Abhinash Sahu

Production Coordinator

Saiprasad Kadam

Cover Work

Saiprasad Kadam

Table of Contents

About the Authors

Ankit Jain holds a Bachelor's degree in Computer Science Engineering. He has 4 years of experience in designing and architecting solutions for the Big Data domain and has been involved with several complex engagements. His technical strengths include Hadoop, Storm, S4, HBase, Hive, Sqoop, Flume, ElasticSearch, Machine Learning, Kafka, Spring, Java, and J2EE. He is currently employed with Impetus Infotech Pvt. Ltd.

He also shares his thoughts on his personal blog at http://ankitasblogger.blogspot.in/. You can follow him on Twitter at @mynameisanky. He spends most of his time reading books and playing with different technologies. When not at work, he spends time with his family and friends watching movies and playing games.

I would like to thank my family and colleagues for always being there for me. Special thanks to the Packt Publishing team; without you guys, this work would not have been possible.

Anand Nalya is a full stack engineer with over 8 years of extensive experience in designing, developing, deploying, and benchmarking Big Data and web-scale applications for both start-ups and enterprises. He focuses on reducing the complexity in getting things done with brevity in code.

He blogs about Big Data, web applications, and technology in general at http://anandnalya.com/. You can also follow him on Twitter at @anandnalya. When not working on projects, he can be found stargazing or reading.

I would like to thank my wife, Nidhi, for putting up with so many of my side projects and my family members who are always there for me. Special thanks to my colleagues who helped me validate the writing, and finally, the reviewers and editors at Packt Publishing, without whom this work would not have been possible.

About the Reviewers

Vinoth Kannan is a solution architect at WidasConcepts, Germany, that focuses on creating robust, highly scalable, real-time systems for storage, search, and analytics. He now works in Germany after his professional stints in France, Italy, and India.

Currently, he works extensively with open source frameworks based on Storm, Hadoop, and NoSQL databases. He has helped design and develop complex, real-time Big Data systems for some of the largest financial institutions and e-commerce companies.

He also co-organizes the Big Data User group in Karlsruhe and Stuttgart in Germany, and is a regular speaker at user group meets and international conferences on Big Data. He holds a double Master's degree in Communication Systems Engineering from Politecnico di Torino, Italy, and Grenoble Institute of Technology, France.

This is for my wonderful parents and my beloved wife, Sudha.

Sonal Raj is a Pythonista, technology enthusiast, and an entrepreneur. He is an engineer with dreams. He has been a research fellow at SERC, IISc, Bangalore, and he has pursued projects on distributed computing and real-time operations. He has spoken at PyCon India on Storm and Neo4J and has published articles and research papers in leading magazines and international journals. Presently, he works at Sigmoid Analytics, where he is actively involved in the development of machine-learning frameworks and Big Data solutions.

I am grateful to Ankit and Anand for patiently listening to my critiques, and I'd like to thank the open source community for keeping their passion alive and contributing to remarkable projects such as Storm. A special thank you to my parents, without whom I never would have grown to love learning as much as I do.

Danijel Schiavuzzi is a software engineer and technology enthusiast with a passionate interest in systems programming and distributed systems.

Currently, he works at Infobip, where he finds new usages for Storm and other Big Data technologies in the telecom domain on a daily basis. He has a strong focus on real-time data analytics, log processing, and external systems monitoring and alerting. He is passionate about open source, having contributed a few minor patches to Storm itself.

In his spare time, he enjoys reading a book, following space exploration and scientific and technological news, tinkering with various gadgets, listening and occasionally playing music, discovering old art movie masterpieces, and enjoying cycling around beautiful natural sceneries.

I would like to thank the Apache Storm community for developing such a great technology and making distributed computing more fun.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read, and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

Real-time data processing is no longer a luxury exercised by a few big companies but has become a necessity for businesses that want to compete, and Apache Storm is becoming the de facto standard to develop real-time processing pipelines. The key features of Storm are that it is horizontally scalable, fault-tolerant, and provides guaranteed message processing. Storm can solve various types of analytical problems, such as machine learning, log processing, and graph analysis.

Learning Storm will serve both as a getting-started guide for inexperienced developers and as a reference to implement advanced use cases with Storm for experienced developers. In the first two chapters, you will learn the basics of a Storm topology and various components of a Storm cluster. In the later chapters, you will learn how to build a Storm application that can interact with various other Big Data technologies and how to create transactional topologies. Finally, the last two chapters cover case studies for log processing and machine learning.

What this book covers

[Chapter 1](), *Setting Up Storm on a Single Machine*, gives an introduction to Storm and its components, followed by setting up a single-node Storm cluster, developing a sample Storm topology, and deploying it on a single-node cluster.

[Chapter 2](), *Setting Up a Storm Cluster*, covers the deployment of Storm in the cluster, deploys sample topology on a Storm cluster, discusses how we can achieve parallelism in Storm and how we can change the parallelism of the Storm topology in runtime, and even covers the basic Storm commands.

[Chapter 3](), *Monitoring the Storm Cluster*, introduces you to various ways of monitoring a Storm cluster, including the Storm UI and the Nimbus thrift client.

[Chapter 4](), *Storm and Kafka Integration*, introduces Apache Kafka, a message-queuing system, and shows how to integrate it with Storm to interact with data coming from external systems.

[Chapter 5](), *Exploring High-level Abstraction in Storm with Trident*, gives an introduction to Trident's function, filter, projection, aggregator, and repartitioning operations. It also covers a description of the transactional, non-transactional, and opaque transactional topologies. At the end, we cover how we can develop the sample Trident topology and how we can use the distributed RPC feature.

[Chapter 6](#), *Integration of Storm with Batch Processing Tools*, shows you how to integrate Storm with Hadoop using the Storm-YARN framework.

[Chapter 7](#), *Integrating Storm with JMX, Ganglia, HBase, and Redis*, shows you how to integrate Storm with various other Big Data technologies. It also focuses on how we can publish Storm's JVM metrics on Ganglia.

[Chapter 8](#), *Log Processing with Storm*, covers a sample log processing application in which, we parse Apache web server logs and generate some business information from logfiles.

[Chapter 9](#), *Machine Learning*, walks you through a case study of implementing a machine learning topology in Storm.

What you need for this book

All of the code in this book has been tested on CentOS 6.4. It will run on other variants of Linux and Windows as well with respective changes in commands.

We have tried to keep the chapters self-contained, and the setup and installation of all the software used in each chapter is included in the chapter itself. The following software packages are used throughout the book:

- CentOS 6.4
- Oracle JDK 6/7
- Apache ZooKeeper 3.4.5
- Apache Storm 0.9.0.1
- Eclipse or Spring Tool Suite

Who this book is for

If you are a Java developer who wants to enter the world of real-time stream processing applications using Apache Storm, then this book is for you. No previous experience in Storm is required as this book starts from the basics. After finishing this book, you will be able to develop simple Storm applications.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The LearningStormBolt class extends the serialized BaseRichBolt class."

A block of code is set as follows:

```
public void open(Map conf, TopologyContext context,SpoutOutputCollector spoutOutputCollector) {
    this.spoutOutputCollector = spoutOutputCollector;
}
```

Any command-line input or output is written as follows:

 # bin/storm nimbus

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Specify com.learningstorm as **Group Id** and storm-example as **Artifact Id**."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Setting Up Storm on a Single Machine

With the exponential growth in the amount of data being generated and advanced data-capturing capabilities, enterprises are facing the challenge of making sense out of this mountain of raw data. On the **batch processing** front, Hadoop has emerged as the go-to framework to deal with Big Data. Until recently, there has been a void when one looks for frameworks to build real-time stream processing applications. Such applications have become an integral part of a lot of businesses as they enable them to respond swiftly to events and adapt to changing situations. Examples of this are monitoring social media to analyze public response to any new product that you launch and predicting the outcome of an election based on the sentiments of the election-related posts.

Apache Storm has emerged as the platform of choice for the industry leaders to develop such distributed, real-time, data processing platforms. It provides a set of primitives that can be used to develop applications that can process a very large amount of data in real time in a highly scalable manner.

Storm is to real-time processing what Hadoop is to batch processing. It is an open source software, currently being incubated at the Apache Software Foundation. Being in incubation does not mean that it is not yet ready for actual production. Indeed, it has been deployed to meet real-time processing needs by companies such as Twitter, Yahoo!, and Flipboard. Storm was first developed by Nathan Marz at BackType, a company that provided social search applications. Later, BackType was acquired by Twitter, and now it is a critical part of their infrastructure. Storm can be used for the following use cases:

- **Stream processing**: Storm is used to process a stream of data and update a variety of databases in real time. This processing occurs in real time and the processing speed needs to match the input data speed.
- **Continuous computation**: Storm can do continuous computation on data streams and stream the results into clients in real time. This might require processing each message as it comes or creating small batches over a little time. An example of continuous computation is streaming trending topics on Twitter into browsers.
- **Distributed RPC**: Storm can parallelize an intense query so that you can compute it in real time.

- **Real-time analytics**: Storm can analyze and respond to data that comes from different data sources as they happen in real time.

In this chapter, we will cover the following topics:

- Features of Storm
- Various components of a Storm cluster
- What is a Storm topology
- Local and remote operational modes to execute Storm topologies
- Setting up a development environment to develop a Storm topology
- Developing a sample topology
- Setting up a single-node Storm cluster and its prerequisites
- Deploying the sample topology

Features of Storm

The following are some of the features of Storm that make it a perfect solution to process streams of data in real time:

- **Fast**: Storm has been reported to process up to 1 million tuples per second per node.
- **Horizontally scalable**: Being fast is a necessary feature to build a high volume/velocity data processing platform, but a single-node will have an upper limit on the number of events that it can process per second. A node represents a single machine in your setup that execute Storm applications. Storm, being a distributed platform, allows you to add more nodes to your Storm cluster and increase the processing capacity of your application. Also, it is linearly scalable, which means that you can double the processing capacity by doubling the nodes.
- **Fault tolerant**: Units of work are executed by worker processes in a Storm cluster. When a worker dies, Storm will restart that worker, and if the node on which the worker is running dies, Storm will restart that worker on some other node in the cluster. The descriptions of the worker process is mentioned in the *Configuring the parallelism of a topology* section of Chapter 2, *Setting Up a Storm Cluster*.
- **Guaranteed data processing**: Storm provides strong guarantees that each message passed on to it to process will be processed at least once. In the event of failures, Storm will replay the lost tuples. Also, it can be configured so that each message will be processed only once.
- **Easy to operate**: Storm is simple to deploy and manage. Once the cluster is deployed, it requires little maintenance.

- **Programming language agnostic**: Even though the Storm platform runs on Java Virtual Machine, the applications that run over it can be written in any programming language that can read and write to standard input and output streams.

Storm components

A Storm cluster follows a master-slave model where the master and slave processes are coordinated through ZooKeeper. The following are the components of a Storm cluster.

Nimbus

The Nimbus node is the master in a Storm cluster. It is responsible for distributing the application code across various worker nodes, assigning tasks to different machines, monitoring tasks for any failures, and restarting them as and when required.

Nimbus is stateless and stores all of its data in ZooKeeper. There is a single Nimbus node in a Storm cluster. It is designed to be fail-fast, so when Nimbus dies, it can be restarted without having any effects on the already running tasks on the worker nodes. This is unlike Hadoop, where if the JobTracker dies, all the running jobs are left in an inconsistent state and need to be executed again.

Supervisor nodes

Supervisor nodes are the worker nodes in a Storm cluster. Each supervisor node runs a supervisor daemon that is responsible for creating, starting, and stopping worker processes to execute the tasks assigned to that node. Like Nimbus, a supervisor daemon is also fail-fast and stores all of its state in ZooKeeper so that it can be restarted without any state loss. A single supervisor daemon normally handles multiple worker processes running on that machine.

The ZooKeeper cluster

In any distributed application, various processes need to coordinate with each other and share some configuration information. ZooKeeper is an application that provides all these services in a reliable manner. Being a distributed application, Storm also uses a ZooKeeper cluster to coordinate various processes. All of the states associated with the cluster and the various tasks submitted to the Storm are stored in ZooKeeper. Nimbus and supervisor nodes do not communicate directly with each other but through ZooKeeper. As all data is stored in ZooKeeper,

both Nimbus and the supervisor daemons can be killed abruptly without adversely affecting the cluster.

The following is an architecture diagram of a Storm cluster:
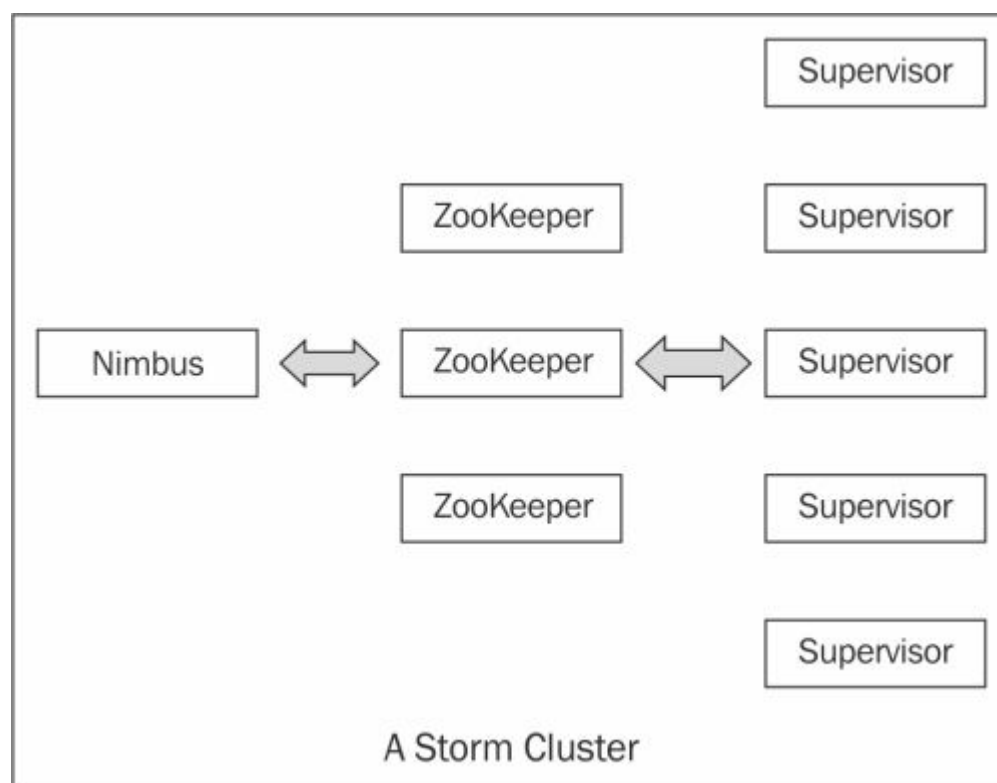
A Storm cluster's architecture

The Storm data model

The basic unit of data that can be processed by a Storm application is called a **tuple**. Each tuple consists of a predefined list of fields. The value of each field can be a byte, char, integer, long, float, double, Boolean, or byte array. Storm also provides an API to define your own data types, which can be serialized as fields in a tuple.

A tuple is dynamically typed, that is, you just need to define the names of the fields in a tuple and not their data type. The choice of dynamic typing helps to simplify the API and makes it easy to use. Also, since a processing unit in Storm can process multiple types of tuples, it's not practical to declare field types.

Each of the fields in a tuple can be accessed by its name getValueByField(String) or its positional index getValue(int) in the tuple. Tuples also provide convenient methods such as getIntegerByField(String)that save you from typecasting the objects. For example, if you have a

Fraction(numerator, denominator) tuple, representing fractional numbers, then you can get the value of the numerator by either using getIntegerByField("numerator") or getInteger(0).

You can see the full set of operations supported by backtype.storm.tuple.backtype.storm.tuple.Tuple in the javadoc located at https://storm.incubator.apache.org/apidocs/backtype/storm/tuple/Tuple.html.

Definition of a Storm topology

In Storm terminology, a topology is an abstraction that defines the graph of the computation. You create a Storm topology and deploy it on a Storm cluster to process the data. A topology can be represented by a direct acyclic graph, where each node does some kind of processing and forwards it to the next node(s) in the flow. The following is a sample Storm topology:
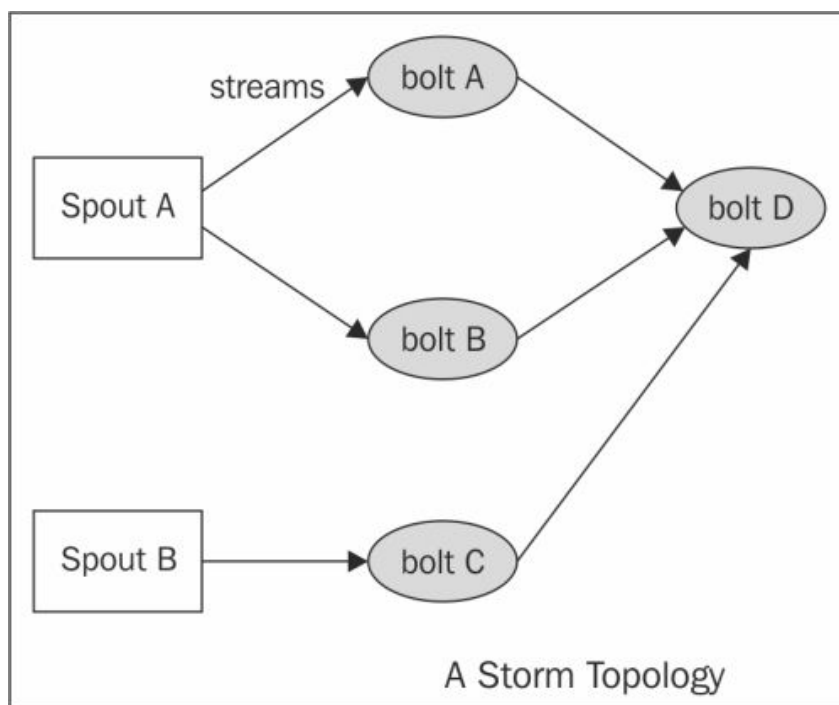
A Storm Topology

Graphical representation of the Storm topology

The following are the components of a Storm topology:

- **Stream**: The key abstraction in Storm is that of a stream. A **stream** is an unbounded sequence of tuples that can be processed in parallel by Storm. Each stream can be processed by a single or multiple types of bolts (the processing units in Storm, which are defined later in this section). Thus, Storm can also be viewed as a platform to transform streams. In the preceding diagram, streams are represented by arrows.

Each stream in a Storm application is given an ID and the bolts can produce and consume tuples from these streams on the basis of their ID. Each stream also has an associated schema for the tuples that will flow through it.

- **Spout**: A spout is the source of tuples in a Storm topology. It is responsible for reading or listening to data from an external source, for example, by reading from a logfile or listening for new messages in a queue and publishing them—emitting, in Storm terminology—into streams. A spout can emit multiple streams, each of different schemas. For example, it can read 10-field records from a logfile and emit them as different streams of 7-tuples and 4-tuples each.

  The backtype.storm.spout.ISpout interface is the interface used to define spouts. If you are writing your topology in Java, then you should use backtype.storm.topology.IRichSpout as it declares methods to use the TopologyBuilder API. Whenever a spout emits a tuple, Storm tracks all the tuples generated while processing this tuple, and when the execution of all the tuples in the graph of this source tuple is complete, it will send back an acknowledgement to the spout. This tracking happens only if a message ID was provided while emitting the tuple. If null was used as message ID, this tracking will not happen.

  A tuple-processing timeout can also be defined for a topology, and if a tuple is not processed within the specified timeout, a fail message will be sent back to the spout. Again, this will happen only if you define a message ID. A small performance gain can be extracted out of Storm at the risk of some data loss by disabling the message acknowledgements, which can be done by skipping the message ID while emitting tuples.

  The important methods of spout are:

  - nextTuple(): This method is called by Storm to get the next tuple from the input source. Inside this method, you will have the logic of reading data from the external sources and emitting them to an instance of backtype.storm.spout.ISpoutOutputCollector. The schema for streams can be declared by using the declareStream method of backtype.storm.topology.OutputFieldsDeclarer.

    If a spout wants to emit data to more than one stream, it can declare multiple streams using the declareStream method and specify a stream ID while emitting the tuple. If there are no more tuples to emit at the moment, this method would not be blocked. Also, if this method does not emit a tuple, then Storm will wait for 1 millisecond before calling it again. This waiting time can be configured using the topology.sleep.spout.wait.strategy.time.ms setting.

- **ack(Object msgId):** This method is invoked by Storm when the tuple with the given message ID is completely processed by the topology. At this point, the user should mark the message as processed and do the required cleaning up such as removing the message from the message queue so that it does not get processed again.

- **fail(Object msgId):** This method is invoked by Storm when it identifies that the tuple with the given message ID has not been processed successfully or has timed out of the configured interval. In such scenarios, the user should do the required processing so that the messages can be emitted again by the nextTuple method. A common way to do this is to put the message back in the incoming message queue.

- **open():** This method is called only once—when the spout is initialized. If it is required to connect to an external source for the input data, define the logic to connect to the external source in the open method, and then keep fetching the data from this external source in the nextTuple method to emit it further.

Another point to note while writing your spout is that none of the methods should be blocking, as Storm calls all the methods in the same thread. Every spout has an internal buffer to keep track of the status of the tuples emitted so far. The spout will keep the tuples in this buffer until they are either acknowledged or failed, calling the ack or fail method respectively. Storm will call the nextTuple method only when this buffer is not full.

- **Bolt:** A bolt is the processing powerhouse of a Storm topology and is responsible for transforming a stream. Ideally, each bolt in the topology should be doing a simple transformation of the tuples, and many such bolts can coordinate with each other to exhibit a complex transformation.

The backtype.storm.task.IBolt interface is preferably used to define bolts, and if a topology is written in Java, you should use the backtype.storm.topology.IRichBolt interface. A bolt can subscribe to multiple streams of other components—either spouts or other bolts—in the topology and similarly can emit output to multiple streams. Output streams can be declared using the declareStream method of backtype.storm.topology.OutputFieldsDeclarer.

The important methods of a bolt are:

- **execute(Tuple input):** This method is executed for each tuple that comes through the subscribed input streams. In this method, you can do whatever processing is required for the tuple and then produce the output either in the form of emitting more tuples to the declared output streams or other things such as persisting the results in a database.

You are not required to process the tuple as soon as this method is called, and the tuples can be held until required. For example, while joining two streams, when a tuple arrives, you can hold it until its counterpart also comes, and then you can emit the joined tuple.

The metadata associated with the tuple can be retrieved by the various methods defined in the Tuple interface. If a message ID is associated with a tuple, the execute method must publish an ack or fail event using OutputCollector for the bolt or else Storm will not know whether the tuple was processed successfully or not. The backtype.storm.topology.IBasicBolt interface is a convenient interface that sends an acknowledgement automatically after the completion of the execute method. In the case that a fail event is to be sent, this method should throw backtype.storm.topology.FailedException.

- prepare(Map stormConf, TopologyContext context, OutputCollector collector): A bolt can be executed by multiple workers in a Storm topology. The instance of a bolt is created on the client machine and then serialized and submitted to Nimbus. When Nimbus creates the worker instances for the topology, it sends this serialized bolt to the workers. The work will desterilize the bolt and call the prepare method. In this method, you should make sure the bolt is properly configured to execute tuples now. Any state that you want to maintain can be stored as instance variables for the bolt that can be serialized/deserialized later.

Operation modes

Operation modes indicate how the topology is deployed in Storm. Storm supports two types of operation modes to execute the Storm topology

- **The local mode**: In the local mode, Storm topologies run on the local machine in a single JVM. This mode simulates a Storm cluster in a single JVM and is used for the testing and debugging of a topology.
- **The remote mode**: In the remote mode, we will use the Storm client to submit the topology to the master along with all the necessary code required to execute the topology. Nimbus will then take care of distributing your code.

Setting up your development environment

Before you can start developing Storm topologies, you must first check/set up your development environment, which involves installing the following software packages on your development computer:

- Java SDK 6
- Maven
- Git: Distributed version control
- Spring Tool Suite: IDE

The following installation steps are valid for CentOS, and going forward, all the commands used in this book are valid for CentOS.

Installing Java SDK 6

Perform the following steps to install the Java SDK 6 on your machine:

1. Download the Java SDK 6 RPM from Oracle's site
   (http://www.oracle.com/technetwork/java/javase/downloads/index.html).
2. Install the Java jdk-6u31-linux-amd64.rpm file on your CentOS machine using the following command:
3. **sudo rpm -ivh jdk-6u31-linux-amd64.rpm**
4. Add the environment variable in the ~/.bashrc file:
5. **export JAVA_HOME=/usr/java/jdk1.6.0_31/**
6. Add the path of the bin directory of the JDK in the PATH system environment variable in the ~/.bashrc file:
7. **export PATH=$JAVA_HOME/bin:$PATH**

Note

The PATH variable is the system variable that your operating system uses to locate the required executables from the command line or terminal window.

8. Run the following command to reload the bashrc file on the current login terminal:
9. **source ~/.bashrc**
10. Check the Java installation as follows:
11. **java -version**

The output of the preceding command is:

java version "1.6.0_31"
Java(TM) SE Runtime Environment (build 1.6.0_31-b04)
Java HotSpot(TM) 64-Bit Server VM (build 20.6-b01, mixed mode)

Installing Maven

Apache Maven is a software dependency management tool and is used to manage the project's build, reporting, and documentation. We are using this so that we do not need to download all the dependencies manually. Perform the following steps to install the Maven on your machine:

1. Download the stable release of Maven from Maven's site (http://maven.apache.org/download.cgi).
2. Once you have downloaded the latest version, unzip it. Now, set the MAVEN_HOME environment variable in the ~/.bashrc file to make the setting up of Maven easier.
3. **export MAVEN_HOME=/home/root/apache-maven-3.0.4**
4. Add the path to the bin directory of Maven in the $PATH environment variable in the ~/.bashrc file:
5. **export PATH=$JAVA_HOME/bin:$PATH:$MAVEN_HOME/bin**
6. Run the following command to reload the bashrc file on the current login terminal:
7. **source ~/.bashrc**
8. Check the Maven installation as follows:
9. **mvn –version**

   The following information will be displayed:

   Apache Maven 3.0.4 (r1232337; 2012-01-17 14:14:56+0530)
   Maven home: /home/root/apache-maven-3.0.4
   Java version: 1.6.0_31, vendor: Sun Microsystems Inc.
   Java home: /usr/java/jdk1.6.0_31/jre
   Default locale: en_US, platform encoding: UTF-8
   OS name: "linux", version: "2.6.32-279.22.1.el6.x86_64", arch: "amd64", family: "unix"

Installing Git – distributed version control

Git is one of the most used open source version control systems. It is used to track content such as files and directories and allows multiple users to work on the same file. Perform the following steps to install Git on your machine:

1. The command to install Git on a CentOS machine is:
2. **sudo yum install git**
3. Check the installation of Git using the following command:
4. **git --version**

The preceding command's output is:

git version 1.7.1

Installing the STS IDE

The STS IDE is an integrated development environment and is used to develop applications. We will be using this to develop all the examples in this book. Perform the following steps to install the STS IDE on your machine:

1. Download the latest version of STS from the Spring site (https://spring.io/tools/sts/all).
2. Once you have downloaded the latest version, unzip it.
3. Start the STS IDE.
4. Go to **Windows** | **Preferences** | **Maven** | **Installations** and add the path of maven-3.0.4, as shown in the following screenshot:
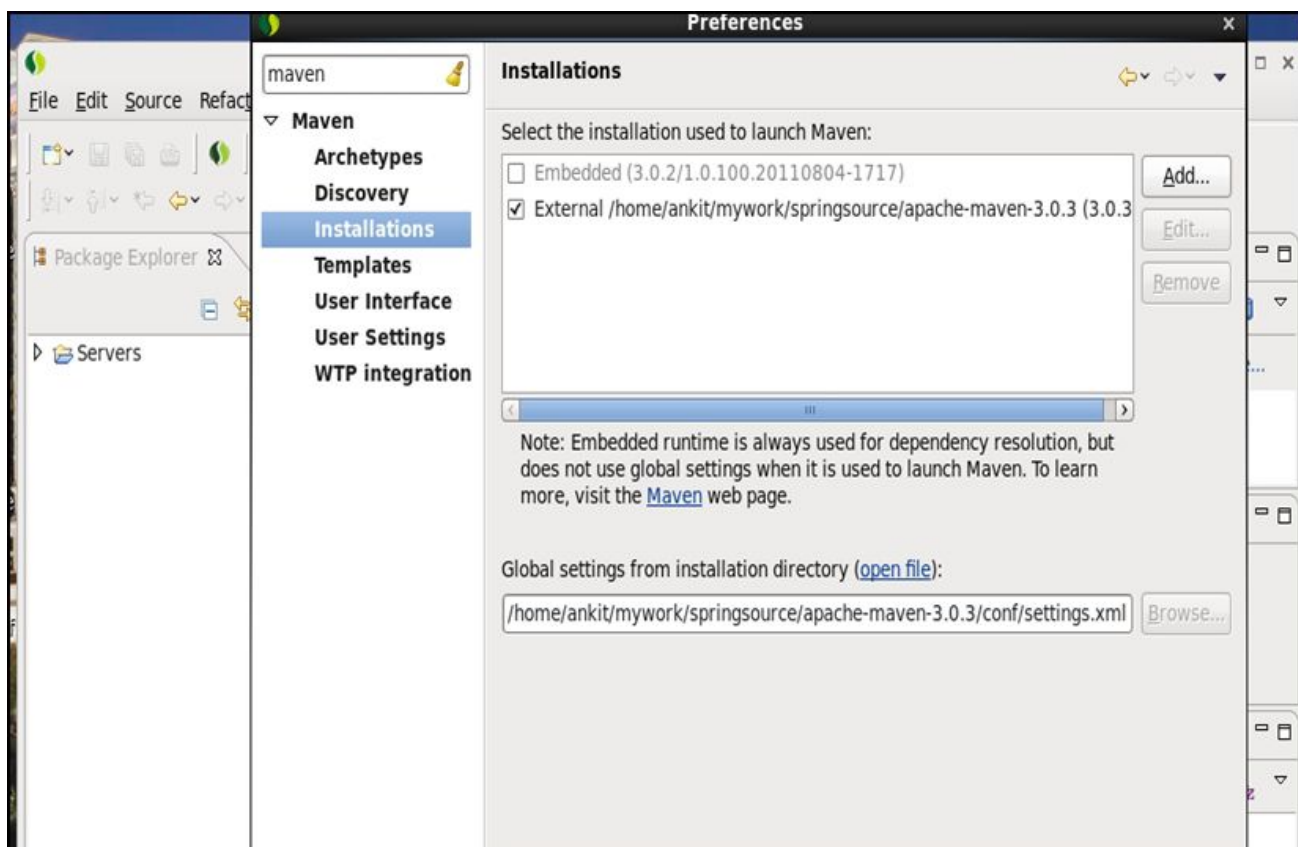
Add maven-3.0.4 to launch Maven

5. Go to **Window** | **Preferences** | **Java** | **Installed JREs** and add the path of **Java Runtime Environment 6** (**JRE 6**), as shown in the following screenshot:
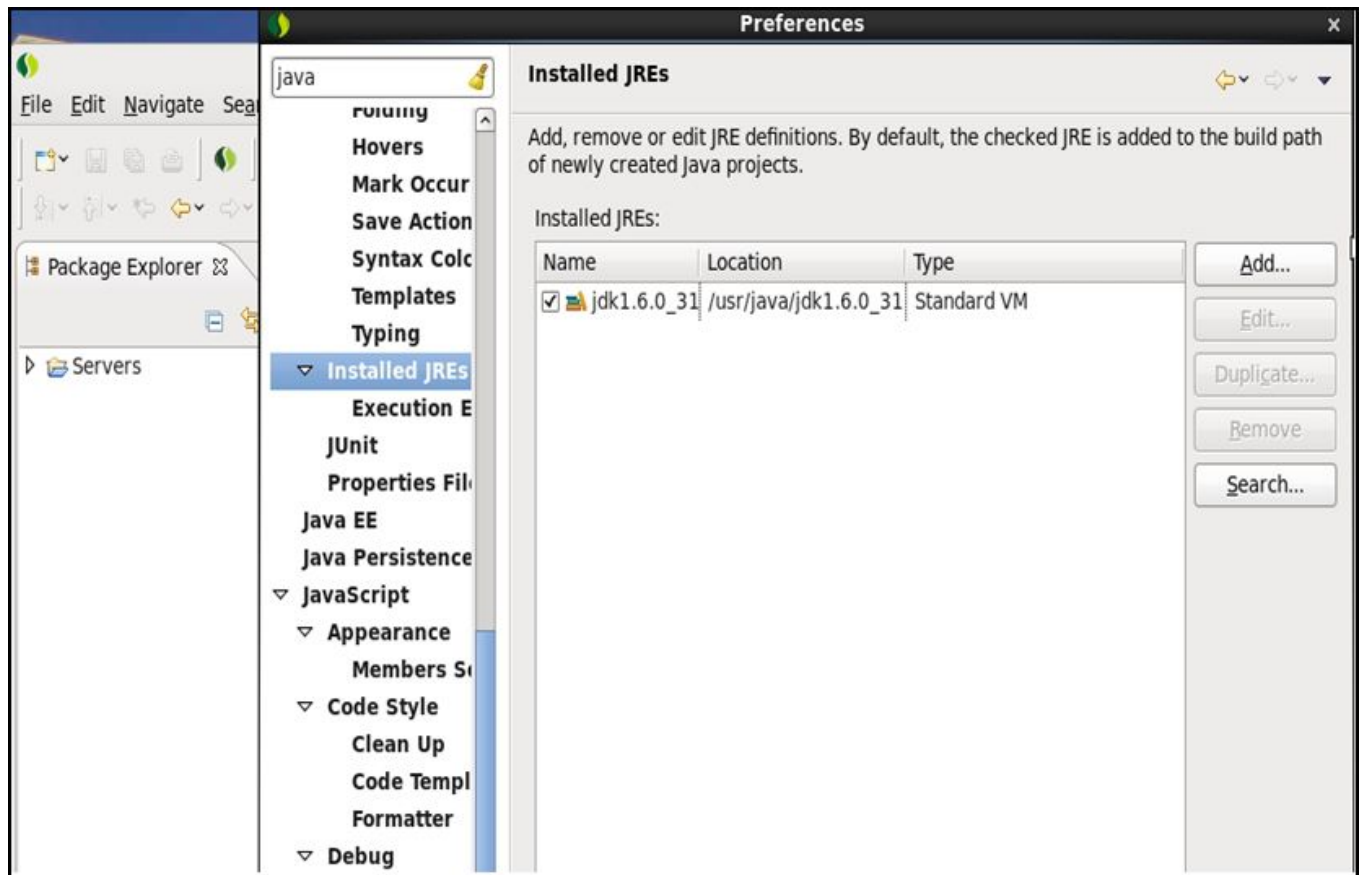
Add jdk1.6.0_31 to the build path

From now on, we will use the Spring Tool Suite to develop all the sample Storm topologies.

Developing a sample topology

The sample topology shown in the following diagram will cover how to create a basic Storm project, including a spout and bolt, build it, and execute it:

Hello World Topology

Spout → Bolt

This spout continuously emits strings as tuples chosen randomly from a map.

This bolt read the stream subscribed by a bolt and print the tuples value.
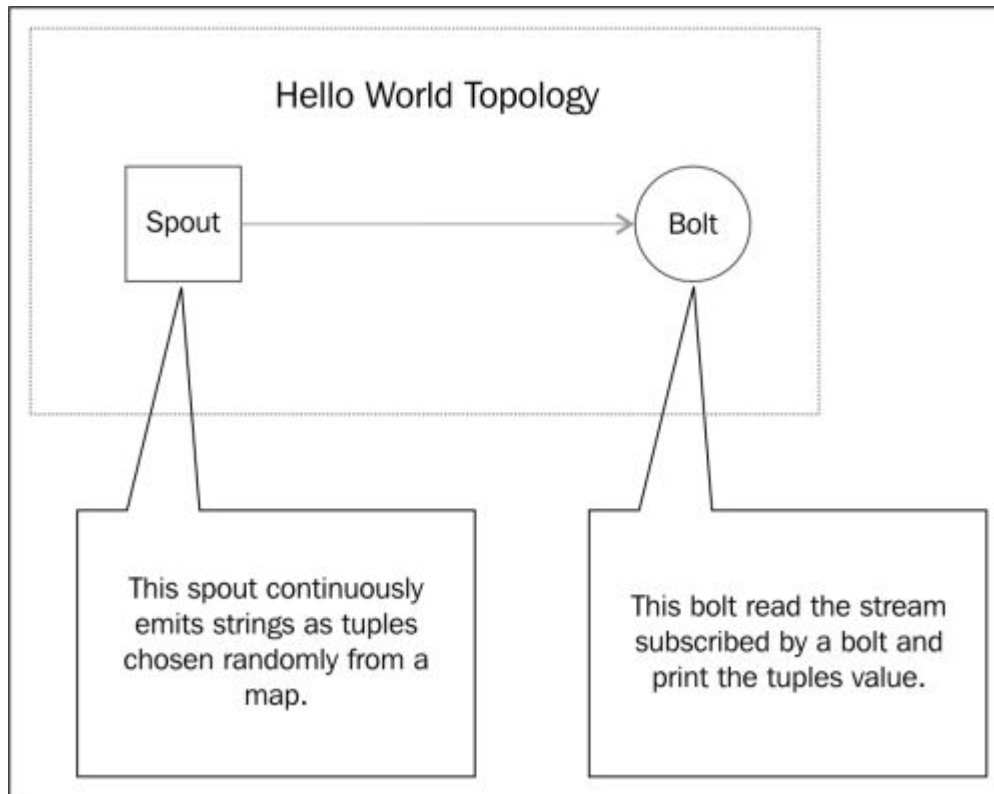
A sample Hello World topology

Perform the following steps to create and execute a sample topology:

1. Start your STS IDE and create a Maven project as shown in the following screenshot:
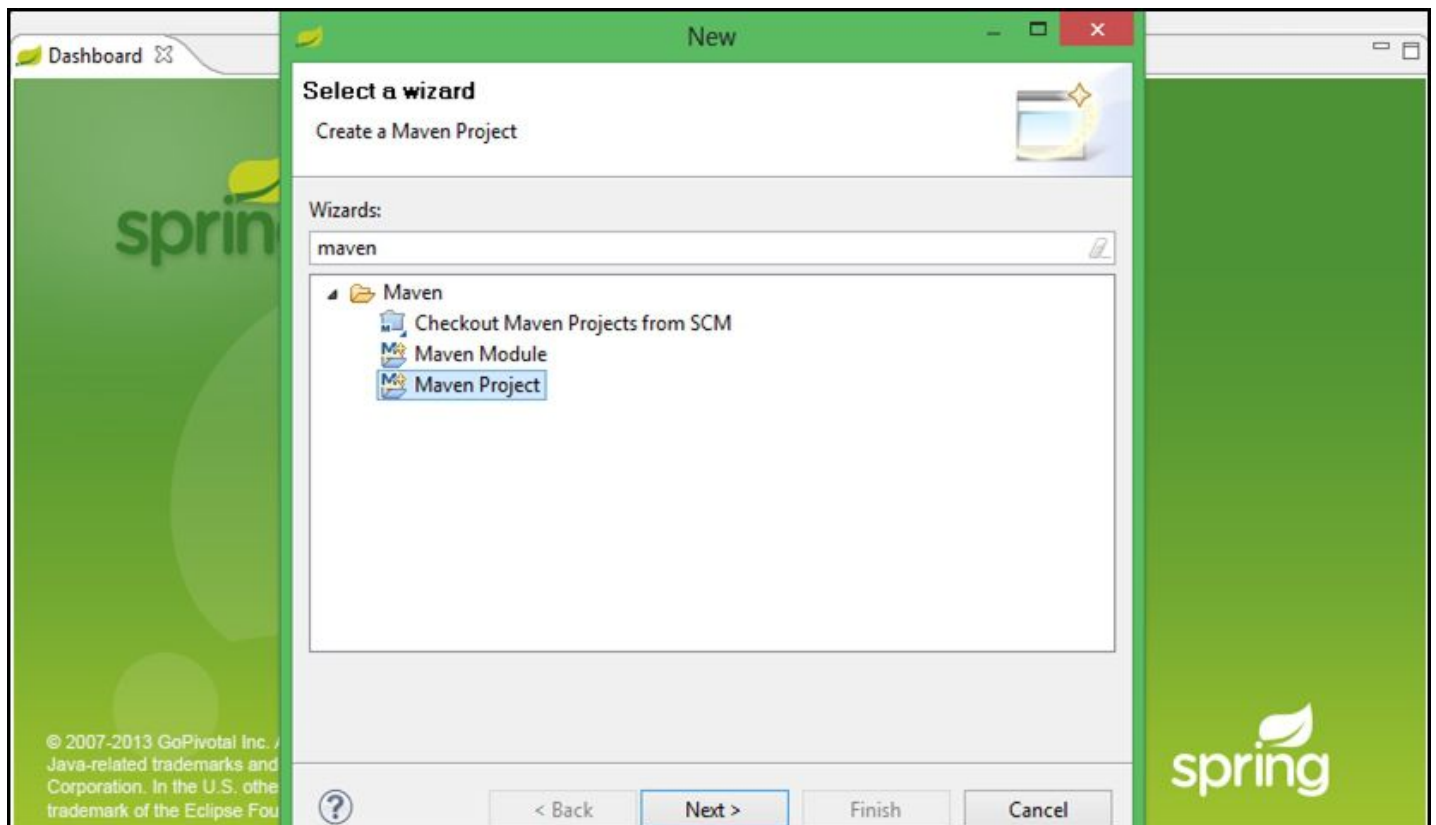
Create a Maven project

2. Specify com.learningstorm as **Group Id** and storm-example as **Artifact Id**, as shown in the following screenshot:
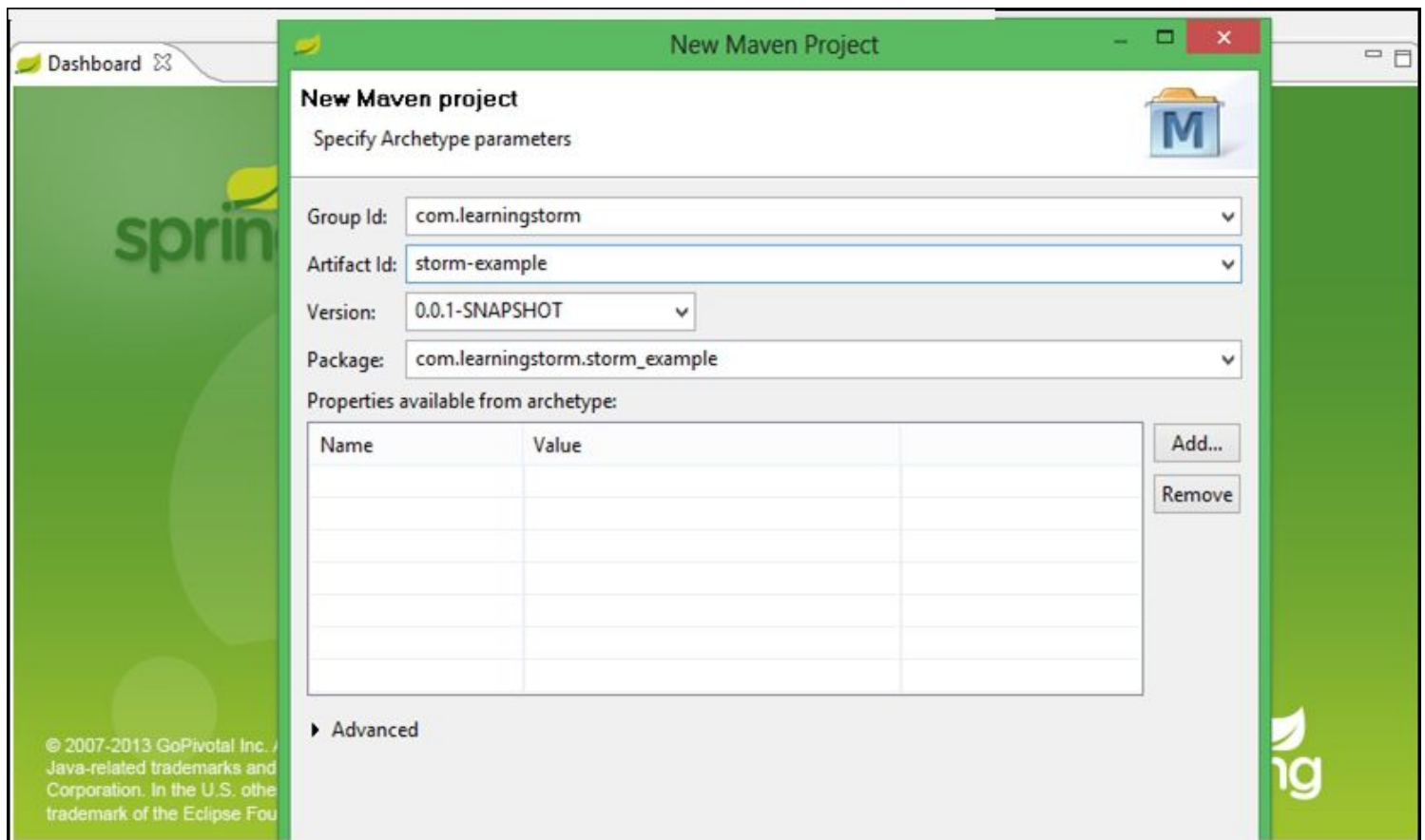
Specify Archetype Parameters

3. Add the following Maven dependencies in the pom.xml file:
4.   \<dependencies>
5.   \<dependency>
6.   \<groupId>junit\</groupId>
7.   \<artifactId>junit\</artifactId>
8.   \<version>3.8.1\</version>
9.   \<scope>test\</scope>
10.   \</dependency>
11.   \<dependency>
12.   \<groupId>storm\</groupId>
13.   \<artifactId>storm\</artifactId>
14.   \<version>0.9.0.1\</version>
15.   \<scope>provided\</scope>

16.   </dependency>
    </dependencies>

17. Add the following Maven repository in the pom.xml file:

18. <repositories>

19.   <repository>

20.   <id>clojars.org</id>

21.   <url>http://clojars.org/repo</url>

22.   </repository>

    </repositories>

23. Add the following Maven build plugins in the pom.xml file:

24. <build>

25.   <plugins>

26.   <plugin>

27.   <artifactId>maven-assembly-plugin</artifactId>

28.   <version>2.2.1</version>

29.   <configuration>

30.   <descriptorRefs>

31.   <descriptorRef>jar-with-dependencies</descriptorRef>

32.   </descriptorRefs>

33.   <archive>

34.   <manifest>

35.   <mainClass />

36.   </manifest>

37.   </archive>

38.   </configuration>

39.   <executions>

40.   <execution>

41.   <id>make-assembly</id>

42.   <phase>package</phase>

43.   <goals>

44.   <goal>single</goal>

45.   </goals>

46.   </execution>

47.   </executions>

48.   </plugin>

49.   </plugins>

    </build>

50. Write your first sample spout by creating a LearningStormSpout class in the com.learningstorm.storm_example package. The LearningStormSpout class extends the serialized BaseRichSpout class. This spout does not connect to an external source to fetch data but randomly generates the data and emits a continuous stream of records. The following is the source code of the LearningStormSpout class with an explanation:

```
51.    public class LearningStormSpout extends BaseRichSpout{
52.        private static final long serialVersionUID = 1L;
53.        private SpoutOutputCollectorspoutOutputCollector;
54.        private static final Map<Integer, String> map = new HashMap<Integer, String>();
55.        static {
56.            map.put(0, "google");
57.            map.put(1, "facebook");
58.            map.put(2, "twitter");
59.            map.put(3, "youtube");
60.            map.put(4, "linkedin");
61.        }
62.        public void open(Map conf, TopologyContext context, SpoutOutputCollector spoutOutputCollector) {
63.            // Open the spout
64.            this.spoutOutputCollector = spoutOutputCollector;
65.        }
66.
67.        public void nextTuple() {
68.            // Storm cluster repeatedly calls this method to emit a continuous
69.            // stream of tuples.
70.            final Random rand = new Random();
71.            // generate the random number from 0 to 4.
72.            int randomNumber = rand.nextInt(5);
73.            spoutOutputCollector.emit(new Values(map.get(randomNumber)));
74.
75.        }
76.
77.        public void declareOutputFields(OutputFieldsDeclarer declarer) {
78.            // emit the tuple with field "site"
79.            declarer.declare(new Fields("site"));
80.        }
    }
```

81. Write your first sample bolt by creating a LearningStormBolt class within the same package. The LearningStormBolt class extends the serialized BaseRichBolt class. This bolt will consume the tuples emitted by LearningStormSpout spout and will print the value of the field "site" on the console. The following is the source code of the LearningStormBolt class with an explanation:

82.   public class LearningStormBolt extends BaseBasicBolt{

83.

84.   private static final long serialVersionUID = 1L;

85.

86.   public void execute(Tuple input, BasicOutputCollector collector) {

87.   // fetched the field "site" from input tuple.

88.   String test = input.getStringByField("site");

89.   // print the value of field "site" on console.

90.   System.out.println("Name of input site is : " + test);

91.   }

92.

93.   public void declareOutputFields(OutputFieldsDeclarer declarer) {

94.

95.   }
    }

96. Create a main LearningStormTopology class within the same package. This class creates an instance of the spout and bolt, classes and chained together using a TopologyBuilder class. The following is the implementation of the main class:

97.   public class LearningStormTopology {

98.   public static void main(String[] args) throws AlreadyAliveException, InvalidTopologyException {

99.   // create an instance of TopologyBuilder class

100.   TopologyBuilder builder = new TopologyBuilder();

101.   // set the spout class

102.   builder.setSpout("LearningStormSpout", new LearningStormSpout(), 2);

103.   // set the bolt class

104.   builder.setBolt("LearningStormBolt", new LearningStormBolt(),
    4).shuffleGrouping("LearningStormSpout");

105.

106.   Config conf = new Config();

107.   conf.setDebug(true);

108.   // create an instance of LocalCluster class for

109.   // executing topology in local mode.

110.   LocalCluster cluster = new LocalCluster();

```
111.
112.        // LearningStormTopolgy is the name of submitted topology.
113.        cluster.submitTopology("LearningStormToplogy", conf, builder.createTopology());
114.        try {
115.          Thread.sleep(10000);
116.        } catch (Exception exception) {
117.          System.out.println("Thread interrupted exception : " + exception);
118.        }
119.        // kill the LearningStormTopology
120.        cluster.killTopology("LearningStormToplogy");
121.        // shutdown the storm test cluster
122.        cluster.shutdown();
123.
124.    }
     }
```

125. Go to your project's home directory and run the following commands to execute the topology in the local mode:

126. **mvn compile exec:java -Dexec.classpathScope=compile**

127. **-**

128. **Dexec.mainClass=com.learningstorm.storm_example.LearningStormTopology**

Tip

Downloading the example code

Also, we can execute the topology by simply running the main class through the STS IDE.

In the preceding example, we used a utility called LocalCluster to execute the topology in a single JVM. The LocalCluster class simulates the Storm cluster and starts all the Storm processes in a single JVM.

We have submitted a topology in a simulated cluster by calling the submitTopology method of the LocalCluster class. The submitTopology method takes the name of a topology, a configuration for the topology, and then the topology itself as arguments.

The topology name is used to identify the topology in the Storm cluster. Hence, it is good practice to use a unique name for each topology.

Running the Storm infrastructure in local mode is useful when we want to test and debug the topology.

The upcoming sections will cover the deployment of ZooKeeper, Storm native dependencies, and Storm, and how we can submit the topology on a single-node Storm cluster.

Setting up ZooKeeper

This section describes how you can set up a ZooKeeper cluster. We are deploying ZooKeeper in standalone mode, but in the distributed cluster mode, it is always recommended that you should run a ZooKeeper ensemble of at least three nodes to support failover and high availability. Perform the following steps to set up ZooKeeper on your machine:

1. Download the latest stable ZooKeeper release from the ZooKeeper's site (http://www.apache.org/dyn/closer.cgi/zookeeper/); at this moment, the latest version is ZooKeeper 3.4.5.
2. Once you have downloaded the latest version, unzip it and set the ZK_HOME environment variable.
3. Create the configuration file, zoo.cfg, at the $ZK_HOME/conf directory using the following command:
4. **cd $ZK_HOME/conf**
5. **touch zoo.cfg**
6. Add the following three properties in the zoo.cfg file:
7. **tickTime=2000**
8. **dataDir=/tmp/zookeeper**
9. **clientPort=2181**

   The following are the definitions of each of these properties:

   - tickTime: This is the basic time unit in milliseconds used by ZooKeeper. It is used to send heartbeats and the minimum session timeout will be twice the tickTime value.
   - dataDir: This is an empty directory to store the in-memory database snapshots and transactional log.
   - clientPort: This is the port used to listen for client connections.

2. The command to start the ZooKeeper node is as follows:
3. **bin/zkServer.sh start**

   The following information is displayed:

JMX enabled by default
Using config: /home/root/zookeeper-3.4.5/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED

4. At this point, the following Java process must be started:

5. **jps**

   The following information is displayed:

   23074 QuorumPeerMain

6. The command to check the status of running the ZooKeeper node is as follows:

7. **bin/zkServer.sh status**

   The following information is displayed:

   JMX enabled by default
   Using config: ../conf/zoo.cfg
   Mode: standalone

Setting up Storm on a single development machine

This section describes you how to install Storm on a single machine. Download the latest stable Storm release from https://storm.incubator.apache.org/downloads.html; at the time of this writing, the latest version is storm-0.9.0.1. Perform the following steps to set up Storm on a single development machine:

1. Once you have downloaded the latest version, unzip it and set the STORM_HOME environment variable.

2. Perform the following steps to edit the storm.yaml configuration file:

3. **cd $STORM_HOME/conf**

4. **vi storm.yaml**

   Add the following information:

   storm.zookeeper.servers:
   **   - "127.0.0.1"**
   storm.zookeeper.port: 2181
   nimbus.host: "127.0.0.1"
   storm.local.dir: "/tmp/storm-data"
   java.library.path: "/usr/local/lib"

storm.messaging.transport: backtype.storm.messaging.netty.Context
supervisor.slots.ports:
    - 6700
    - 6701
    - 6702
    - 6703

5. The following is a definition of properties used in the storm.yaml file:

    - storm.zookeeper.servers: This property contains the IP addresses of ZooKeeper servers.
    - storm.zookeeper.port: This property contains the ZooKeeper client port.
    - storm.local.dir: The Nimbus and supervisor daemons require a directory on the local disk to store small amounts of state (such as JARs, CONFs, and more).
    - java.library.path: This is used to load the Java native libraries that Storm uses (ZeroMQ and JZMQ). The default location of Storm native libraries is /usr/local/lib: /opt/local/lib:/usr/lib.
    - nimbus.host: This specifies the IP address of the master (Nimbus) node:
    - supervisor.slots.ports: For every worker machine, we can configure how many workers run on that machine with this property. Each worker binds with a single port and uses that port to receive incoming messages.

2. Start the master node using the following commands:
3. **cd $STORM_HOME**
4. **bin/storm nimbus**
5. Start the supervisor node using the following commands:
6. **cd $STORM_HOME**
7. **bin/storm supervisor**

Deploying the sample topology on a single-node cluster

In the previous example, we executed the Storm topology in the local mode. Now, we will deploy the topology on the single-node Storm cluster.

1. We will first create a LearningStormSingleNodeTopology class within the same package. The following LearningStormSingleNodeTopology class will use the submitTopology method of the StormSubmitter class to deploy the topology on the Storm cluster:
2. public class LearningStormSingleNodeTopology {
3.   public static void main(String[] args) {
4.   TopologyBuilder builder = new TopologyBuilder();
5.   // set the spout class

6.   builder.setSpout("LearningStormSpout", new LearningStormSpout(), 4);

7.   // set the bolt class

8.   builder.setBolt("LearningStormBolt", new LearningStormBolt(),
2).shuffleGrouping("LearningStormSpout");

9.

10.   Config conf = new Config();

11.   conf.setNumWorkers(3);

12.   try {

13.   // This statement submit the topology on remote cluster.

14.   // args[0] = name of topology

15.   StormSubmitter.submitTopology(args[0], conf, builder.createTopology());

16.   }catch(AlreadyAliveException alreadyAliveException) {

17.   System.out.println(alreadyAliveException);

18.   } catch (InvalidTopologyException invalidTopologyException) {

19.   System.out.println(invalidTopologyException);

20.   }

21.   }
    }

22. Build your Maven project by running the following command on the project home directory:

23. **mvn clean install**

   The output of the preceding command is:

```
-------------------------------------------------------------------
[INFO] -------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] -------------------------------------------------------------
[INFO] Total time: 58.326s
[INFO] Finished at: Mon Jan 20 00:55:52 IST 2014
[INFO] Final Memory: 14M/116M
[INFO] -------------------------------------------------------------
```

24. We can deploy the topology to the cluster using the following Storm client command:

25. **bin/storm jar jarName.jar [TopologyMainClass] [Args]**

   The preceding command runs TopologyMainClass with the arguments, arg1 and arg2. The main function of TopologyMainClass is to define the topology and submit it to Nimbus. The Storm JAR part takes care of connecting to Nimbus and uploading the JAR part.

26. Go to the $STORM_HOME directory and run the following command to deploy LearningStormSingleNodeTopology to the Storm cluster:

27. **bin/storm jar $PROJECT_HOME/target/storm-example-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.learningstorm.storm_example.LearningStormSingleNodeTopology LearningStormSingleNodeTopology**

    The following information is displayed:

    0   [main] INFO   backtype.storm.StormSubmitter   - Jar not uploaded to master yet. Submitting jar...
    7   [main] INFO   backtype.storm.StormSubmitter   - Uploading topology jar /home/root/storm-example/target/storm-example-0.0.1-SNAPSHOT-jar-with-dependencies.jar to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-dfce742b-ca0b-4121-bcbe-1856dc1846a4.jar
    19   [main] INFO   backtype.storm.StormSubmitter   - Successfully uploaded topology jar to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-dfce742b-ca0b-4121-bcbe-1856dc1846a4.jar
    19   [main] INFO   backtype.storm.StormSubmitter   - Submitting topology LearningStormSingleNodeTopologyin distributed mode with conf{"topology.workers":3}
    219   [main] INFO   backtype.storm.StormSubmitter   - Finished submitting topology: LearningStormSingleNodeTopology

28. Run the jps command to see the number of running JVM processes as follows:

29. **jps**

    The preceding command's output is:

    26827 worker
    26530 supervisor
    26824 worker
    26468 nimbus
    15987 QuorumPeerMain
    26822 worker

30. Storm supports deactivating a topology. In the deactivated state, spouts will not emits any new tuples into pipeline, but the processing of already emitted tuples will continue. The following is the command to deactivate the running topology:

31. **bin/storm deactivate topologyName**

32. Deactivate LearningStormSingleNodeTopology using the following command:

33. **bin/storm deactivate LearningStormSingleNodeTopology**

The following information is displayed:

**0	[main] INFO	backtype.storm.thrift	- Connecting to Nimbus at localhost:6627r**
**76	[main] INFO	backtype.storm.command.deactivate	- Deactivated topology: LearningStormSingleNodeTopology**

34. Storm also supports activating a topology. When a topology is activated, spouts will again start emitting tuples. The following is the command to activate the topology:

35. **bin/storm activate topologyName**

36. Activate LearningStormSingleNodeTopology using the following command:

37. **bin/storm activate LearningStormSingleNodeTopology**

The following information is displayed:

**0	[main] INFO	backtype.storm.thrift	- Connecting to Nimbus at localhost:6627**
**65	[main] INFO	backtype.storm.command.activate	- Activated topology: LearningStormSingleNodeTopology**

38. Storm topologies are never-ending processes. To stop a topology, we need to kill it. When killed, the topology first enters into the deactivation state, processes all the tuples already emitted into it, and then stops. Run the following command to kill LearningStormSingleNodeTopology:

39. **bin/storm kill LearningStormSingleNodeTopology**

The following information is displayed:

**0	[main] INFO	backtype.storm.thrift	- Connecting to Nimbus at localhost:6627**
**80	[main] INFO	backtype.storm.command.kill-topology	- Killed topology: LearningStormSingleNodeTopology**

40. Now, run the jps command again to see the remaining JVM processes as follows:

41. **jps**

The preceding command's output is:

26530 supervisor
27193 Jps
26468 nimbus
15987 QuorumPeerMain

42. To update a running topology, the only option available is to kill the currently running topology and submit a new one.

Summary

In this chapter, we introduced you to the basics of Storm and the various components that make up a Storm cluster. We saw the different operation modes in which a Storm cluster can operate. We deployed a single-node Storm cluster and also developed a sample topology to run it on the single-node Storm cluster.

In the next chapter, we will set up a three-node Storm cluster to run the sample topology. We will also see different types of Stream groupings supported by Storm and the guaranteed message semantic provided by Storm.

Chapter 2. Setting Up a Storm Cluster

In the last chapter, we saw how to write a minimal Storm topology and run it on the local mode and a single-node Storm cluster. In this chapter, we will cover the following topics:

- How to run the sample topology in a distributed Storm cluster
- How to configure the parallelism of a topology
- How to partition a stream using different stream grouping

In the last chapter, we saw how to set up single-node ZooKeeper to use with Storm. Even though we can proceed with the same ZooKeeper setup for a distributed Storm cluster setup, then it will be a single point of failure in the cluster. To avoid this, we are deploying a distributed ZooKeeper cluster.

It is advised to run an odd number of ZooKeeper nodes, as the ZooKeeper cluster keeps working as long as the majority (the number of live nodes is greater than $n/2$, where n is the number of deployed nodes) of the nodes are running. So, if we have a cluster of four ZooKeeper nodes ($3 > 4/2$, only one node can die), then we can handle only one node failure, while if we had five nodes ($3 > 5/2$, two nodes can die) in the cluster, we can handle two node failures.

We will be deploying a ZooKeeper ensemble of three nodes that will handle one node failure. The following is the deployment diagram of the three-node ZooKeeper ensemble:
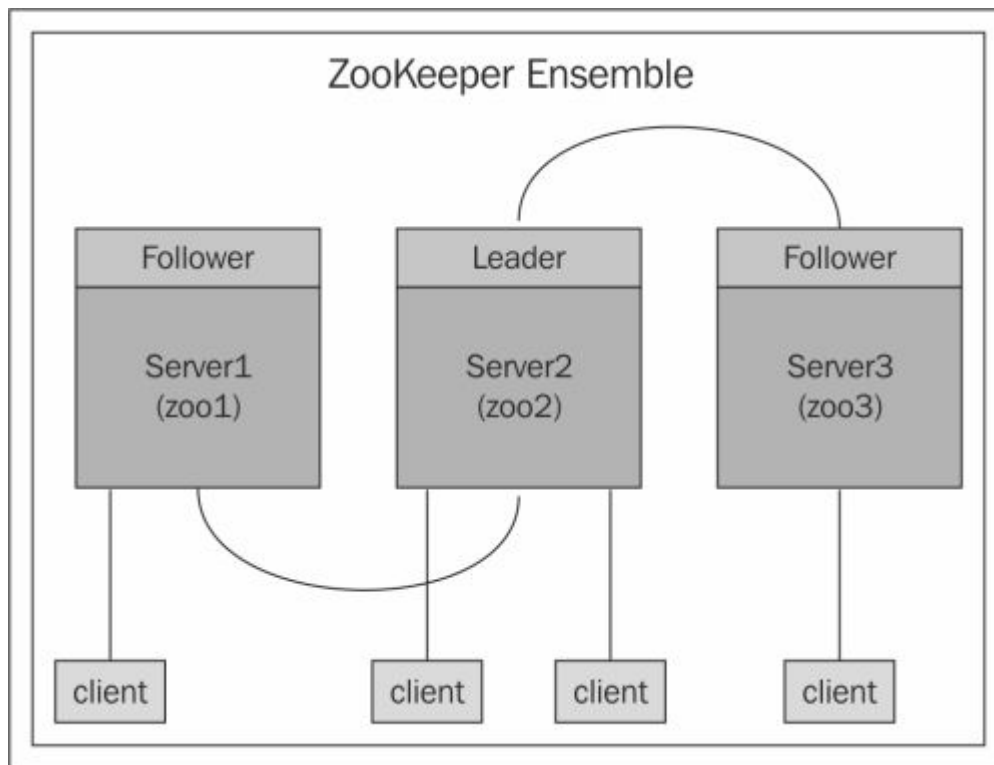
A ZooKeeper ensemble

In the ZooKeeper ensemble, one node in the cluster acts as the leader, while the rest are followers. If the leader node of the ZooKeeper cluster dies, then an election for the new leader takes places among the remaining live nodes, and a new leader is elected. All write requests coming from clients are forwarded to the leader node, while the follower nodes only handle the read requests. Also, we can't increase the write performance of the ZooKeeper ensemble by increasing the number of nodes because all write operations go through the leader node.

The following steps need to be performed on each node to deploy theZooKeeper ensemble:

1. Download the latest stable ZooKeeper release from the ZooKeeper site (http://zookeeper.apache.org/releases.html). At this moment, the latest version is ZooKeeper 3.4.5.
2. Once you have downloaded the latest version, unzip it. Now, we set up the ZK_HOME environment variable to make the setup easier.
3. Point the ZK_HOME environment variable to the unzipped directory. Create the configuration file, zoo.cfg, at $ZK_HOME/conf directory using the following commands:
4. **cd $ZK_HOME/conf**
5. **touch zoo.cfg**
6. Add the following properties to the zoo.cfg file:
7. **tickTime=2000**
8. **dataDir=/var/zookeeper**

9. **clientPort=2181**
10. **initLimit=5**
11. **syncLimit=2**
12. **server.1=z001:2888:3888**
13. **server.2=z002:2888:3888**
14. **server.3=z003.2888.3888**

Here, z001, z002, and z003 are the IP addresses of the ZooKeeper nodes. The following are the definitions for each of the properties:

- tickTime: This is the basic unit of time in milliseconds used by ZooKeeper. It is used to send heartbeats, and the minimum session timeout will be twice the tickTime value.
- dataDir: This is the directory to store the in-memory database snapshots and transactional log.
- clientPort: This is the port used to listen to client connections.
- initLimit: This is the number of tickTime values to allow followers to connect and sync to a leader node.
- syncLimit: This is the number of tickTime values that a follower can take to sync with the leader node. If the sync does not happen within this time, the follower will be dropped from the ensemble.

The last three lines of the server.id=host:port:port format specifies that there are three nodes in the ensemble. In an ensemble, each ZooKeeper node must have a unique ID between 1 and 255. This ID is defined by creating a file named myid in the dataDir directory of each node. For example, the node with the ID 1 (server.1=z001:2888:3888) will have a myid file at /var/zookeeper with the text 1 inside it.

For this cluster, create the myid file at three locations, shown as follows:

At   z001 /var/zookeeper/myid contains   1
At   z002 /var/zookeeper/myid contains   2
At   z003 /var/zookeeper/myid contains   3

2. Run the following command on each machine to start the ZooKeeper cluster:
3. **bin/zkServer.sh start**
4. Check the status of the ZooKeeper nodes by performing the following steps:

   0. Run the following command on the z001 node to check the first node's status:
   1. **bin/zkServer.sh status**

The following information is displayed:

JMX enabled by default
Using config: /home/root/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: follower

The first node is running in the follower mode.

2. Check the status of the second node by performing the following command:
3.   **bin/zkServer.sh status**

The following information is displayed:

JMX enabled by default
Using config: /home/root/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: leader

The second node is running in the leader mode.

4. Check the status of the third node by performing the following command:
5.   **bin/zkServer.sh status**

The following information is displayed:

JMX enabled by default
Using config: /home/root/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: follower

The third node is running in the follower mode.

2. Run the following command on the leader machine to stop the leader node:
3.   **bin/zkServer.sh stop**
4. Now, check the status of the remaining two nodes by performing the following steps:
   0. Check the status of the first node using the following command:
   1.   **bin/zkServer.sh status**

   The following information is displayed:

   JMX enabled by default
   Using config: /home/root/zookeeper-3.4.5/bin/../conf/zoo.cfg
   Mode: follower

The first node is again running in the follower mode.

2. Check the status of the third node using the following command:
3. **bin/zkServer.sh status**

The following information is displayed:

JMX enabled by default
Using config: /home/root/zookeeper-3.4.5/bin/../conf/zoo.cfg
Mode: leader

The third node is elected as the new leader.

4. Now, restart the third node with the following command:
5. **bin/zkServer.sh status**

This was a quick introduction to setting up ZooKeeper that can be used for development; however, it is not suitable for production. For a complete reference on ZooKeeper administration and maintenance, please refer to the online documentation at the ZooKeeper site at http://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html.

Setting up a distributed Storm cluster

In the last chapter, we saw how to set up a single-node Storm cluster. In this chapter, we will learn how to set up a three-node Storm cluster, of which one node will be the master node (Nimbus) and the other two will be worker nodes (supervisors).

The following is the deployment diagram of our three-node Storm cluster:
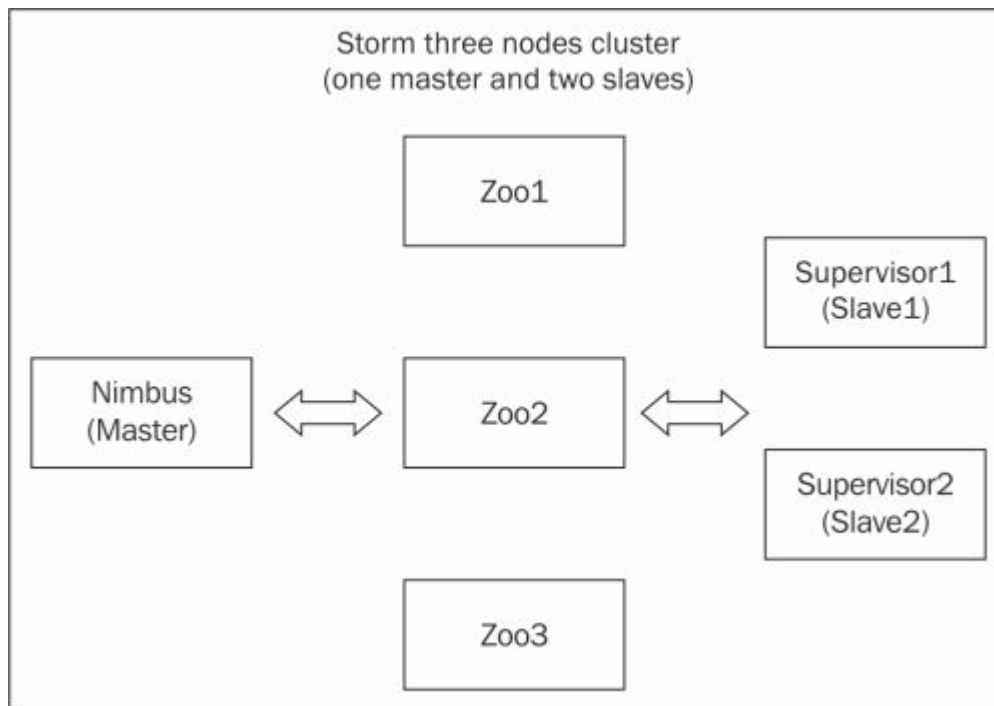
A three-node Storm cluster

The following are the steps that need to be performed to set up a three-node Storm cluster:

1. Install and run the ZooKeeper cluster. The steps for installing ZooKeeper are mentioned in the previous section.
2. Download the latest stable Storm release from https://storm.incubator.apache.org/downloads.html; at the time of this writing, the latest version is Storm 0.9.0.1.
3. Once you have downloaded the latest version, copy and unzip it in all three machines. Now, we will set the $STORM_HOME environment variable on each machine to make the setup easier.
4. Go to the $STORM_HOME/conf directory at the master node and add the following lines to the storm.yaml file:
5. **storm.zookeeper.servers:**
6. **- "zoo1"**
7. **- "zoo2"**
8. **- "zoo3"**
9. **storm.zookeeper.port: 2181**
10. **nimbus.host: "nimbus.host.ip"**
11. **storm.local.dir: "/tmp/storm-data"**
12. **java.library.path: "/usr/local/lib"**
13. **storm.messaging.transport: backtype.storm.messaging.netty.Context**

Here, zoo1, zoo2, and zoo3 are the IP addresses of the ZooKeeper machines, and nimbus.host.ip is the IP address of the master machine. The storm.local.dir path is a path to a local directory where Nimbus and supervisor store some local data such as state and topology JARs.

14. Go to the $STORM_HOME/conf directory at each worker node and add the following lines to the storm.yaml file:

15. **storm.zookeeper.servers:**

16. **- "zoo1"**

17. **- "zoo2"**

18. **- "zoo3"**

19. **storm.zookeeper.port: 2181**

20. **nimbus.host: "nimbus.host.ip"**

21. **storm.local.dir: "/tmp/storm-data"**

22. **java.library.path: "/usr/local/lib"**

23. **storm.messaging.transport: backtype.storm.messaging.netty.Context**

24. **supervisor.slots.ports:**

25. **- 6700**

26. **- 6701**

27. **- 6702**

28. **- 6703**

29. Go to the $STORM_HOME directory at the master node and execute the following command to start the master daemon:

30. **bin/storm nimbus**

31. Go to the $STORM_HOME directory at each worker node and execute the following command to start the worker daemons:

32. **bin/storm supervisor**

Deploying a topology on a remote Storm cluster

In this section, we will focus on how we can deploy topologies on a remote Storm cluster. We will start with the installation of a Storm client on the client machine, which can be different

from the machines in the Storm cluster because submitting and deploying topologies on a remote Storm cluster requires a Storm client.

The following are the steps that need to be performed to set up a Storm client:

1. Download the latest stable Storm release from https://storm.incubator.apache.org/downloads.html.
2. Once you have downloaded the latest version, copy and unzip it to the client machine. Now, we set the STORM_HOME environment variable to make the installation easier.
3. Go to the $STORM_HOME/conf directory at the client node and add the following line to the storm.yaml file:
4. **nimbus.host: "nimbus.host.ip"**
5. Also, now place the copy of the storm.yaml file located at $STORM_HOME/conf in the ~/.storm folder on the client machine.

Once the installation of the Storm client is done, we are good to go to deploy a topology on the remote machine. To demonstrate how we can deploy a topology on the remote cluster, we will use the sample topology developed in Chapter 1, *Setting Up Storm on a Single Machine*. The following are the commands that need to be executed to deploy a topology on the remote cluster.

Go to the $STORM_HOME directory on the client machine and run the following command:

bin/storm jar jarName.jar [TopologyMainClass] [Args]

Deploying the sample topology on the remote cluster

This section will explain how we can deploy the sample topology created in Chapter 1, *Setting Up Storm on a Single Machine*, on the Storm cluster by performing the following steps:

1. Execute the following command on the Storm client machine to deploy the sample topology on the remote Storm cluster. The client will then submit this topology across the network to the Nimbus, which will then distribute it to the supervisors.
2. **bin/storm jar $STORM_PROJECT_HOME/target/storm-example-0.0.1-SNAPSHOT-jar-with-dependencies.jar com.learningstorm.storm_example.LearningStormSingleNodeTopology LearningStormClusterTopology**

   The output of the preceding command is as follows:

18   [main] INFO   backtype.storm.StormSubmitter   - Uploading topology jar ../storm-example/target/storm-example-0.0.1-SNAPSHOT-jar-with-dependencies.jar to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-aa96e582-1676-4654-a995-15a4e88b6a50.jar

28   [main] INFO   backtype.storm.StormSubmitter   - Successfully uploaded topology jar to assigned location: /tmp/storm-data/nimbus/inbox/stormjar-aa96e582-1676-4654-a995-15a4e88b6a50.jar

29   [main] INFO   backtype.storm.StormSubmitter   - Submitting topology test-ack in distributed mode with conf {"topology.workers":3}

196   [main] INFO   backtype.storm.StormSubmitter   - Finished submitting topology: LearningStormClusterTopology

The preceding console output shows that the LearningStormClusterTopology topology is submitted on the remote cluster, and three worker processes are executed.

3. Run the jps commands on the supervisor machines to view the worker process:
    1. Run the jps command on the first supervisor machine:
    2. **jps**

        The preceding command's output is as follows:

        24347 worker
        ** 23940 supervisor**
        ** 24593 Jps**
        24349 worker

        Two worker processes are assigned to the first supervisor machine.

    3. Run the jps command on the second supervisor machine:
    4. **jps**

        The preceding command's output is as follows:

        24344 worker
        ** 23941 supervisor**
        ** 24543 Jps**

        One worker process is assigned to the second supervisor machine.

Configuring the parallelism of a topology

There are a number of components in a Storm topology. The throughput (processing speed) of the topology is decided by the number of instances of each component running in parallel. This is known as the parallelism of a topology. Let's first look at the processes or components responsible for the parallelism feature of the Storm cluster.

The worker process

A Storm topology is executed across multiple nodes in the Storm cluster. Each of the nodes in the cluster can run one or more JVMs called **worker processes** that are responsible for processing a part of the topology.

A Storm cluster can run multiple topologies at the same time. A worker process is bound to one of these topologies and can execute multiple components of that topology. If multiple topologies are run at the same time, none of them will share any of the workers, thus providing some degree of isolation between topologies.

The executor

Within each worker process, there can be multiple threads that execute parts of the topology. Each of these threads is called an **executor**. An executor can execute only one of the components, that is, any one spout or bolt in the topology.

Each executor, being a single thread, can only execute tasks assigned to it serially. The number of executors defined for a spout or bolt can be changed dynamically while the topology is running. This means that you can easily control the degree of parallelism for various components in your topology.

Tasks

A task is the most granular unit of task execution in Storm. Each task is an instance of a spout or bolt. While defining a Storm topology, you can specify the number of tasks for each spout and bolt. Once defined, the number of tasks cannot be changed for a component at runtime. Each task can be executed alone or with another task of the same type or another instance of the same spout or bolt.

The following diagram depicts the relationship between the worker process, executors, and tasks. Each of the blocks that contains tasks is an executor, for example, there are two executors for each component, and each component hosts a different number of tasks.

Also, as you can see in the following diagram, there are two executors and eight instances for **Task A**. The two executors are running in two different workers. If you are not getting enough performance out of this configuration, you can easily change the number of executors to four or eight for **Task A** to increase performance. The following diagram shows the relationship between various components of a topology:
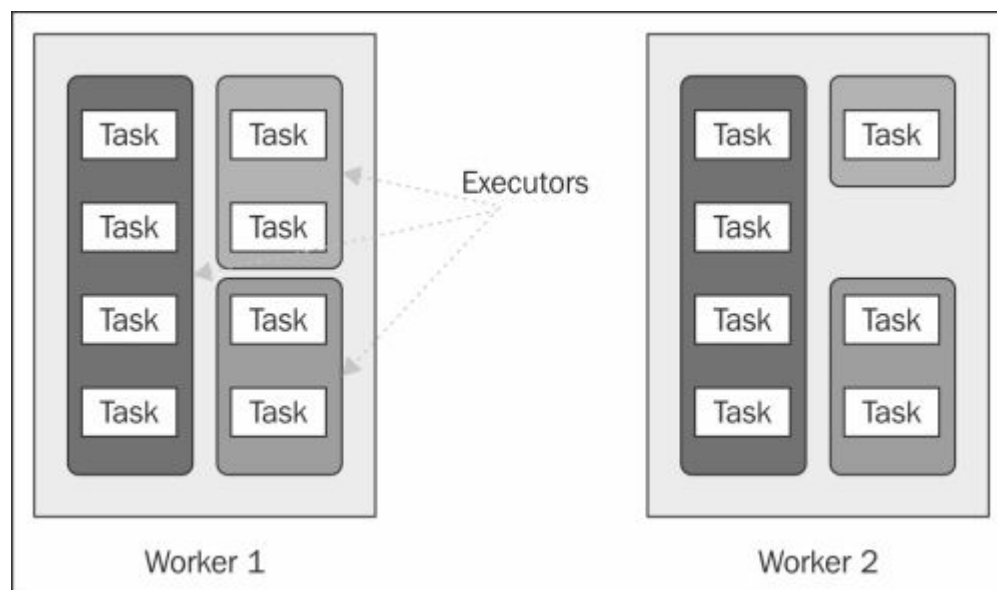
Relationship between executors, tasks, and worker processes

Configuring parallelism at the code level

In Storm, we can achieve the desired level of parallelism for tuning parameters such as the number of worker processes, number of executors, and number of tasks. Storm provides an API to configure these parameters. In this section, the following steps will show how we can configure parallelism at the code level:

1. Set the number of worker processes.

   We can set the number of worker processes at the code level using the setNumWorkers method of the backtype.storm.Config class. The following is the code snippet that shows these settings in practice:

   Config conf = new Config();
   conf.setNumWorkers(3);

In the preceding code, we have configured the number of workers to three. Storm will run the three workers for the LearningStormSingleNodeTopology topology.

2. Set the number of executors.

   We can set the number of executors at the code level by passing the parallelism_hint argument in the setSpout(args,args,parallelism_hint) or setBolt(args,args,parallelism_hint) method of the backtype.storm.topology.TopologyBuilder class. The following is the code snippet to show these settings in practice:

   ```
   TopologyBuilder builder = new TopologyBuilder();
   builder.setSpout("LearningStormSpout", new LearningStormSpout(), 2);
   builder.setBolt("LearningStormBolt", new LearningStormBolt(), 4);
   ```

   In the preceding code, we have set the parallelism_hint parameter to 2 for LearningStormSpout and 4 for LearningStormBolt. At the time of execution, Storm will assign two executors for LearningStormSpout and four executors for LearningStormBolt.

3. Set the number of tasks.

   We can configure the number of tasks that can execute inside the executors. The following is the code snippet to show these settings in practice:

   ```
   builder.setSpout("LearningStormSpout", new LearningStormSpout(),
   2).setNumTasks(4);
   ```

   In the preceding code, we have configured the two executors and four tasks of LearningStormSpout. For LearningStormSpout, Storm will assign two tasks per executor. By default, Storm will run one task per executor if the user does not set the number of tasks at the code level.

Distributing worker processes, executors, and tasks in the sample topology

Let's assume the number of worker processes set for the sample topology is three, the number of executors for LearningStormSpout is three, and the number of executors for LearningStormBolt is three. Also, we have configured the number of tasks for LearningStormBolt as six, which means each executor will run two tasks. Then, the following diagram shows how the sample topology would look in the operation:

Number of worker processes =3
Number of executors for spout =3
Number of tasks on each spout executor =1
Number of executor for bolt =3
Number of tasks on each bolt executor =2

Total parallelism = (Number of spout tasks)+
(Number of bolt tasks)
=>3+3*2=>9

Learning Spout → Learning Bolt

Hello-World Topology

Task Task Task
Task Task
Worker Process

Task Task Task
Task Task
Worker Process

Task Task Task
Task Task
Worker Process

The Hello-World topology distribution

The total parallelism of the topology can be calculated with the *total parallelism = number of spout tasks + number of bolt tasks* formula.

If the total parallelism of the topology is not a multiple of the number of workers, Storm will distribute the tasks as evenly as possible.

Rebalancing the parallelism of a topology

As explained in the previous section, one of the key features of Storm is that it allows us to modify the parallelism of a topology at runtime. The process of updating a topology parallelism at runtime is called **rebalance**. If we add new supervisor nodes to a Storm cluster and don't rebalance the topology, the new nodes will remain idle.

There are two ways to rebalance the topology:

- Using the Storm Web UI
- Using the Storm CLI

The Storm Web UI will be covered in detail in the next chapter. This section covers how we can rebalance the topology using the Storm CLI tool. The following is the command we need to execute on the Storm CLI tool to rebalance the topology:

bin/storm rebalance [TopologyName] -n [NumberOfWorkers] -e [Spout]=[NumberOfExecutos] -e [Bolt1]=[NumberOfExecutos] [Bolt2]=[NumberOfExecutos]

The rebalance command will first deactivate the topology for the duration of the message timeout and then redistribute the workers evenly around the Storm cluster. After a few seconds or minutes, the topology will be back in the previous state of activation and restart the processing of input streams.

Rebalancing the parallelism of the sample topology

With the following steps, let's first check the number of worker processes that are running in the Storm cluster by running jps commands on the supervisor machines:

1. Run the jps command on the first supervisor machine:
2. **jps**

   The following information is displayed:

   24347 worker
       23940 supervisor
       24593 Jps
   24349 worker

   Two worker processes are assigned to the first supervisor machine.

3. Run the jps command on the second supervisor machine:
4. **jps**

   The following information is displayed:

   24344 worker
       23941 supervisor
       24543 Jps

   One worker process is assigned to the second supervisor machine.

In total, three worker processes are running on the Storm cluster.

Let's try to reconfigure the LearningStormClusterTopology topology to use two worker processes, the LearningStormSpout spout to use four executors, and the LearningStormBolt bolt to use four executors using the following command:

bin/storm rebalance LearningStormClusterTopology -n 2 -e LearningStormSpout=4 -e LearningStormBolt=4

The following is the output displayed:

0[main] INFO   backtype.storm.thrift   - Connecting to Nimbus at nimbus.host.ip:6627
58   [main] INFO   backtype.storm.command.rebalance   - Topology LearningStormClusterTopology is rebalancing

Rerun the jps commands on the supervisor machines to view the number of worker processes as follows:

1. Run the jps command on the first supervisor machine:
2.   **jps**

    The following information is displayed:

    24377 worker
    **   23940 supervisor**
    **   24593 Jps**

    One worker process is assigned to the first supervisor machine.

3. Run the jps command on the second supervisor machine:
4.   **jps**

    The following information is displayed:

    24353 worker
    **   23941 supervisor**
    **   24543 Jps**

    One worker process is assigned to the second supervisor machine.

In total, two worker processes are running on the Storm cluster.

Stream grouping

When defining a topology, we create a graph of computation with a number of bolt-processing streams. At a more granular level, each bolt executes as multiple tasks in the topology. A stream will be partitioned into a number of partitions and divided among the bolts' tasks. Thus, each task of a particular bolt will only get a subset of the tuples from the subscribed streams.

Stream grouping in Storm provides complete control over how this partitioning of tuples happens among many tasks of a bolt subscribed to a stream. Grouping for a bolt can be defined on the instance of the backtype.storm.topology.InputDeclarer class returned when defining bolts using the backtype.storm.topology.TopologyBuilder.setBolt method.

Storm supports the following types of stream groupings:

- Shuffle grouping
- Fields grouping
- All grouping
- Global grouping
- Direct grouping
- Local or shuffle grouping
- Custom grouping

Now, we will look at each of these groupings in detail.

Shuffle grouping

Shuffle grouping distributes tuples in a uniform, random way across the tasks. An equal number of tuples will be processed by each task. This grouping is ideal when you want to distribute your processing load uniformly across the tasks and where there is no requirement of any data-driven partitioning.

Fields grouping

Fields grouping enables you to partition a stream on the basis of some of the fields in the tuples. For example, if you want that all the tweets from a particular user should go to a single task, then you can partition the tweet stream using fields grouping on the username field in the following manner:

builder.setSpout("1", new TweetSpout());

builder.setBolt("2", new TweetCounter()).fieldsGrouping("1", new Fields("username"))

Fields grouping is calculated with the following function:

hash (fields) % (no. of tasks)

Here, hash is a hashing function. It does not guarantee that each task will get tuples to process. For example, if you have applied fields grouping on a field, say *X*, with only two possible values, A and B, and created two tasks for the bolt, then it might be possible that both *hash (A) % 2* and *hash (B) % 2* are equal, which will result in all the tuples being routed to a single task and other tasks being completely idle.

Another common usage of fields grouping is to join streams. Since partitioning happens solely on the basis of field values and not the stream type, we can join two streams with any common join fields. The name of the fields do not need to be the same. For example, in order to process domains, we can join the Order and ItemScanned streams when an order is completed:

```
builder.setSpout("1", new OrderSpout());
builder.setSpout("2", new ItemScannedSpout());
builder.setBolt("joiner", new OrderJoiner())
.fieldsGrouping("1", new Fields("orderId"))
.fieldsGrouping("2", new Fields("orderRefId"));
```

All grouping

All grouping is a special grouping that does not partition the tuples but replicates them to all the tasks, that is, each tuple will be sent to each of the bolt's tasks for processing.

One common use case of all grouping is for sending signals to bolts. For example, if you are doing some kind of filtering on the streams, then you have to pass the filter parameters to all the bolts. This can be achieved by sending those parameters over a stream that is subscribed by all bolts' tasks with all grouping. Another example is to send a reset message to all the tasks in an aggregation bolt.

The following is an example of all grouping:

```
builder.setSpout("1", new TweetSpout());
builder.setSpout("signals", new SignalSpout());
builder.setBolt("2", new TweetCounter()).fieldsGrouping("1",
new Fields("username")).allGrouping("signals");
```

Here, we are subscribing signals for all the TweetCounter bolt's tasks. Now, we can send different signals to the TweetCounter bolt using SignalSpout.

Global grouping

Global grouping does not partition the stream but sends the complete stream to the bolt's task with the smallest ID. A general use case of this is when there needs to be a reduce phase in your topology where you want to combine results from previous steps in the topology in a single bolt.

Global grouping might seem redundant at first, as you can achieve the same results with defining the parallelism for the bolt as one and setting the number of input streams to one. Though, when you have multiple streams of data coming through different paths, you might want only one of the streams to be reduced and others to be processed in parallel.

For example, consider the following topology. In this topology, you might want to route all the tuples coming from **Bolt C** to a single **Bolt D** task, while you might still want parallelism for tuples coming from **Bolt E** to **Bolt D**.
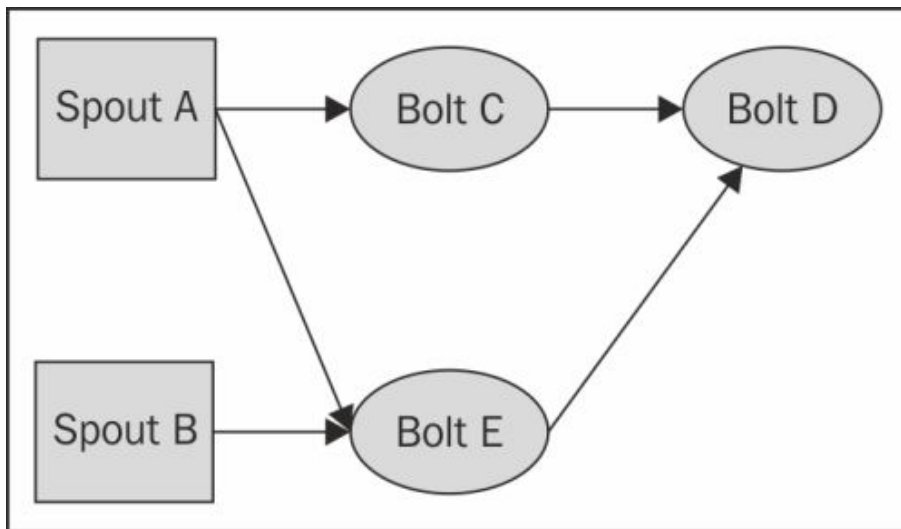
Global grouping

This can be achieved with the following code snippet:

```
builder.setSpout("a", new SpoutA());
builder.setSpout("b", new SpoutB());
builder.setBolt("c", new BoltC());
builder.setBolt("e", new BoltE());
builder.setBolt("d", new BoltD())
.globalGrouping("c")
```

.shuffleGrouping("e");

Direct grouping

In direct grouping, the emitter decides where each tuple will go for processing. For example, say we have a log stream and we want to process each log entry using a specific bolt task on the basis of the type of resource. In this case, we can use direct grouping.

Direct grouping can only be used with direct streams. To declare a stream as a direct stream, use the backtype.storm.topology.OutputFieldsDeclarer.declareStream method that takes a Boolean parameter directly in the following way in your spout:

```
@Override
public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declareStream("directStream", true, new Fields("field1"));
}
```

Now, we need the number of tasks for the component so that we can specify the taskId parameter while emitting the tuple. This can be done using the backtype.storm.task.TopologyContext.getComponentTasks method in the prepare method of the bolt. The following snippet stores the number of tasks in a bolt field:

```
public void prepare(Map stormConf, TopologyContext context, OutputCollector collector) {
    this.numOfTasks = context.getComponentTasks("my-stream");
    this.collector = collector;
}
```

Once you have a direct stream to emit to, use the backtype.storm.task.OutputCollector.emitDirect method instead of the emit method to emit it. The emitDirect method takes a taskId parameter to specify the task. In the following snippet, we are emitting to one of the tasks randomly:

```
public void execute(Tuple input) {
    collector.emitDirect(new Random().nextInt(this.numOfTasks), process(input));
}
```

Local or shuffle grouping

If the tuple source and target bolt tasks are running in the same worker, using this grouping will act as a shuffle grouping only between the target tasks running on the same worker, thus minimizing any network hops resulting in increased performance.

In case there are no target bolt tasks running on the source worker process, this grouping will act similar to the shuffle grouping mentioned earlier.

Custom grouping

If none of the preceding groupings fit your use case, you can define your own custom grouping by implementing the backtype.storm.grouping.CustomStreamGrouping interface.

The following is a sample custom grouping that partitions a stream on the basis of the category in the tuples:

```
public class CategoryGrouping implements CustomStreamGrouping, Serializable {
    // Mapping of category to integer values for grouping
    private static final Map<String, Integer> categories = ImmutableMap.of
    (
        "Financial", 0,
        "Medical", 1,
        "FMCG", 2,
        "Electronics", 3
    );

    // number of tasks, this is initialized in prepare method
    private int tasks = 0;

    public void prepare(WorkerTopologyContext context, GlobalStreamId stream, List<Integer> targetTasks)
    {
        // initialize the number of tasks
        tasks = targetTasks.size();
    }

    public List<Integer> chooseTasks(int taskId, List<Object> values) {
        // return the taskId for a given category
        String category = (String) values.get(0);
        return ImmutableList.of(categories.get(category) % tasks);
```

```
    }
}
```

Now, we can use this grouping in our topologies with the following code snippet:

```
builder.setSpout("a", new SpoutA());
builder.setBolt("b", (IRichBolt)new BoltB())
.customGrouping("a", new CategoryGrouping());
```

The following diagram represents the Storm groupings graphically:

Storm grouping
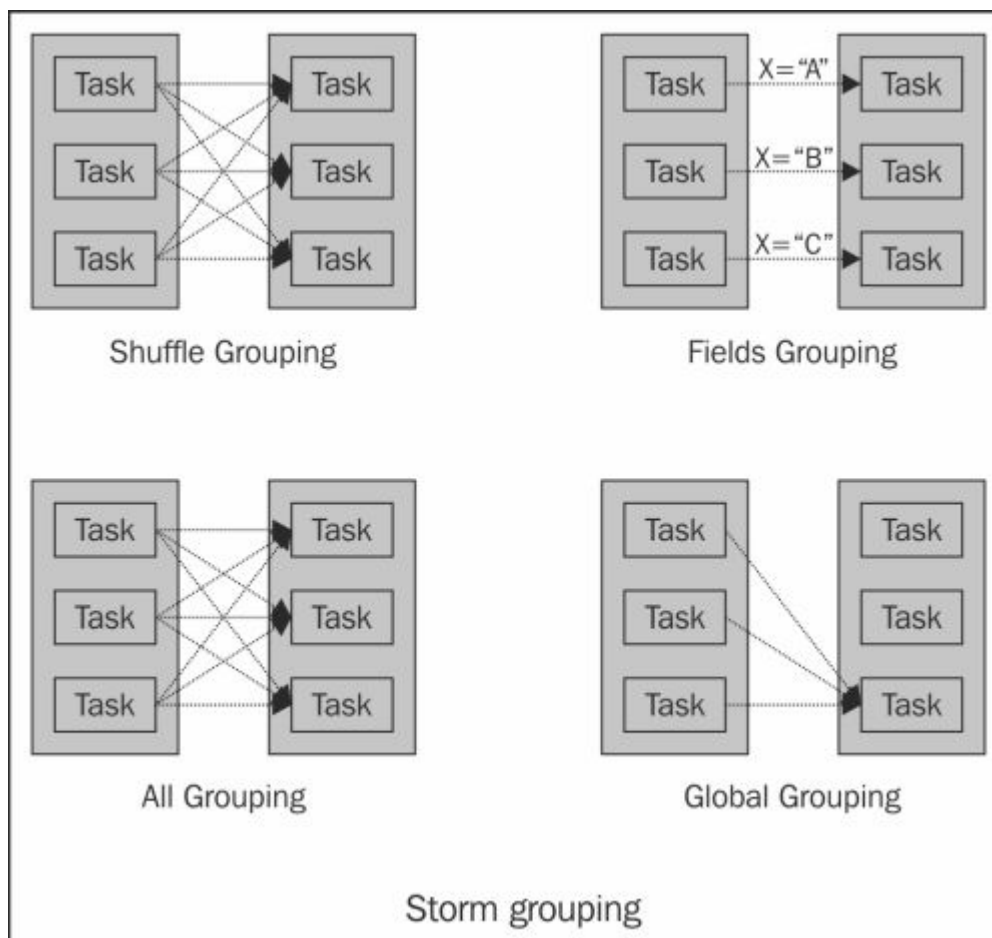
Guaranteed message processing

In a Storm topology, a single tuple being emitted by a spout can result in a number of tuples being generated in the later stages of the topology. For example, consider the following topology:

A Strom Topology

Here, **Spout A** emits a tuple **T(A)**, which is processed by **bolt B**, and **bolt C** which emits the tuples **T(AB)** and **T(AC)**, respectively. So, when all the tuples produced due to tuple **T(A)**—namely the tuple tree **T(A)**, **T(AB)**, and **T(AC)**—are processed, we say that the tuple has been processed completely.
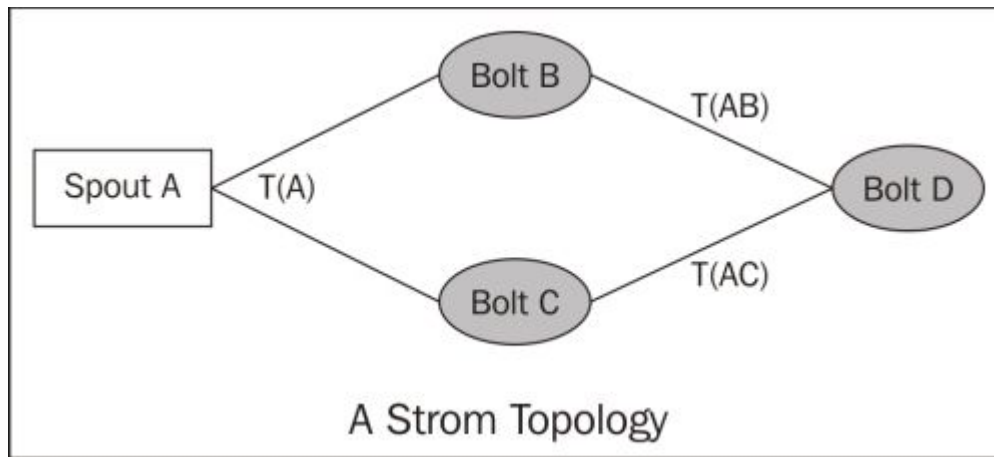
When some of the tuples in a tuple tree fail to process, either due to a runtime error or a timeout, which is configurable for each topology, then Storm considers this to be a failed tuple.

The following are the three steps that are required by Storm in order to guarantee message processing:

1. Tag each tuple emitted by a spout with a unique message ID. This can be done by using the backtype.storm.spout.SpoutOutputColletor.emit method that takes a messageId argument as follows:

   spoutOutputCollector.emit(Collections.singletonList((Object)tuple), generateMessageId(tuple));

   Storm uses this message ID to track the state of the tuple tree generated by this tuple. If you use one of the emit methods that don't take a messageId argument, Storm will not track it for complete processing. When the message is processed completely, Storm will send an acknowledgement with the same messageId argument that was used while emitting the tuple.

   A generic pattern implemented by spouts is that they read a message from a messaging queue, say RabbitMQ, produce the tuple into the topology for further processing, and then dequeue the message once it receives the acknowledgement that the tuple has been processed completely.

2. When one of the bolts in the topology needs to produce a new tuple in the course of processing a message, for example, **bolt B** in the preceding topology, then it should emit the new tuple anchored with the original tuple that it got from the spout. This can be done by using the overloaded emit methods in the backtype.storm.task.OutputCollector class that takes an anchor tuple as an argument. If you are emitting multiple tuples from the same input tuple, then anchor each outgoing tuple. The emit method is given in the following line of code:

   collector.emit(inputTuple, transform(inputTuple));

3. Whenever you are done with processing a tuple in the execute method of your bolt, send an acknowledgment using the backtype.storm.task.OutputCollector.ack method. When the acknowledgement reaches the emitting spout, you can safely mark the message as being processed and dequeue it from the message queue, if any.

   Similarly, if there is a problem in processing a tuple, a failure signal should be sent back using the backtype.storm.task.OutputCollector.fail method so that Storm can replay the failed message.

   One of the general patterns of processing in Storm bolts is to process a tuple in, emit new tuples, and send an acknowledgement at the end of the execute method. Storm provides the backtype.storm.topology.base.BaseBasicBolt class that automatically sends the acknowledgement at the end of the execute method. If you want to signal a failure, throw backtype.storm.topology.FailedException from the execute method. The following code snippet illustrates this:

```
public void execute(Tuple inputTuple, BasicOutputCollector collector) {
    try {
        collector.emit(transform(inputTuple));
        // successful completion will automatically act the tuple
    } catch(Exception e){
        // this will automatically fail the tuple
        throw new FailedException("Exception while processing tuple", e);
    }
}
```

The preceding model results in at-least-once message processing semantics, and your application should be ready to handle the scenario when some of the messages will be processed multiple times. Storm also provides exactly-once message processing semantics that we will discuss in Chapter 5, *Exploring High-level Abstraction in Storm with Trident*.

Even though you can achieve some guaranteed message processing in Storm using the preceding methods, it is always a point to ponder whether you actually require it or not as you can gain a lot of performance boost by risking some of the messages not being completely processed by Storm. This is a tradeoff that you can think of while designing your application.

Summary

In this chapter, we learned how to set up a distributed Storm cluster and how to set up the prerequisites such as ZooKeeper. We also learned how to deploy a topology on a Storm cluster and how to control the parallelism of a topology. Finally, we saw the various ways in which we can partition streams in Storm using various stream groupings provided by Storm. Now, you should be able to develop basic Storm topologies and deploy them.

In the next chapter, we will see how we can monitor the Storm cluster using the Storm UI and also how to collect topology statistics using the Nimbus thrift interface.

Chapter 3. Monitoring the Storm Cluster

In the previous chapter, you learned how we can deploy a sample topology on a remote Storm cluster, how we can configure the parallelism of a topology, different types of stream groupings, and so on. In this chapter, we will focus on how we can monitor and collect the diagnostics of topologies that run in a Storm cluster.

In this chapter, we will be covering the following topics:

- Start the Storm UI
- Monitoring a topology using the Storm UI
- Cluster statistics using the Nimbus thrift client

Starting to use the Storm UI

This section will show you how we can start the Storm UI daemon. However, before starting the Storm UI daemon, we assume that you have a running Storm cluster. The Storm cluster deployment steps are mentioned in [Chapter 2](), *Setting Up a Storm Cluster*. Now, go to the Storm home directory (cd $STORM_HOME) at the Nimbus machine and run the following command to start the Storm UI daemon:

bin/storm ui

By default, the Storm UI starts on the 8080 port of the machine where it is started. Now, we will browse to the http://nimbus-node:8080 page to view the Storm UI, where nimbus-node is the IP address or hostname of the Nimbus machine.

The following is a screenshot of the Storm home page:

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.0.1	11m 33s	2	0	8	8	0	0

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks

Supervisor summary

Id	Host	Uptime	Slots	Used slots
8aac36e9-5904-41a5-a9ea-af96cc1ef890	supervisor1	11m 14s	4	0
f70c269c-7b0b-4164-a8c3-f95bbbea4a8c	supervisor2	11m 30s	4	0

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx768m
drpc.invocations.port	3773
drpc.port	3772
drpc.queue.size	128

The home page of the Storm UI

Monitoring a topology using the Storm UI

This section covers how we can monitor the Storm cluster through the Storm UI. Let's first start with the definition of monitoring. **Monitoring** is used to track the health of various components that are running in a cluster. The statistics or information collected through monitoring is used by an administrator to spot an error or bottleneck in a cluster. The Storm UI daemon provides the following important information:

- **Cluster Summary**: This portion of the Storm UI shows the version of Storm deployed in a cluster, uptime of the nimbus node, number of free worker slots, number of used worker slots, and so on. While submitting a topology to the cluster, the user first needs to make sure that the value of the **Free slots** column should not be zero; otherwise, the topology doesn't get any worker for processing and will wait in the queue till a worker becomes free.
- **Nimbus Configuration**: This portion of the Storm UI shows the configuration of the Nimbus node.
- **Supervisor summary**: This portion of the Storm UI shows the list of supervisor nodes running in the cluster along with their **Id**, **Host**, **Uptime**, **Slots**, and **Used slots** columns.

- **Topology summary**: This portion of the Storm UI shows the list of topologies running in the Storm cluster along with their ID, number of workers assigned to the topology, number of executors, number of tasks, uptime, and so on.

Let's deploy the sample topology (if not running already) in a remote Storm cluster by running the following command:

bin/storm   **jar**
$STORM_PROJECT_HOME/target/storm-example-0.0.1-SNAPSHOT-jar-with-dep
endencies.jar
com.learningstorm.storm_example.LearningStormSingleNodeTopology
LearningStormClusterTopology

As mentioned in the *Configuring parallelism at the code level* section of Chapter 2, *Setting Up a Storm Cluster*, we created the LearningStormClusterTopology topology by defining three worker processes, two executors for LearningStormSpout, and four executors for LearningStormBolt.

After submitting LearningStormClusterTopology on the Storm cluster, the user has to refresh the Storm home page.

The following screenshot shows that the row is added for LearningStormClusterTopology in the **Topology summary** section. The topology section contains the name of the topology, unique ID of the topology, status of the topology, uptime, number of workers assigned to the topology, and so on. The possible values of status fields are **ACTIVE**, **KILLED**, and **INACTIVE**.

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.0.1	14m 15s	2	3	5	8	9	9

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
LearningStormClusterTopology	LearningStormClusterTopology-1-1404194614	ACTIVE	17s	3	9	9

Supervisor summary

Id	Host	Uptime	Slots	Used slots
8aac36e9-5904-41a5-a9ea-af96cc1ef890	supervisor1	13m 54s	4	1
f70c269c-7b0b-4164-a8c3-f95bbbea4a8c	supervisor2	14m 10s	4	2

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper
drpc.childopts	-Xmx768m
drpc.invocations.port	3773
drpc.port	3772

The home page of the Storm UI after deploying the sample topology

Let's click on **LearningStormClusterTopology** to view its detailed statistics. This shown in the following screenshot:

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
LearningStormClusterTopology	LearningStormClusterTopology-1-1404194614	ACTIVE	1m 36s	3	9	9

Topology actions

Activate | Deactivate | Rebalance | Kill

Topology stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	16250600	16250600	0.000	0	0
3h 0m 0s	16250600	16250600	0.000	0	0
1d 0h 0m 0s	16250600	16250600	0.000	0	0
All time	16250600	16250600	0.000	0	0

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
LearningStormSpout	2	2	16250600	16250600	0.000	0	0	

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Last error
LearningStormBolt	4	4	0	0	0.057	0.001	12178960	0.000	12178900	0	

The statistics of LearningStormClusterTopology

The preceding screenshot shows the statistics of the bolt and spout running in LearningStormClusterTopology. The screenshot contains the following major sections:

- **Topology actions**: This section allows us to activate, deactivate, rebalance, and kill the topology's functionality directly through the Storm UI.
- **Topology stats**: This section will give the information about the number of tuples emitted, transferred, and acknowledged, the capacity latency, and so on, within the window of 10 minutes, 3 hours, 1 day, and since the start of the topology.
- **Spouts (All time)**: This section shows the statistics of all the spouts running inside a topology. The following is the major information about a spout:

  - **Executors**: This column gives details about the number of executors assigned to LearningStormSpout. The value of the number of executors is two for LearningStormSpout because we have started LearningStormClusterTopology by assigning two executors for LearningStormSpout.
  - **Tasks**: This column gives details about the number of tasks assigned to LearningStormSpout. As explained in Chapter 2, *Setting Up a Storm Cluster*, the

tasks will run inside the executors, and if we don't specify the tasks, then Storm will automatically assign one task per executor. Hence, the number of tasks of LearningStormSpout is equal to the number of executors assigned to LearningStormSpout.

- **Emitted**: This column gives details about the number of records emitted all time by LearningStormSpout.
- **Port**: This column defines the worker port assigned to LearningStormSpout.
- **Transferred**: This column gives details about the number of records transferred all time by LearningStormSpout.
- **Complete latency (ms)**: This column gives the complete latency of a tuple. The complete latency is the difference in the timestamp when the spout emits the tuple to the timestamp when the ACK tree is completed for the tuple.

The difference between the emitted and transferred records is that the term emitted signifies the number of times the emit method of the OutputCollector class is called. On the other hand, the term transferred signifies the number of tuples actually sent to other tasks.

For example, the bolt Y has two tasks and subscribes to the bolt X using the all grouping type, then the value of emitted and transferred records is 2x for the bolt X. Similarly, if the bolt X emits the stream for which no one is subscribed to, then the value of transferred is zero.

- **Bolts (All time)**: This section shows the statistics of all the bolts running inside a topology. Here is some important information about a bolt:
  - **Executors**: This column gives details about the number of executors assigned to LearningStormBolt. The value of the number of executors is four for LearningStormBolt because we have started LearningStormClusterTopology by assigning four executors to LearningStormBolt.
  - **Tasks**: This column gives the details about the number of tasks assigned to LearningStormBolt. As explained in Chapter 2, *Setting Up a Storm Cluster*, the tasks will run inside the executors, and if we don't specify the tasks, then Storm will automatically assign one task per executor. Hence, the number of tasks of LearningStormBolt is equal to the number of executors assigned to LearningStormBolt.
  - **Emitted**: This column gives the details about the number of records emitted all time by LearningStormBolt.
  - **Port**: This column defines the worker port assigned to LearningStormBolt.

- **Transferred**: This column gives the details about the number of records transferred all time by LearningStormBolt.
- **Capacity (last 10m)**: The capacity metric is very important to monitor the performance of the bolt. This parameter gives an overview of the percent of the time spent by the bolt in actually processing tuples in the last 10 minutes. If the value of the **Capacity (last 10m)** column is close to 1, then the bolt is at capacity, and we will need to increase the parallelism of the bolt to avoid an "at capacity" situation. An "at capacity" situation is a bottleneck for the topology because if spouts start emitting tuples at a faster rate, then most of the tuples will timeout and spout will need to re-emit the tuples into the pipeline.
- **Process latency (ms)**: Process latency means the actual time (in milliseconds) taken by the bolt to process a tuple.
- **Execute latency (ms)**: Execute latency is the sum of the processing time and the time used in sending the acknowledgment.

Let's click on the LearningStormSpout link to view the detailed statistics of a spout, as shown in the following screenshot:

Component summary

Id	Topology	Executors	Tasks
LearningStormSpout	LearningStormClusterTopology	2	2

Spout stats

Window	Emitted	Transferred	Complete latency (ms)	Acked	Failed
10m 0s	29191880	29191880	0.000	0	0
3h 0m 0s	29191880	29191880	0.000	0	0
1d 0h 0m 0s	29191880	29191880	0.000	0	0
All time	29191880	29191880	0.000	0	0

Output stats (All time)

Stream	Emitted	Transferred	Complete latency (ms)	Acked	Failed
default	29191880	29191880	0	0	0

Executors (All time)

Id	Uptime	Host	Port	Emitted	Transferred	Complete latency (ms)	Acked	Failed
[5-5]	41s	supervisor1	6701	2611020	2611020	0.000	0	0
[6-6]	2m 49s	supervisor2	6702	26580860	26580860	0.000	0	0

The statistics of LearningStormSpout

The preceding screenshot shows that the tasks of LearningStormSpout are assigned to two executors. The screenshot also shows that the first executor is assigned to the supervisor1 machine and the second one is assigned to the supervisor2 machine.

Now, let's go to the previous page of the Storm UI and click on the LearningStormBolt link to view detailed statistics for the bolt, as shown in the following screenshot:

Bolt stats

Window	Emitted	Transferred	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed
10m 0s	0	0	0.001	35050720	0.000	35050720	0
3h 0m 0s	0	0	0.001	35050720	0.000	35050720	0
1d 0h 0m 0s	0	0	0.001	35050720	0.000	35050720	0
All time	0	0	0.001	35050720	0.000	35050720	0

Input stats (All time)

Component	Stream	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed
LearningStormSpout	default	0.001	35050720	0.000	35050720	0

Output stats (All time)

Stream	Emitted	Transferred

Executors

Id	Uptime	Host	Port	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed
[1-1]	3m 59s	supervisor2	6703	0	0	0.051	0.001	10185220	0.000	10185220	0
[2-2]	1m 54s	supervisor1	6701	0	0	0.057	0.001	4484260	0.000	4484260	0
[3-3]	3m 59s	supervisor2	6702	0	0	0.040	0.001	10196060	0.000	10196060	0
[4-4]	3m 59s	supervisor2	6703	0	0	0.052	0.001	10185180	0.000	10185180	0

The statistics of LearningStormBolt

The preceding screenshot shows that the tasks of LearningStormBolt are assigned to four executors. The screenshot also shows that the one executor is assigned to the supervisor1 machine and the remaining three executors are assigned to the supervisor2 machine. The **Input stats (All time)** section of the bolt shows the source of tuples for LearningStormBolt; in our case, the source is LearningStormSpout.

Again, go to the previous page and click on the **Kill** button to stop the topology. While killing the topology, Storm will first deactivate the spouts and wait for the kill time mentioned on the alerts box, so the bolts have a chance to finish the processing of the tuples emitted by spouts before the kill command. The following screenshot shows how we can kill the topology through the Storm UI:

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
LearningStormClusterTopology	LearningStormClusterTopology-1-1404194614	ACTIVE	6m 5s	3	9	9

Topology actions

[Activate] [Deactivate] [Rebalance] [Kill]

Topology stats

Window	Emitted	Transferred			Acked	Failed
10m 0s	60597960	60597980			0	0
3h 0m 0s	60597960	60597980			0	0
1d 0h 0m 0s	60597960	60597980			0	0
All time	60597960	60597980			0	0

The page at 192.168.41.122:8080 says: ✕

Do you really want to kill topology "LearningStormClusterTopology"? If yes, please, specify wait time in seconds:

`30`

[OK] [Cancel]

Spouts (All time)

Id	Executors	Tasks	Emitted	Transferred	Complete latency (ms)	Acked	Failed	Last error
LearningStormSpout	2	2	60597960	60597980	0.000	0	0	

Bolts (All time)

Id	Executors	Tasks	Emitted	Transferred	Capacity (last 10m)	Execute latency (ms)	Executed	Process latency (ms)	Acked	Failed	Last error
LearningStormBolt	4	4	0	0	0.053	0.001	54825620	0.000	54825580	0	

Killing a topology

Let's go to the Storm UI's home page to check the status of LearningStormClusterToplogy, as shown in the following screenshot:

Storm UI

Cluster Summary

Version	Nimbus uptime	Supervisors	Used slots	Free slots	Total slots	Executors	Tasks
0.9.0.1	20m 58s	2	3	5	8	9	9

Topology summary

Name	Id	Status	Uptime	Num workers	Num executors	Num tasks
LearningStormClusterTopology	LearningStormClusterTopology-1-1404194614	KILLED	7m 0s	3	9	9

Supervisor summary

Id	Host	Uptime	Slots	Used slots
8aac36e9-5904-41a5-a9ea-af96cc1ef890	supervisor1	20m 38s	4	1
f70c269c-7b0b-4164-a8c3-f95bbbea4a8c	supervisor2	20m 58s	4	2

Nimbus Configuration

Key	Value
dev.zookeeper.path	/tmp/dev-storm-zookeeper

The status of LearningStormClusterTopology

Cluster statistics using the Nimbus thrift client

Thrift is a binary protocol and is used for cross-language communication. The Nimbus node in Storm is a thrift service, and the topologies structure is also defined in the thrift structure. Due to the wide used of thrift in Storm, we can write code in any language to connect to the Nimbus node.

This section covers how we can collect the cluster details (similar to the details shown on the Storm UI page) using the Nimbus thrift client. The extraction or collection of information through the Nimbus thrift client allows us to plot or show the cluster details in a more visual manner.

The Nimbus thrift API is very rich and it exposes all the necessary information required to monitor the Storm cluster.

Fetching information with the Nimbus thrift client

We are going to look at how we can use the Nimbus thrift Java API to perform the following tasks:

- Collecting the Nimbus configuration
- Collecting the supervisor statistics
- Collecting the topology's statistics
- Collecting the spout's statistics for the given topology
- Collecting the bolt's statistics for the given topology
- Killing the given topology

The following are the steps to fetch the cluster details using the Nimbus thrift client:

1. Create a Maven project using com.learningstorm as **Group Id** and monitoring as **Artifact Id**, as shown in the following screenshot:

Create a new Maven project

2. Add the following dependencies in the pom.xml file:

3.   <dependency>

4.   <groupId>org.apache.thrift</groupId>

5.   <artifactId>libthrift</artifactId>

6.   <version>0.7.0</version>

7.   </dependency>

8.   <dependency>

9.   <groupId>storm</groupId>

10.   <artifactId>storm</artifactId>

11.   <version>0.9.0.1</version>

    </dependency>

12. Add the following repository in the pom.xml file:

13.   <repository>

14.   `<id>clojars.org</id>`

15.   `<url>http://clojars.org/repo</url>`

`</repository>`

16. Create a utility class, ThriftClient, in the com.learningstorm.monitoring package. The ThriftClient class contains logic to make a connection to the Nimbus thrift server and return the Nimbus client. The following is the code for the ThriftClient class:

17.   `public class ThriftClient {`

18.   `// IP of the Storm UI node`

19.   `private static final String STORM_UI_NODE = "127.0.0.1";`

20.   `public Client getClient() {`

21.   `// Set the IP and port of thrift server.`

22.   `// By default, the thrift server start on port 6627`

23.   `TSocket socket = new TSocket(STORM_UI_NODE, 6627);`

24.   `TFramedTransport tFramedTransport = new TFramedTransport(socket);`

25.   `TBinaryProtocol tBinaryProtocol = new TBinaryProtocol(tFramedTransport);`

26.   `Client client = new Client(tBinaryProtocol);`

27.   `try {`

28.   `// Open the connection with thrift client.`

29.   `tFramedTransport.open();`

30.   `}catch(Exception exception) {`

31.   `throw new RuntimeException("Error occurred while making connection with nimbus thrift server");`

32.   `}`

33.   `// return the Nimbus Thrift client.`

34.   `return client;`

35.   `}`

`}`

36. Let's create a NimbusConfiguration class in the com.learningstorm.monitoring package. This class contains logic to collect the Nimbus configuration using the Nimbus client. The following is the code for the NimbusConfiguration class:

37.   `public class NimbusConfiguration {`

38.

39.   `public void printNimbusStats() {`

40.   `try {`

41.   `ThriftClient thriftClient = new ThriftClient();`

42.   `Client client = thriftClient.getClient();`

43.   `String nimbusConiguration = client.getNimbusConf();`

```
44.          System.out.println("*********************************");
45.          System.out.println("Nimbus Configuration : "+nimbusConiguration);
46.          System.out.println("*********************************");
47.        }catch(Exception exception) {
48.          throw new RuntimeException("Error occurred while fetching the Nimbus
   statistics : ");
49.        }
50.     }
51.     public static void main(String[] args) {
52.        new NimbusConfiguration().printNimbusStats();
53.     }
   }
```

The preceding program uses the getNimbusConf() method of the
backtype.storm.generated.Nimbus.Client class to fetch the Nimbus configuration.

54. Create a SupervisorStatistics class in the com.learningstorm.monitoring package to
    collect information about all the supervisor nodes running in the Storm cluster. The
    following is the code for the SupervisorStatistics class:

```
55.   public class SupervisorStatistics {
56.
57.     public void printSupervisorStatistics(){
58.        try {
59.          ThriftClient thriftClient = new ThriftClient();
60.          Client client = thriftClient.getClient();
61.          // Get the cluster information.
62.          ClusterSummary clusterSummary = client.getClusterInfo();
63.          // Get the SupervisorSummary iterator
64.          Iterator<SupervisorSummary> supervisorsIterator =
   clusterSummary.get_supervisors_iterator();
65.
66.          while (supervisorsIterator.hasNext()) {
67.             // Print the information of supervisor node
68.             SupervisorSummary supervisorSummary = (SupervisorSummary)
   supervisorsIterator.next();
69.             System.out.println("*********************************");
70.             System.out.println("Supervisor Host IP : "+supervisorSummary.get_host());
71.             System.out.println("Number of used workers :
   "+supervisorSummary.get_num_used_workers());
```

```
72.          System.out.println("Number of workers :
   "+supervisorSummary.get_num_workers());
73.          System.out.println("Supervisor ID :
   "+supervisorSummary.get_supervisor_id());
74.          System.out.println("Supervisor uptime in seconds :
   "+supervisorSummary.get_uptime_secs());
75.          System.out.println("*************************************");
76.       }
77.
78.    }catch (Exception e) {
79.       throw new RuntimeException("Error occurred while getting cluster info : ");
80.    }
81.  }
82.
   }
```

The SupervisorStatistics class uses the getClusterInfo() method of the
backtype.storm.generated.Nimbus.Client class to get the instance of the
backtype.storm.generated.ClusterSummary class and then calls the
get_supervisors_iterator() method of the backtype.storm.generated.ClusterSummary
class to get an iterator over the backtype.storm.generated.SupervisorSummary class. The
following screenshot is the output of the SupervisorStatistics class:

```
*************************************
Supervisor Host IP : supervisor-1
Number of used workers : 1
Number of workers : 4
Supervisor ID : 872a45ce-5f58-466c-ba57-a29799991358
Supervisor uptime in seconds : 491
*************************************
*************************************
Supervisor Host IP : supervisor-2
Number of used workers : 2
Number of workers : 4
Supervisor ID : 5400bc2e-7e74-47af-a3b8-246705c4f1e7
Supervisor uptime in seconds : 475
*************************************
```

The output of the SupervisorStatistics class

83. Create a TopologyStatistics class in the com.learningstorm.monitoring package to collect
    information of all the topologies running in a Storm cluster, as shown in the following
    code:

```
84.  public class TopologyStatistics {
85.
86.     public void printTopologyStatistics() {
87.        try {
88.           ThriftClient thriftClient = new ThriftClient();
89.           // Get the thrift client
90.           Client client = thriftClient.getClient();
91.           // Get the cluster info
92.           ClusterSummary clusterSummary = client.getClusterInfo();
93.           // Get the interator over TopologySummary class
94.           Iterator<TopologySummary> topologiesIterator =
     clusterSummary.get_topologies_iterator();
95.           while (topologiesIterator.hasNext()) {
96.              TopologySummary topologySummary = topologiesIterator.next();
97.              System.out.println("*************************************");
98.              System.out.println("ID of topology: " + topologySummary.get_id());
99.              System.out.println("Name of topology: " + topologySummary.get_name());
100.             System.out.println("Number of Executors: " +
     topologySummary.get_num_executors());
101.             System.out.println("Number of Tasks: " +
     topologySummary.get_num_tasks());
102.             System.out.println("Number of Workers: " +
     topologySummary.get_num_workers());
103.             System.out.println("Status of topology: " + topologySummary.get_status());
104.             System.out.println("Topology uptime in seconds: " +
     topologySummary.get_uptime_secs());
105.             System.out.println("*************************************");
106.          }
107.       }catch (Exception exception) {
108.          throw new RuntimeException("Error occurred while fetching the topologies
     information");
109.       }
110.    }
     }
```

The TopologyStatistics class uses the get_topologies_iterator() method of the backtype.storm.generated.ClusterSummary class to get an iterator over the backtype.storm.generated.TopologySummary class. The class TopologyStatistics will print the value of the number of executors, the number of tasks, and the number of

worker processes assigned to each topology. The following is the console output of the TopologyStatistics class:

```
******************************************
ID of topology: LearningStormClusterTopology-1-1393847956
Name of topology: LearningStormClusterTopology
Number of Executors: 7
Number of Tasks: 7
Number of Workers: 3
Status of toplogy: ACTIVE
Topology uptime in seconds: 133
******************************************
```

The output of the TopologyStatistics class

111. Create a SpoutStatistics class in the com.learningstorm.monitoring package to get the statistics of spouts. The SpoutStatistics class contains a printSpoutStatistics(String topologyId) method to print the details about all the spouts served by the given topology, as shown in the following code:

112.　public class SpoutStatistics {

113.

114.　　private static final String DEFAULT = "default";

115.　　private static final String ALL_TIME = ":all-time";

116.

117.　　public void printSpoutStatistics(String topologyId) {

118.　　　try {

119.　　　　ThriftClient thriftClient = new ThriftClient();

120.　　　　// Get the nimbus thrift client

121.　　　　Client client = thriftClient.getClient();

122.　　　　// Get the information of given topology

123.　　　　TopologyInfo topologyInfo = client.getTopologyInfo(topologyId);

124.　　　　Iterator<ExecutorSummary> executorSummaryIterator = topologyInfo.get_executors_iterator();

125.　　　　while (executorSummaryIterator.hasNext()) {

126.　　　　　ExecutorSummary executorSummary = executorSummaryIterator.next();

127.　　　　　ExecutorStats executorStats = executorSummary.get_stats();

128.　　　　　if(executorStats !=null) {

129.　　　　　　ExecutorSpecificStats executorSpecificStats = executorStats.get_specific();

130.　　　　　　String componentId = executorSummary.get_component_id();

131.　　　　　　//

132.　　　　　　if (executorSpecificStats.is_set_spout()) {

```
133.            SpoutStats spoutStats = executorSpecificStats.get_spout();
134.            System.out.println("***********************************");
135.            System.out.println("Component ID of Spout:- " + componentId);
136.            System.out.println("Transferred:- " +
   getAllTimeStat(executorStats.get_transferred(), ALL_TIME));
137.            System.out.println("Total tuples emitted:- " +
   getAllTimeStat(executorStats.get_emitted(), ALL_TIME));
138.            System.out.println("Acked: " + getAllTimeStat(spoutStats.get_acked(),
   ALL_TIME));
139.            System.out.println("Failed: " + getAllTimeStat(spoutStats.get_failed(),
   ALL_TIME));
140.            System.out.println("***********************************");
141.          }
142.        }
143.      }
144.    }catch (Exception exception) {
145.      throw new RuntimeException("Error occurred while fetching the spout
   information : "+exception);
146.    }
147.  }
148.
149.  private static Long getAllTimeStat(Map<String, Map<String, Long>> map, String
   statName) {
150.    if (map != null) {
151.      Long statValue = null;
152.      Map<String, Long> tempMap = map.get(statName);
153.      statValue = tempMap.get(DEFAULT);
154.      return statValue;
155.    }
156.    return 0L;
157.  }
158.
159.  public static void main(String[] args) {
160.    new
   SpoutStatistics().printSpoutStatistics("LearningStormClusterTopology-1-1393847956");
161.  }
  }
```

The preceding class uses the getTopologyInfo(topologyId) method of the backtype.storm.generated.Nimbus.Client class to fetch the spout information of the given topology. The output of the TopologyStatistics class prints the ID of each topology; we can pass this ID as an argument to the getTopologyInfo(topologyId) method to get information about spouts running inside a topology. The SpoutStatistics class prints the following statistics of the spout:

- The spout ID
- The number of tuples emitted and transferred
- The number of tuples failed
- The number of tuples acknowledged

The following is the console output of the SpoutStatistics class:

```
******************************************
Component ID of Spout:- LearningStormSpout
Transferred:- 6584500
Total tuples emitted:- 6584500
Acked: null
Failed: null
******************************************
******************************************
Component ID of Spout:- LearningStormSpout
Transferred:- 60134980
Total tuples emitted:- 60134980
Acked: null
Failed: null
******************************************
```

The output of the SpoutStatistics class

2. Create a BoltStatistics class in the com.learningstorm.monitoring package to get the statistics of bolts. The BoltStatistics class contains a printBoltStatistics(String topologyId) method to print information about all the bolts served by the given topology, as shown in the following code:

3.   public class BoltStatistics {

4.   private static final String DEFAULT = "default";

5.   private static final String ALL_TIME = ":all-time";

6.

7.   public void printBoltStatistics(String topologyId) {

8.   try {

9.   ThriftClient thriftClient = new ThriftClient();

10.   // Get the Nimbus thrift server client

```
11.          Client client = thriftClient.getClient();
12.
13.          // Get the information of given topology
14.          TopologyInfo topologyInfo = client.getTopologyInfo(topologyId);
15.          Iterator<ExecutorSummary> executorSummaryIterator =
     topologyInfo.get_executors_iterator();
16.          while (executorSummaryIterator.hasNext()) {
17.            // get the executor
18.            ExecutorSummary executorSummary = executorSummaryIterator.next();
19.            ExecutorStats executorStats = executorSummary.get_stats();
20.            if (executorStats != null) {
21.              ExecutorSpecificStats executorSpecificStats = executorStats.get_specific();
22.              String componentId = executorSummary.get_component_id();
23.              if (executorSpecificStats.is_set_bolt()) {
24.                BoltStats boltStats = executorSpecificStats.get_bolt();
25.                System.out.println("*************************************");
26.                System.out.println("Component ID of Bolt " + componentId);
27.                System.out.println("Transferred: " +
     getAllTimeStat(executorStats.get_transferred(), ALL_TIME));
28.                System.out.println("Emitted:"
     +.getAllTimeStat(executorStats.get_emitted(), ALL_TIME));
29.                System.out.println("Acked: " + getBoltStats(boltStats.get_acked(),
     ALL_TIME));
30.                System.out.println("Failed: " + getBoltStats(boltStats.get_failed(),
     ALL_TIME));
31.                System.out.println("Executed: " +.getBoltStats(boltStats.get_executed(),
     ALL_TIME));
32.                System.out.println("*************************************");
33.              }
34.            }
35.          }
36.        } catch (Exception exception) {
37.          throw new RuntimeException("Error occurred while fetching the bolt
     information :"+exception);
38.        }
39.    }
40.
```

```
41.    private static Long getAllTimeStat(Map<String, Map<String, Long>> map, String
       statName) {
42.        if (map != null) {
43.            Long statValue = null;
44.            Map<String, Long> tempMap = map.get(statName);
45.            statValue = tempMap.get(DEFAULT);
46.            return statValue;
47.        }
48.        return 0L;
49.    }
50.
51.    public static Long getBoltStats(Map<String, Map<GlobalStreamId, Long>> map,
       String statName) {
52.        if (map != null) {
53.            Long statValue = null;
54.            Map<GlobalStreamId, Long> tempMap = map.get(statName);
55.            Set<GlobalStreamId> key = tempMap.keySet();
56.            if (key.size() > 0) {
57.                Iterator<GlobalStreamId> iterator = key.iterator();
58.                statValue = tempMap.get(iterator.next());
59.            }
60.            return statValue;
61.        }
62.        return 0L;
63.    }
64.
65.    public static void main(String[] args) {
66.        new
    BoltStatistics().printBoltStatistics("LearningStormClusterTopology-1-1393847956");
67.    }
    }
```

The preceding class uses the getTopologyInfo(topologyId) method of the backtype.storm.generated.Nimbus.Client class to fetch information about the given topology. The output of the TopologyStatistics class prints the ID of each topology; we can pass this ID as an argument to the getTopologyInfo(topologyId) method to get information about spouts running inside a topology. The BoltStatistics class prints the following statistics about a bolt:

- The bolt ID
- The number of tuples emitted and executed
- The number of tuples failed
- The number of tuples acknowledged

The following is the console output of the BoltStatistics class:

```
**********************************
Component ID of Bolt LearningStormBolt
Transferred: null
Emitted: null
Acked: 22000280
Failed: null
Executed : 22000320
**********************************
**********************************
Component ID of Bolt LearningStormBolt
Transferred: null
Emitted: null
Acked: 10872120
Failed: null
Executed : 10872140
**********************************
**********************************
Component ID of Bolt LearningStormBolt
Transferred: null
Emitted: null
Acked: 24950400
Failed: null
Executed : 24950400
**********************************
**********************************
Component ID of Bolt LearningStormBolt
Transferred: null
Emitted: null
Acked: 10874640
Failed: null
Executed : 10874640
```

The output of the BoltStatistics class

68. Create a killTopology class in the com.learningstorm.monitoring package to kill a topology. The following is the code for the killTopology class:

69. public class killTopology {

70.   public void kill(String topologyId) {

71.   try {

72.   ThriftClient thriftClient = new ThriftClient();

```
73.          // Get the nimbus thrift client
74.          Client client = thriftClient.getClient();
75.          // kill the given topology
76.          client.killTopology(topologyId);
77.
78.      }catch (Exception exception) {
79.          throw new RuntimeException("Error occurred while killing the topology :
    "+exception);
80.      }
81.    }
82.    public static void main(String[] args) {
83.      new killTopology().kill("topologyId");
84.    }
  }
```

The preceding class uses the killTopology(topologyName) method of the backtype.storm.generated.Nimbus.Client class to kill the topology.

In this section, we covered several examples that enable you to collect Storm cluster metrics or details using the Nimbus thrift client. The Nimbus thrift API is very rich and can collect all the metrics that are available on the Storm UI through this API.

Summary

In the first two chapters, we primarily focused on how to set up the local mode and the distributed mode of Storm cluster. You also learned how we can develop and deploy the topology on a Storm cluster.

In this chapter, we mainly concentrated on different ways of monitoring the Storm cluster. We began by starting the Storm UI and covered how we can monitor the topology using the Storm UI. We also walked through the Nimbus thrift client and covered sample examples that demonstrate how we can collect the Storm cluster's details using the Nimbus thrift client.

Chapter 4. Storm and Kafka Integration

Apache Kafka is a high-throughput, distributed, fault tolerant, and replicated messaging system that was first developed at LinkedIn. The use cases of Kafka vary from log aggregation to stream processing to replacing other messaging systems.

Kafka has emerged as one of the important components of real-time processing pipelines in combination with Storm. Kafka can act as a buffer or feeder for messages that need to be processed by Storm. Kafka can also be used as the output sink for results emitted from the Storm topologies.

In this chapter, we will cover the following topics:

- An overview of Apache Kafka and how it differs from traditional messaging platforms
- Setting up a single node and multinode Kafka cluster
- Producing data into a Kafka partition
- Using KafkaSpout in a Storm topology to consume messages from Kafka

The Kafka architecture

Kafka has an architecture that differs significantly from other messaging systems. Kafka is a peer-to-peer system in which each node is called a **broker**. The brokers coordinate their actions with the help of a ZooKeeper ensemble.
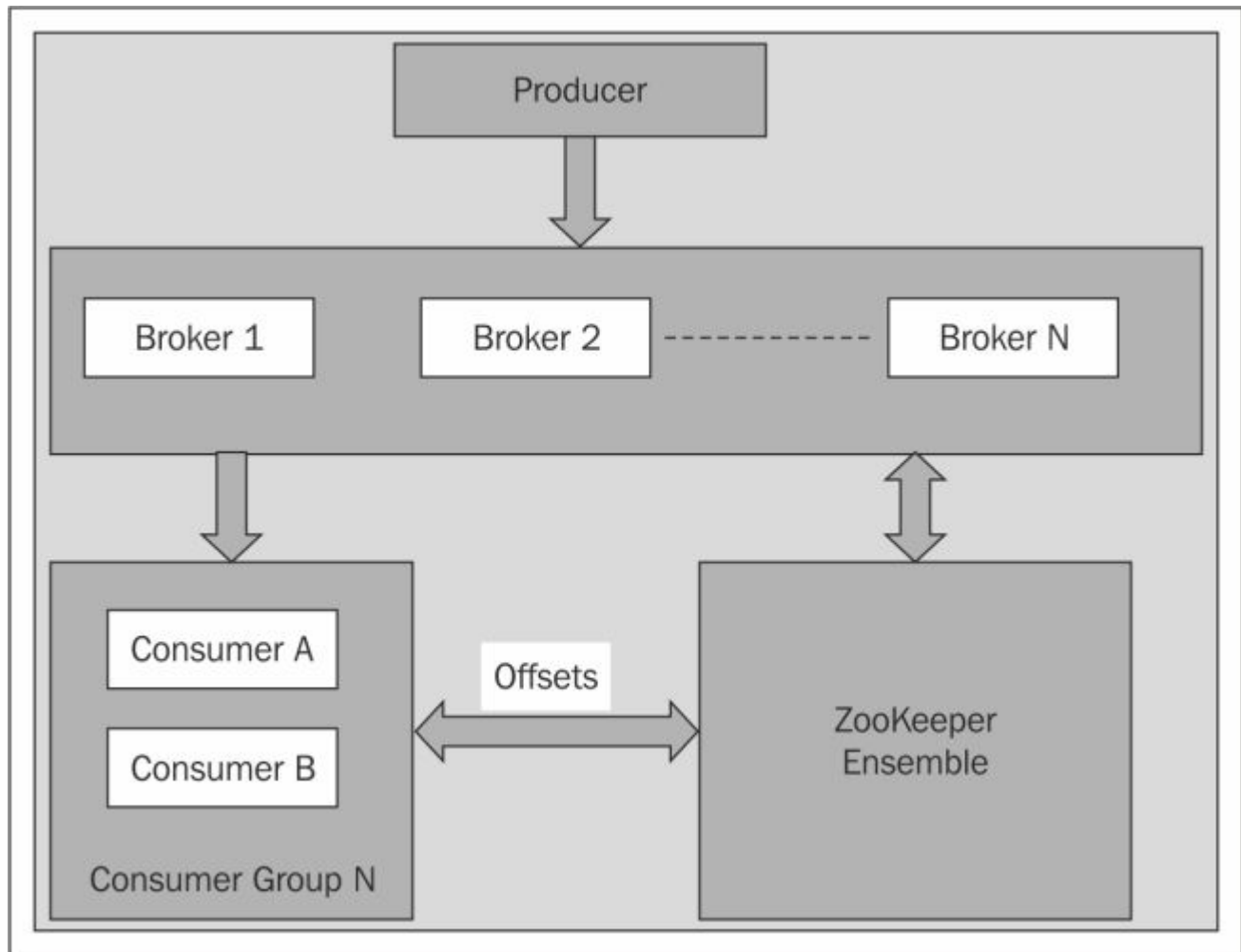
A Kafka cluster

The following are the important components of Kafka.

The producer

In Kafka, messages are published by a producer to named entities called **topics**. A topic is a queue that can be consumed by multiple consumers. For parallelism, a Kafka topic can have multiple partitions. Reads and writes can happen to each partition in parallel. Data for each partition of a topic is stored in a different directory on the disk. Each of these directories can be on different disks, allowing us to overcome the I/O limitations of a single disk. Also, two partitions of a single topic can be allocated on different brokers, thus increasing throughput as each partition is independent of each other. Each message in a partition has a unique sequence number associated with it called an **offset**.

Have a look at the following diagram showing the Kafka topic distribution:
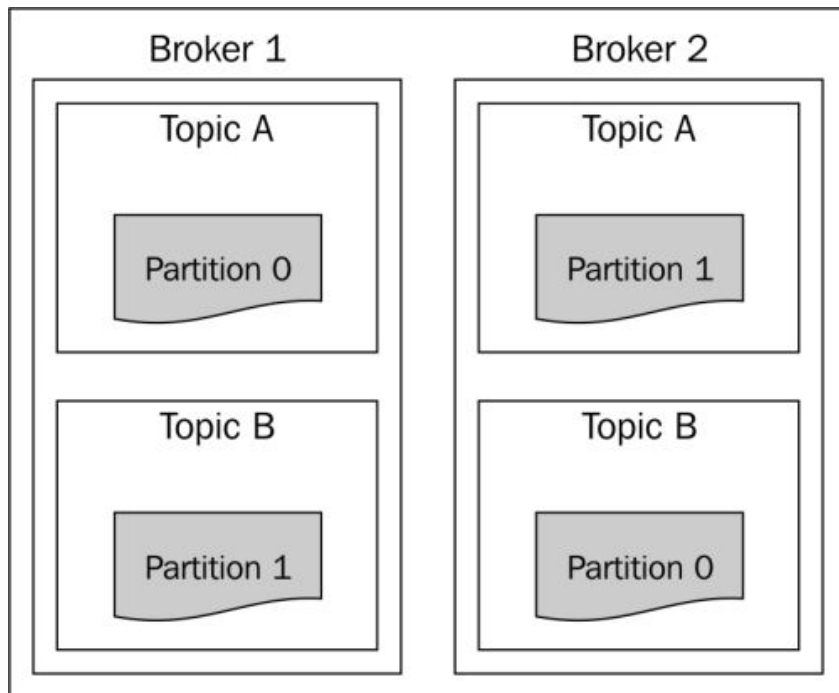
Kafka topics distribution

Replication

Kafka supports the replication of partitions of a topic to support fault tolerance. It automatically handles the replication of a partition and makes sure that the replica of the partition will be assigned to different brokers. Kafka elects one broker as the leader of a partition, and all the writes and reads must go to the leader partition. The replication feature was introduced in Kafka 0.8.0.

Consumers

A consumer reads a range of messages from a broker. A group ID is associated with each consumer. All the consumers with the same group ID act as a single logical consumer. Each message of the topic is delivered to one consumer from a consumer group (with the same group ID). Different consumer groups for a particular topic can process messages at their own pace as messages are not removed from the topics as soon as they are consumed. In fact, it is the responsibility of the consumers to keep track of how many messages they have consumed.

The following diagram depicts the relationship between consumers and consumer groups. We have a topic and two consumer groups with group ID 1 and 2. The consumer group 1 has two consumers, namely A and B, and each of them will consume from one of the partitions of the topic. Here, consumer A is consuming from partition p and consumer B is consuming from

partition q. For the consumer group 2, we only have a single consumer, X, that will consume the message from both the p and q partitions of the topic.
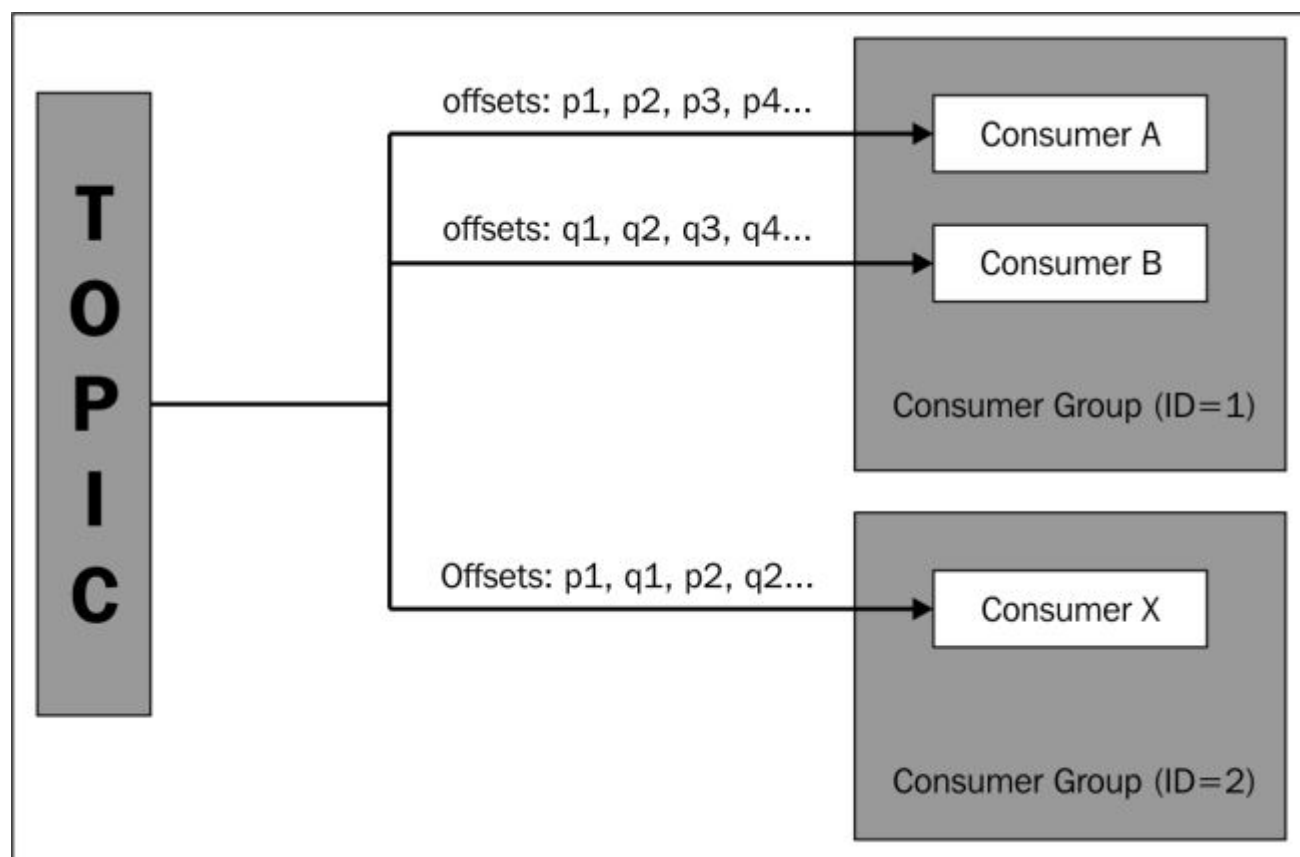
Kafka consumer groups

As mentioned earlier in this section, each message in a partition has a unique sequence number associated with it, called an offset. It is through this offset that consumers know how much of the stream they have already processed. If a consumer decides to replay already-processed messages, all it needs to do is just set the value of the offset to an earlier value while consuming messages from Kafka.

Brokers

A broker receives the messages from a producer (push mechanism) and delivers the messages to a consumer (pull mechanism). A broker also manages the persistence of messages on the disk. For each topic, it will create a directory on the disk. This directory will contain multiple files. The Kafka broker is very lightweight; it only opens the file handlers for partitions to persist messages and manage the TCP connections.

Data retention

Each topic in Kafka has an associated retention time that can be controlled with the log.retention.minutes property in the broker configuration. When this time expires, Kafka deletes the expired data files for that particular topic. This is a very efficient operation as it's a file delete operation.

Another way of controlling retention is through the log.retention.bytes property. It tells Kafka to delete expired data files when a certain size is reached for a partition. If both the properties are configured, the deletion will happen when any of the limits are reached.

Setting up Kafka

At the time of this writing, the stable version of Kafka is 0.8.1. The prerequisites for running Kafka is a ZooKeeper ensemble and Java Version 1.6 or above. Kafka comes with a convenience script that can start a single-node ZooKeeper, but it is not recommended to use it in a production environment. We will be using the ZooKeeper cluster we deployed in the *Setting up a ZooKeeper cluster* section of [Chapter 2](#), *Setting Up a Storm Cluster*.

We will see both how to set up a single-node Kafka cluster first and how to add two more nodes to it to run a full-fledged three-node Kafka cluster with replication enabled.

Setting up a single-node Kafka cluster

The following are the steps to set up a single-node Kafka cluster:

1. Download the Kafka 0.8.1.1 binary distribution named kafka_2.8.0-0.8.1.1.tgz from http://kafka.apache.org/downloads.html.
2. Extract the archive to where you want to install Kafka with the following command:
3. **tar -xvzf kafka_2.8.0-0.8.1.1.tgz**
4. **cd kafka_2.8.0-0.8.1.1**

   We will refer to the Kafka installation directory as $KAFKA_HOME from now onwards.

5. Change the following properties in the Kafka server properties file, server.properties, placed at $KAFKA_HOME/config:
6. log.dirs=/var/kafka-logs
   zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181

   Here, zoo1, zoo2, and zoo3 represent the hostnames of the ZooKeeper nodes. The following are the definitions of the important properties in the server.properties file:

- broker.id: This is a unique integer ID for each broker in a Kafka cluster.
- port: This is the port number for a Kafka broker. Its default value is 9092. If you want to run multiple brokers on a single machine, give a unique port to each broker.
- host.name: This is the hostname to which the broker should bind and advertise itself.
- log.dirs: The name of this property is a bit unfortunate as it represents not the log directory for Kafka, but the directory where Kafka stores its actual data. This can take a single directory or a comma-separated list of directories to store data. Kafka throughput can be increased by attaching multiple physical disks to the broker node and specifying multiple data directories, each lying on a different disk. It is not of much use to specify multiple directories on the same physical disk as all the I/O will still be happening on the same disk.
- num.partitions: This represents the default number of partitions for newly created topics. This property can be overridden when creating new topics. A greater number of partitions results in greater parallelism at the cost of a larger number of files. By default, this value is set to 1.
- log.retention.hours: Kafka does not delete messages immediately after consumers consume them. It retains them for the number of hours defined by this property so that in case of any issues, the consumers can replay the messages from Kafka. The default value is one week. Alternatively, you can also use the log.retention.minutes property to specify the retention policy in minutes or the log.retention.bytes property to specify the retention policy in terms of topic size.
- zookeeper.connect: This is the comma-separated list of ZooKeeper nodes in the hostname:port form.

2. Start the Kafka server by running the following command:

3. **./bin/kafka-server-start.sh config/server.properties**

The following information is displayed:

**[2014-06-28 09:40:21,954] INFO Verifying properties
(kafka.utils.VerifiableProperties)
[2014-06-28 09:40:22,094] INFO Property broker.id is overridden to 0
(kafka.utils.VerifiableProperties)
[2014-06-28 09:40:24,190] INFO [Kafka Server 0], started
(kafka.server.KafkaServer)
[2014-06-28 09:40:24,307] INFO New leader is 0
(kafka.server.ZookeeperLeaderElector$LeaderChangeListener)**

If you get something similar to the preceding lines on your console, then your Kafka broker is up and running and we can proceed to test it.

4. Now, we will verify that the Kafka broker has been set up correctly by sending and receiving a test message.

First, let's create a verification topic for testing by executing the following command:

./bin/kafka-topics.sh --create --zookeeper zoo1:2181 --partitions 1 --replication-factor 1 --topic verification-topic

We will receive the following output:

creation succeeded!

Now, let's verify that the topic creation was successful by listing all the topics:

./bin/kafka-topics.sh --zookeeper zoo1:2181 --list

We will receive the following output:

verification-topic

Now that the topic is created, let's produce sample messages to Kafka. Kafka comes with a command-line producer that we can use to produce messages as follows:

./bin/kafka-console-producer.sh --broker-list localhost:9092 --topic verification-topic

Write the following messages on the console:

Message 1
Test Message 2
Message 3

Let's consume these messages by starting a console consumer on a new console window and use the following command:

./bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic verification-topic --from-beginning

The following output is displayed on the console:

Message 1
Test Message 2
Message 3

Now, as you type any message on the producer console, it will automatically be consumed by this consumer and displayed on the command line.

Note

Using Kafka's single-node ZooKeeper instance

If you don't want to use an external ZooKeeper ensemble, you can use the single-node ZooKeeper instance that comes with Kafka for quick and dirty development. To start using it, first modify the zookeeper.properties file at $KAFKA_HOME/config to specify the data directory by supplying the following property:

dataDir=/var/zookeeper

Now, you can start the ZooKeeper instance with the following command:

./bin/zookeeper-server-start.sh config/zookeeper.properties

Setting up a three-node Kafka cluster

Now that we have a single-node Kafka cluster, let's see how we can set up a multinode Kafka cluster using the following steps:

1. Download and unzip Kafka on the three nodes, following steps 1 and 2 of the previous section.
2. Change the following properties in the Kafka server properties file, server.properties, at $KAFKA_HOME/config:
3. broker.id=0
4. port=9092
5. host.name=kafka1
6. log.dirs=/var/kafka-logs
   zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181

   Make sure that the value of the broker.id property is unique for each Kafka broker.

7. Start the Kafka brokers on the nodes by executing the following command on the three nodes:
8. **./bin/kafka-server-start.sh config/server.properties**
9. Now, let's verify the setup. First, we create a topic with the following command:
10. **./bin/kafka-topics.sh --create --zookeeper zoo1:2181 --partitions 3 --replication-factor 1 --topic verification**

We will receive the following output:

creation succeeded!

Now, we will list the topics to see whether the topic was created successfully using the following command:

./bin/kafka-topics.sh --describe --zookeeper zoo1:2181 --topic verification

The following information is displayed:

Topic:verification PartitionCount:3   ReplicationFactor:1   Configs:
Topic: verification Partition: 0   Leader: 0   Replicas: 0   Isr: 0
Topic: verification Partition: 1   Leader: 1   Replicas: 0   Isr: 0
Topic: verification Partition: 2   Leader: 2   Replicas: 0   Isr: 0

Now, we will verify the setup by using the Kafka console producer and consumer as done in the previous section using the following command:

./bin/kafka-console-producer.sh   --broker-list kafka1:9092,kafka2:9092,kafka3:9092 --topic verification

Here, kafka1, kafka2, and kafka3 are the IP addresses of Kafka brokers. Write the following messages on the console:

First
Second
Third

Let's consume these messages by starting a new console consumer on a new console window as follows:

./bin/kafka-console-consumer.sh --zookeeper zoo1:2181 --topic verification --from-beginning

We will receive the following output:

First
Second
Third

So now, we have a working three-broker Kafka cluster. In the next section, we will see how to write a producer that can produce messages to Kafka.

Running multiple Kafka brokers on a single node

If you don't have multiple machines and you want to test how partitions are distributed among various brokers, then you can run multiple Kafka brokers on a single node. The following are the steps to set up multiple Kafka brokers on a single node:

1. Copy the server.properties file from the config folder to create the server1.properties and server2.properties files in the config folder.
2. Populate the following properties in the server.properties file:
3.   broker.id=0
4.   port=9092
5.   log.dirs=/var/kafka-logs
     zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181

6. Populate the following properties in the server1.properties file:
7.   broker.id=1
8.   port=9093
9.   log.dirs=/var/kafka-1-logs
     zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181

10. Populate the following properties in the server2.properties file:
11.   broker.id=2
12.   port=9094
13.   log.dirs=/var/kafka-2-logs
      zookeeper.connect=zoo1:2181,zoo2:2181,zoo3:2181

14. Run the following commands on the three different terminals to start Kafka brokers:
15.   **./bin/kafka-server-start.sh config/server.properties**
16.   **./bin/kafka-server-start.sh config/server1.properties**
17.   **./bin/kafka-server-start.sh config/server2.properties**

A sample Kafka producer

In this section, we will learn how to write a producer that will publish events into the Kafka messaging queue. In the next section, we will process the events published in this section with a

Storm topology that reads data from Kafka using KafkaSpout. Perform the following steps to create the producer:

1. Create a new Maven project with the com.learningstorm group ID and the kafka-producer artifact ID.
2. Add the following dependencies for Kafka in the pom.xml file:
3.   `<dependency>`
4.   `<groupId>org.apache.kafka</groupId>`
5.   `<artifactId>kafka_2.8.0</artifactId>`
6.   `<version>0.8.1.1</version>`
7.   `<exclusions>`
8.   `<exclusion>`
9.   `<groupId>javax.jms</groupId>`
10.   `<artifactId>jms</artifactId>`
11.   `</exclusion>`
12.   `<exclusion>`
13.   `<groupId>com.sun.jdmk</groupId>`
14.   `<artifactId>jmxtools</artifactId>`
15.   `</exclusion>`
16.   `<exclusion>`
17.   `<groupId>com.sun.jmx</groupId>`
18.   `<artifactId>jmxri</artifactId>`
19.   `</exclusion>`
20.   `</exclusions>`
     `</dependency>`

21. Add the following build plugins to the pom.xml file; it will execute the producer using Maven:
22.   `<build>`
23.   `<plugins>`
24.   `<plugin>`
25.   `<groupId>org.codehaus.mojo</groupId>`
26.   `<artifactId>exec-maven-plugin</artifactId>`
27.   `<version>1.2.1</version>`
28.   `<executions>`
29.   `<execution>`
30.   `<goals>`
31.   `<goal>exec</goal>`
32.   `</goals>`

```
33.            </execution>
34.         </executions>
35.         <configuration>
36.            <executable>java</executable>
37.            <includeProjectDependencies>true</includeProjectDependencies>
38.            <includePluginDependencies>false</includePluginDependencies>
39.            <classpathScope>compile</classpathScope>
40.            <mainClass>com.learningstorm.kafka.WordsProducer</mainClass>
41.         </configuration>
42.      </plugin>
43.   </plugins>
   </build>
```

44. Now, we will create the WordsProducer class in the com.learningstorm.kafka package. This class will produce each word from the first paragraph of Franz Kafka's Metamorphosis into the words_topic topic in Kafka as a single message. The following is the code of the WordsProducer class with explanation:

```
45.  public class WordsProducer {
46.     public static void main(String[] args) {
47.        // Build the configuration required for connecting to Kafka
48.        Properties props = new Properties();
49.
50.        //List of Kafka brokers. Complete list of brokers is not
51.        //required as the producer will auto discover the rest of
52.        //the brokers. Change this to suit your deployment.
53.        props.put("metadata.broker.list", "localhost:9092");
54.
55.        // Serializer used for sending data to kafka. Since we are sending string,
56.        // we are using StringEncoder.
57.        props.put("serializer.class", "kafka.serializer.StringEncoder");
58.
59.        // We want acks from Kafka that messages are properly received.
60.        props.put("request.required.acks", "1");
61.
62.        // Create the producer instance
63.        ProducerConfig config = new ProducerConfig(props);
64.        Producer<String, String> producer = new Producer<String, String>(config);
65.
66.        // Now we break each word from the paragraph
```

```
67.        for (String word : METAMORPHOSIS_OPENING_PARA.split("\\s")) {
68.            // Create message to be sent to "words_topic" topic with the word
69.            KeyedMessage<String,    String>    data    =new    KeyedMessage<String,
   String>("words_topic", word);
70.
71.            // Send the message
72.            producer.send(data);
73.        }
74.
75.        System.out.println("Produced data");
76.
77.        // close the producer
78.        producer.close();
79.    }
80.
81.    // First paragraph from Franz Kafka's Metamorphosis
82.    private static String METAMORPHOSIS_OPENING_PARA = "One morning, when
   Gregor Samsa woke from troubled dreams, " + "he found himself transformed in his bed
   into a horrible " + "vermin. He lay on his armour-like back, and if he lifted " + "his head a
   little he could see his brown belly, slightly " + "domed and divided by arches into stiff
   sections.";
   }
```

83. Now, we can run the producer by executing the following command:

84. **mvn compile exec:java**

The following output is displayed:

Produced data

85. Now, let's verify that the message has been produced using Kafka's console consumer by
   executing the following command:

86. **bin/kafka-console-consumer.sh   --zookeeper   localhost:2181   --topic
   words_topic --from-beginning**

The following output is displayed:

One
morning,
when

Gregor

Samsa

woke

from

troubled

dreams,

he

found

himself

transformed

in

his

bed

into

a

horrible

vermin.

So, we are able to produce messages into Kafka. In the next section, we will see how we can use KafkaSpout to read messages from Kafka and process them inside a Storm topology.

Integrating Kafka with Storm

Now, we will create a Storm topology that will consume messages from a Kafka topic, word_topic, and aggregate words into sentences.

The complete message flow is shown in the following diagram:
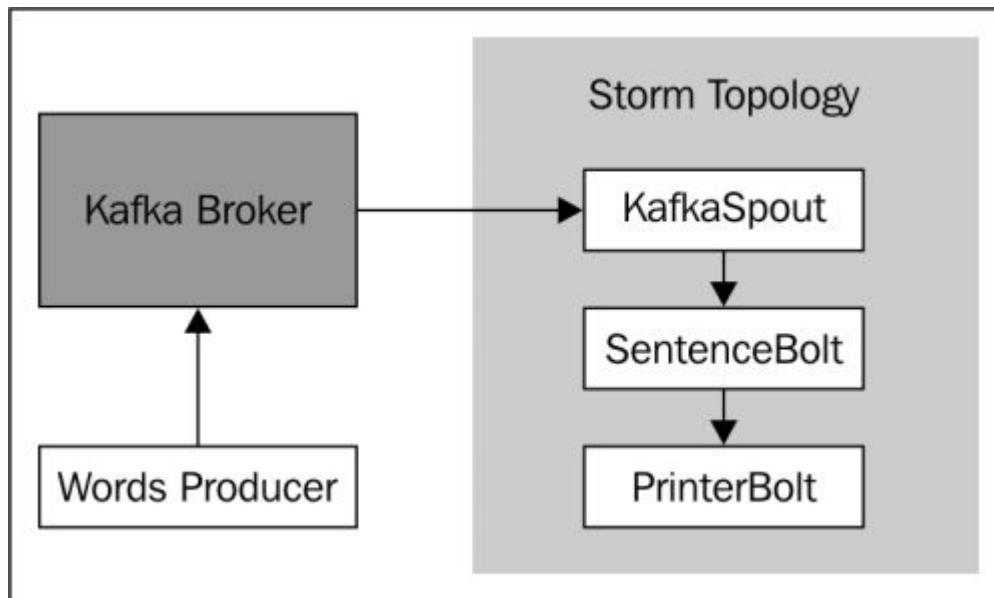
The message flow in the example Storm-Kafka integration

We have already seen the WordsProducer class that produces words into the Kafka broker. Now, we will create a Storm topology that will read these words from Kafka and aggregate them into sentences. For this, we will have one KafkaSpout in the application that will read the messages from Kafka and two bolts: SentenceBolt, which receives words from KafkaSpout and then aggregates them into sentences which are then passed onto PrinterBolt, which simply prints them on the output stream. We will be running this topology in a local mode. Perform the following steps to create the Storm topology:

1. Create a new Maven project with the com.learningstorm group ID and the kafka-storm-topology artifact ID.
2. Add the following dependencies for KafkaSpout and Storm in the pom.xml file:
3.   <!-- Dependency for Storm-Kafka spout   -->
4.
5.   <dependency>
6.   <groupId>net.wurstmeister.storm</groupId>
7.   <artifactId>storm-kafka-0.8-plus</artifactId>
8.   <version>0.4.0</version>
9.   </dependency>
10.
11.   <!-- Dependency for Storm -->
12.
13.   <dependency>
14.   <groupId>storm</groupId>
15.   <artifactId>storm-core</artifactId>

```
16.    <version>0.9.0.1</version>
17.    </dependency>
18.
19.  <!-- Utilities -->
20.
21.  <dependency>
22.    <groupId>commons-collections</groupId>
23.    <artifactId>commons-collections</artifactId>
24.    <version>3.2.1</version>
25.    </dependency>
26.
27.  <dependency>
28.    <groupId>com.google.guava</groupId>
29.    <artifactId>guava</artifactId>
30.    <version>15.0</version>
     </dependency>
```

31. Add the exec-maven-plugin plugin to the pom.xml file so that we are able to run the topology from the command line in a local mode using the following code:

```
32.  <plugin>
33.    <groupId>org.codehaus.mojo</groupId>
34.    <artifactId>exec-maven-plugin</artifactId>
35.    <version>1.2.1</version>
36.    <executions>
37.      <execution>
38.        <goals>
39.          <goal>exec</goal>
40.        </goals>
41.      </execution>
42.    </executions>
43.    <configuration>
44.      <executable>java</executable>
45.      <includeProjectDependencies>true</includeProjectDependencies>
46.      <includePluginDependencies>false</includePluginDependencies>
47.      <classpathScope>compile</classpathScope>
48.      <mainClass>${main.class}</mainClass>
49.    </configuration>
     </plugin>
```

50. Add the maven-assembly-plugin plugin to the pom.xml file so that we can package the topology to deploy it on Storm using the following code:

```
51.    <plugin>

52.      <artifactId>maven-assembly-plugin</artifactId>

53.      <configuration>

54.        <descriptorRefs>

55.          <descriptorRef>jar-with-dependencies</descriptorRef>

56.        </descriptorRefs>

57.        <archive>

58.          <manifest>

59.            <mainClass></mainClass>

60.          </manifest>

61.        </archive>

62.      </configuration>

63.      <executions>

64.        <execution>

65.          <id>make-assembly</id>

66.          <phase>package</phase>

67.          <goals>

68.            <goal>single</goal>

69.          </goals>

70.        </execution>

71.      </executions>

     </plugin>
```

72. Now, add the repositories for the KafkaSpout dependencies in the pom.xml file:

```
73.    <repositories>

74.      <repository>

75.        <id>github-releases</id>

76.        <url>http://oss.sonatype.org/content/repositories/github-releases/</url>

77.      </repository>

78.      <repository>

79.        <id>clojars.org</id>

80.        <url>http://clojars.org/repo</url>

81.      </repository>

     </repositories>
```

82. Now, we will first create SentenceBolt, which will aggregate the words into sentences. For this, create a class called SentenceBolt in the com.learningstorm.kafka package. The following is the code for the SentenceBolt class with explanation:

83.   public class SentenceBolt extends BaseBasicBolt {

84.   // list used for aggregating the words

85.   private List<String> words = new ArrayList<String>();

86.   public void execute(Tuple input, BasicOutputCollector collector) {

87.   // Get the word from the tuple

88.   String word = input.getString(0);

89.   if(StringUtils.isBlank(word)){

90.   // ignore blank lines

91.   return;

92.   }

93.   System.out.println("Received Word:" + word);

94.   // add word to current list of words

95.   words.add(word);

96.   if (word.endsWith(".")) {

97.   // word ends with '.' which means this is the end

98.   // the SentenceBolt publishes a sentence tuple

99.   collector.emit(ImmutableList.of((Object) StringUtils.join(words, ' ')));

100.

101.   // and reset the words list.

102.   words.clear();

103.   }

104.   }

105.   public void declareOutputFields(OutputFieldsDeclarer declarer) {

106.   // here we declare we will be emitting tuples with

107.   // a single field called "sentence"

108.   declarer.declare(new Fields("sentence"));

109.   }
    }

110. Next is PrinterBolt, which just prints the sentences that are received. Create the PrinterBolt class in the com.learningstorm.kafka package. The following is the code with explanation:

111.   public class PrinterBolt extends BaseBasicBolt {

112.   public void execute(Tuple input, BasicOutputCollector collector) {

113.   // get the sentence from the tuple and print it

114.   String sentence = input.getString(0);

```
115.        System.out.println("Received Sentence: " + sentence);
116.    }
117.    public void declareOutputFields(OutputFieldsDeclarer declarer) {
118.        // we don't emit anything
119.    }
        }
```

120. Now, we will create KafkaTopology, which will define KafkaSpout and wire it with PrinterBolt and SentenceBolt. Create a new KafkaTopology class in the com.learningstorm.kafka package. The following is the code with explanation:

```
121.  public class KafkaTopology {
122.    public static void main(String[] args) throws
123.    AlreadyAliveException, InvalidTopologyException {
124.        // zookeeper hosts for the Kafka cluster
125.        ZkHosts zkHosts = new ZkHosts("localhost:2181");
126.        // Create the KafkaSpout configuration
127.        // Second argument is the topic name
128.        // Third argument is the ZooKeeper root for Kafka
129.        // Fourth argument is consumer group id
130.        SpoutConfig kafkaConfig = new SpoutConfig(zkHosts, "words_topic", "", "id7");
131.        // Specify that the kafka messages are String
132.        kafkaConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
133.        // We want to consume all the first messages in
134.        // the topic every time we run the topology to
135.        // help in debugging. In production, this
136.        // property should be false
137.        kafkaConfig.forceFromStart = true;
138.        // Now we create the topology
139.        TopologyBuilder builder = new TopologyBuilder();
140.        // set the kafka spout class
141.        builder.setSpout("KafkaSpout", new KafkaSpout(kafkaConfig), 1);
142.        // configure the bolts
143.        builder.setBolt("SentenceBolt",                new                SentenceBolt(),
    1).globalGrouping("KafkaSpout");
144.        builder.setBolt("PrinterBolt",                 new                 PrinterBolt(),
    1).globalGrouping("SentenceBolt");
145.        // create an instance of LocalCluster class
146.        // for executing topology in local mode.
147.        LocalCluster cluster = new LocalCluster();
```

```
148.        Config conf = new Config();
149.        // Submit topology for execution
150.        cluster.submitTopology("KafkaToplogy", conf, builder.createTopology());
151.        try {
152.            // Wait for some time before exiting
153.            System.out.println("Waiting to consume from kafka");
154.            Thread.sleep(10000);
155.        } catch (Exception exception) {
156.            System.out.println("Thread interrupted exception : " + exception);
157.        }
158.        // kill the KafkaTopology
159.        cluster.killTopology("KafkaToplogy");
160.        // shut down the storm test cluster
161.        cluster.shutdown();
162.    }
    }
```

163. Now, we will run the topology. Make sure the Kafka cluster is running and you have executed the producer in the last section so that there are messages in Kafka for consumption.

Run the topology by executing the following command:

mvn clean compile exec:java
-Dmain.class=com.learningstorm.kafka.KafkaTopology

This will execute the topology. You should see messages similar to the following output:

RecievedWord:One
RecievedWord:morning,
RecievedWord:when
RecievedWord:Gregor
RecievedWord:Samsa
RecievedWord:woke
RecievedWord:from
RecievedWord:troubled
RecievedWord:dreams,
RecievedWord:he
RecievedWord:found
RecievedWord:himself

RecievedWord:transformed

RecievedWord:in

RecievedWord:his

RecievedWord:bed

RecievedWord:into

RecievedWord:a

RecievedWord:horrible

RecievedWord:vermin

RecievedSentence:One morning, when Gregor Samsa woke from troubled dreams, he found himself transformed in his bed into a horrible vermin.

So, we were able to consume messages from Kafka and process them in a Storm topology.

Summary

In this chapter, we learned about the basics of Apache Kafka and how to use it as part of a real-time stream processing pipeline build with Storm. We learned about the architecture of Apache Kafka and how it can be integrated into Storm processing by using KafkaSpout.

In the next chapter, we will have a look at Trident, which is a high-level abstraction for defining Storm topologies. We will also see transactional topologies in Storm that support exactly-once message processing semantics.

Chapter 5. Exploring High-level Abstraction in Storm with Trident

In the previous chapter, we learned how we can set up a cluster of Kafka, how we can write the Kafka producer, integration of Kafka and Storm, and so on.

In this chapter, we will cover the following topics:

- Introducing Trident
- Trident's data model
- Trident functions, filters, and projections
- Trident repartitioning operations
- Trident aggregators
- Trident's groupBy operation
- A non-transactional topology
- A sample Trident topology
- Trident's state
- Distributed RPC
- When to use Trident

Introducing Trident

Trident is a high-level abstraction built on top of Storm. Trident supports stateful stream processing, while pure Storm is a stateless processing framework. The main advantage of using Trident is that it will guarantee that every message that enters the topology is processed only once, which is difficult to achieve in the case of Vanilla Storm. The concept of Trident is similar to high-level batch processing tools such as Cascading and Pig developed over Hadoop. Trident processes the input stream as small batches to achieve exactly once processing in Storm. We will cover this in greater detail in the *Maintaining the topology state with Trident* section of this chapter.

So far, we have learned that in the Vanilla Storm topology, the spout is the source of tuples, a tuple is a unit of data that can be processed by a Storm application, and the bolt is the processing powerhouse where we write the transformation logic. However, in the Trident topology, the bolt is replaced with higher-level semantics of functions, aggregates, filters, and states.

Understanding Trident's data model

The TridentTuple interface is the data model of a Trident topology. The TridentTuple interface is the basic unit of data that can be processed by a Trident topology. Each tuple consists of a predefined list of fields. The value of each field can be a byte, character, integer, long, float, double, Boolean, or byte array. During the construction of a topology, operations are performed on the tuple, which will either add new fields to the tuple or replace the tuple with a new set of fields.

Each of the fields in a tuple can be accessed by the name getValueByField(String) or its positional index getValue(int) in the tuple. The TridentTuple interface also provides convenient methods such as getIntegerByField(String) that saves you from type casting the objects.

Writing Trident functions, filters, and projections

This section covers the definitions of Trident functions, filters, and projections. Trident functions, filters, and projections are used to modify or filter the input tuples based on certain criteria. This section also covers how we can write Trident functions, filters, and projections.

Trident functions

Trident's function contain the logic to modify the original tuple. A function gets a set of fields of a tuple as input and emits one or more tuples as output. The output fields of the tuple are merged with the input fields of a tuple to form the complete tuple, which will pass to the next action in the topology. If the function emits a zero tuple that corresponds to the input tuple, then that tuple is removed from the stream.

We can write a custom Trident function by extending the storm.trident.operation.BaseFunction class and implementing the execute(TridentTuple tuple, TridentCollector collector) method.

Let's write a sample Trident function that will calculate the sum of first two fields and emit the new sum field. The following is the code of the SumFunction class:

```
public class SumFunction extends BaseFunction {
    private static final long serialVersionUID = 5L;
    public void execute(TridentTuple tuple, TridentCollector collector) {
        int number1 = tuple.getInteger(0);
```

```
        int number2 = tuple.getInteger(1);
        int sum = number1+number2;
        // emit the sum of first two fields
        collector.emit(new Values(sum));


    }
}
```

Suppose we are getting the dummyStream stream as an input that contains four fields, a, b, c, and d, and only the a and b fields are passed as input fields to the SumFunction class. The SumFunction class emits the new sum field. The sum field emitted by the execute method of the SumFunction class is merged with the input tuple to form the complete tuple. Hence, the total number of fields in the output tuple is 5 (a, b, c, d, and sum). The following is a sample piece of code that shows how we can pass the input fields and the name of a new field to the Trident function:

dummyStream.each(new Fields("a","b"), new SumFunction (), new Fields("sum"))

The following diagram shows the input tuples, SumFunction, and output tuples. The output tuples contain five fields, a, b, c, d, and sum:
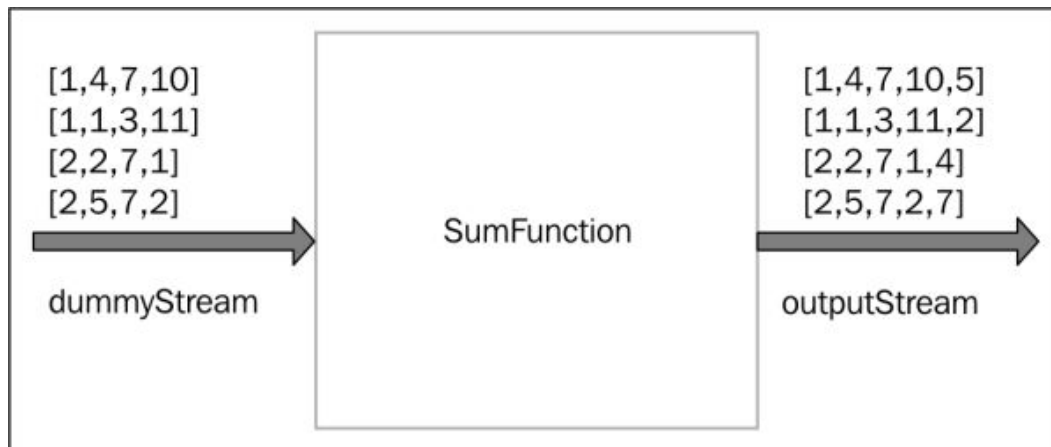
Working of the Trident function

Trident filters

A Trident filter gets a set of fields as input and returns either true or false depending on whether certain conditions are satisfied or not. If true is returned, then the tuple is kept in the output stream; otherwise, the tuple is removed from the stream.

We can write a custom Trident filter by extending the storm.trident.operation.BaseFilter class and implementing the isKeep(TridentTuple tuple) method.

Let's write a sample Trident filter that will check whether the sum of the input fields is even or odd. If the sum is even, then the Trident filter emits true; otherwise, it emits false. The following is the code of the CheckEvenSumFilter class:

```
public static class CheckEvenSumFilter extends BaseFilter{

  private static final long serialVersionUID = 7L;

  public boolean isKeep(TridentTuple tuple) {
    int number1 = tuple.getInteger(0);
    int number2 = tuple.getInteger(1);
    int sum = number1+number2;
    if(sum % 2 == 0) {
      return true;
    }
    return false;
  }

}
```

Suppose you get dummyStream as input, which contains four fields, a, b, c, and d, and only the a and b fields are passed as input fields in the CheckEvenSumFilter class. The execute method of the CheckEvenSumFilter class will emit only those tuples whose sum of the a and b fields is even. The following is the sample piece of code that shows how we can define the input fields for the Trident filter:

```
dummyStream.each(new Fields("a","b"), new CheckEvenSumFilter ())
```

The following diagram shows the input tuples, CheckEvenSumFilter, and output tuples. The outputStream stream contains only those tuples whose sum of the a and b fields is even.
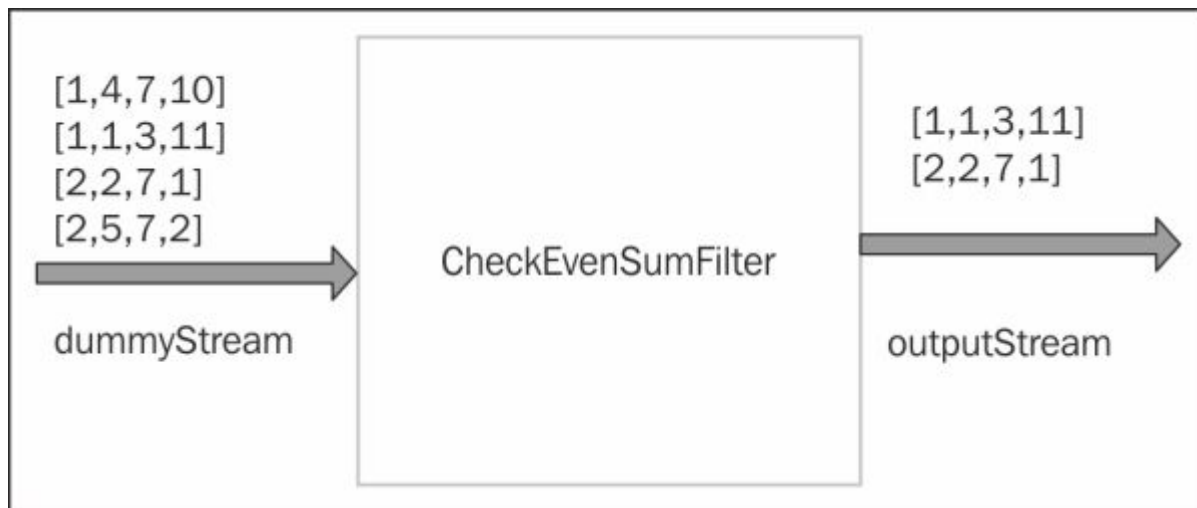
Working of the Trident filter

Trident projections

Trident projections keep only those fields in the stream that are specified in the projection operation. Suppose an input stream contains three fields, x, y, and z, and we are passing the x field in the projection operation. Then, the output stream will contain tuples with the single field x. The following is the piece of code that shows how we can use the projection operation:

mystream.project(new Fields("x"))

The following diagram shows the projection operation:
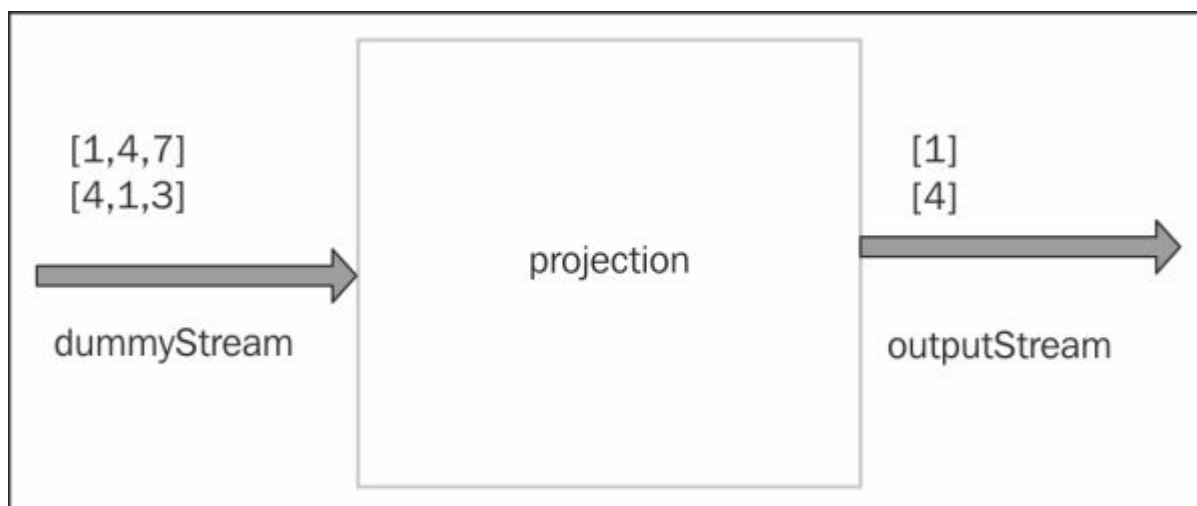
Working of the Trident projection

Trident repartitioning operations

By performing repartitioning operations, a user can partition tuples across multiple tasks. A repartitioning operation doesn't make any changes to the content of tuples. Also, the tuples will only pass over the network in the case of a repartitioning operation. The different types of repartitioning operations are explained in this section.

The shuffle operation

The shuffle repartitioning operation partitions the tuples in a uniform, random way across multiple tasks. This repartitioning operation is generally used when we want to distribute our processing load uniformly across tasks. The following diagram shows how the input tuples are repartitioned using the shuffle operation:
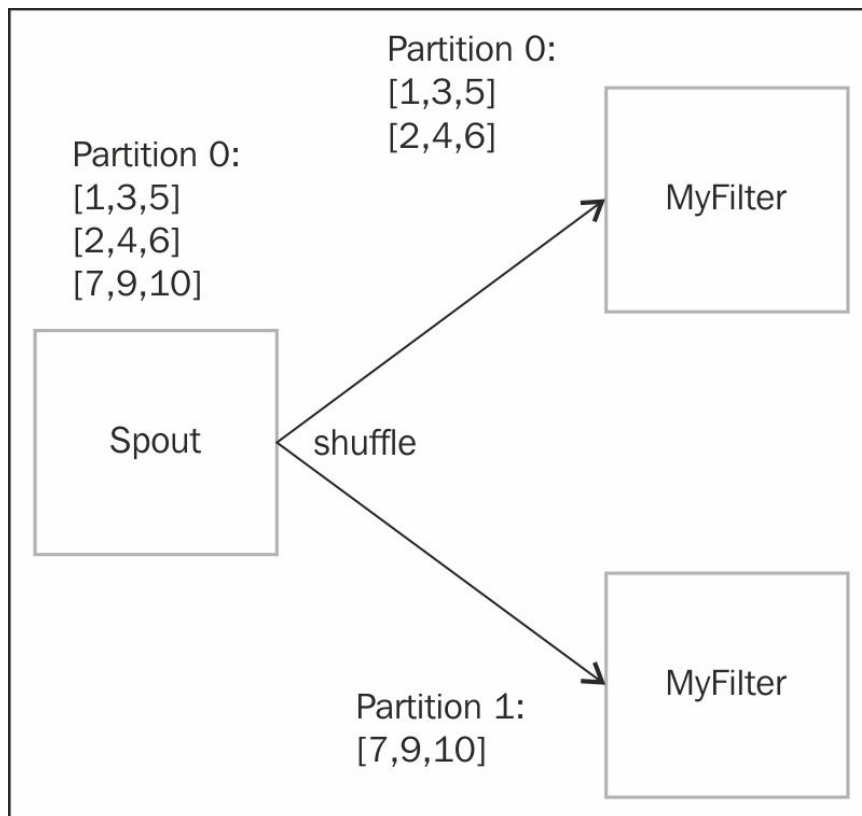
Working of the shuffle repartitioning operation

The following piece of code shows how we can use the shuffle operation:

mystream.shuffle().each(new Fields("a","b"), new myFilter()).parallelismHint(2)

The partitionBy operation

The partitionBy repartitioning operation enables you to partition a stream on the basis of some fields in the tuples. For example, if you want all tweets from a particular user to be delivered to the same target partition, then you can partition the tweet stream by applying the partitionBy operation on the username field in the following manner:

mystream.partitionBy(new Fields("username")).each(new Fields("username","text"), new myFilter()).parallelismHint(2)

The partitionBy operation applies the *target partition = hash (fields) % (number of target partition)* formula to decide the target partition.

As the preceding formula shows, the partitionBy operation calculates the hash of input fields to decide the target partition. Hence, it does not guarantee that all the tasks will get tuples to process. For example, if you have applied a partitionBy operation on a field, say X, with only two possible values, A and B, and created two tasks for the myFilter filter, then it is possible that both *hash (A) % 2* and *hash (B) % 2* are equal. This will result in all the tuples being routed to a single task and the other being completely idle.

The following diagram shows how the input tuples are repartitioned using the partitionBy operation:
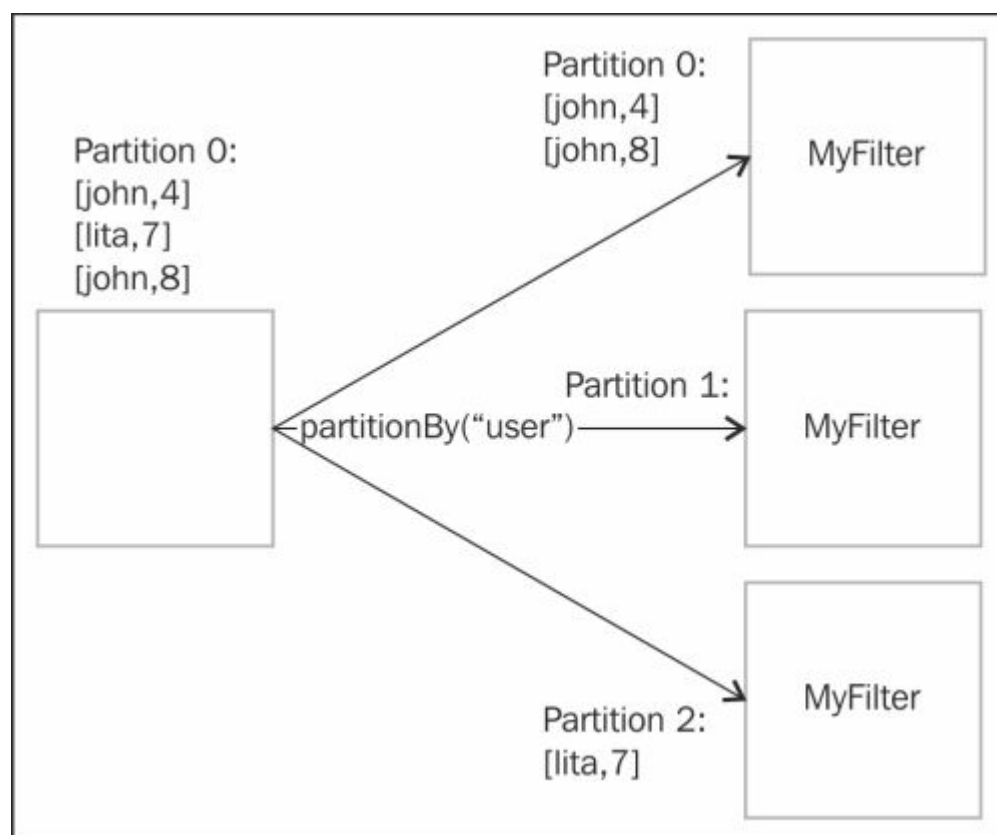
Working of the partitionBy repartitioning operation

As seen in the preceding diagram, partitions **0** and **2** contain the set of tuples, but partition **1** is empty.

The global operation

The global repartitioning operation routes all tuples to the same partition. Hence, the same target partition is selected for all the batches in the stream. The following diagram shows how the tuples are repartitioned using the global operation:
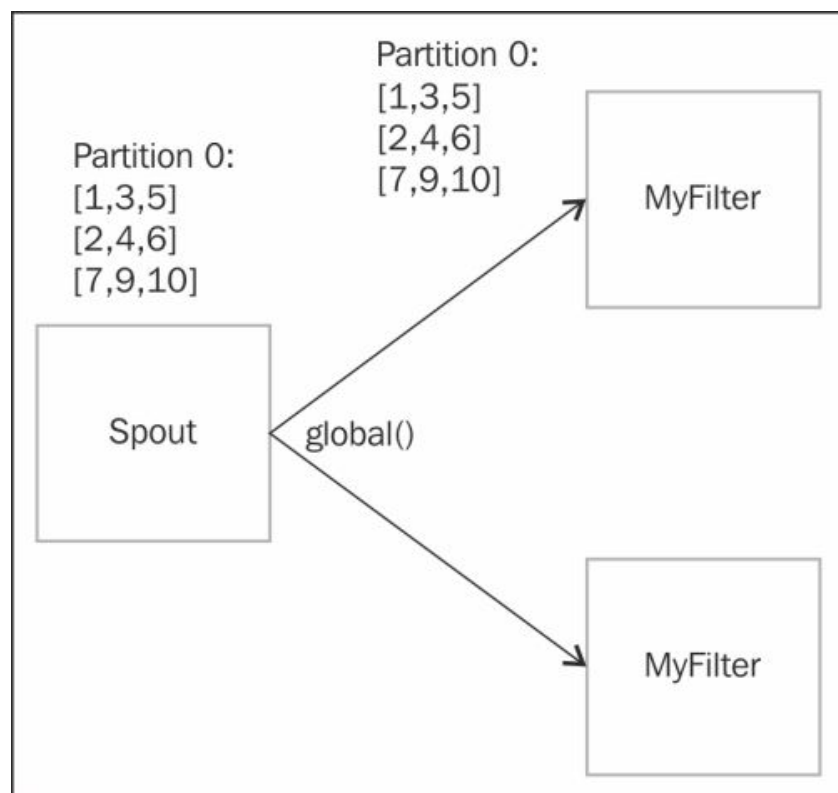
Working of the global repartitioning operation

The following piece of code shows how we can use the global operation:

mystream.global().each(new Fields("a","b"), new myFilter()).parallelismHint(2)

The broadcast operation

The broadcast operation is a special repartitioning operation that does not partition the tuples but replicates them to all partitions. The following is a diagram that shows how the tuples are sent over the network:
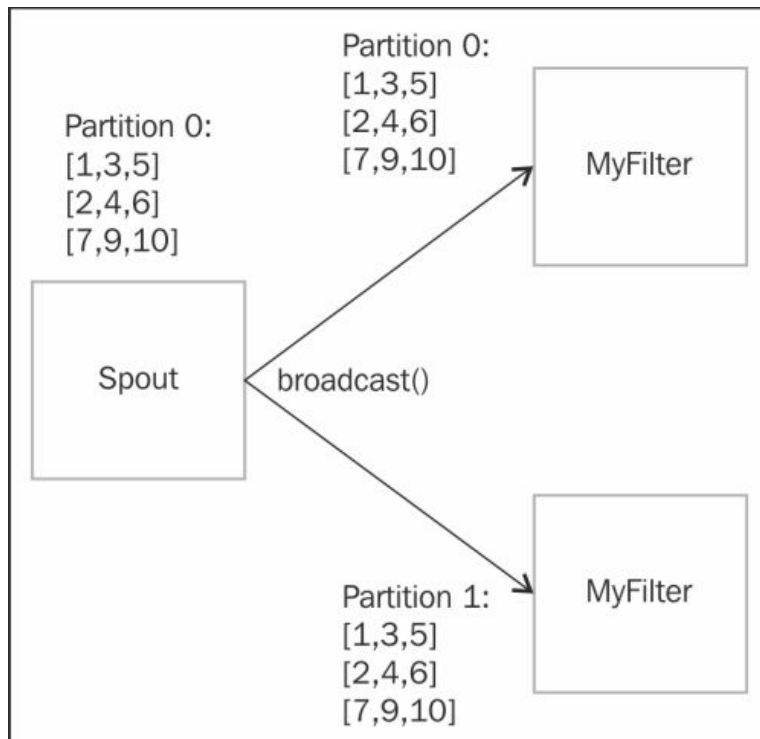
Working of the broadcast repartitioning operation

The following piece of code shows how we can use the broadcast operation:

mystream.broadcast().each(new Fields("a","b"), new myFilter()).parallelismHint(2)

The batchGlobal operation

This repartitioning operation routes all tuples that belong to one batch to the same target partition. The other batches of the same stream may go to a different partition. As the name suggests, this repartition is global at the batch level. The following diagram shows how the tuples are repartitioned using the batchGlobal operation:
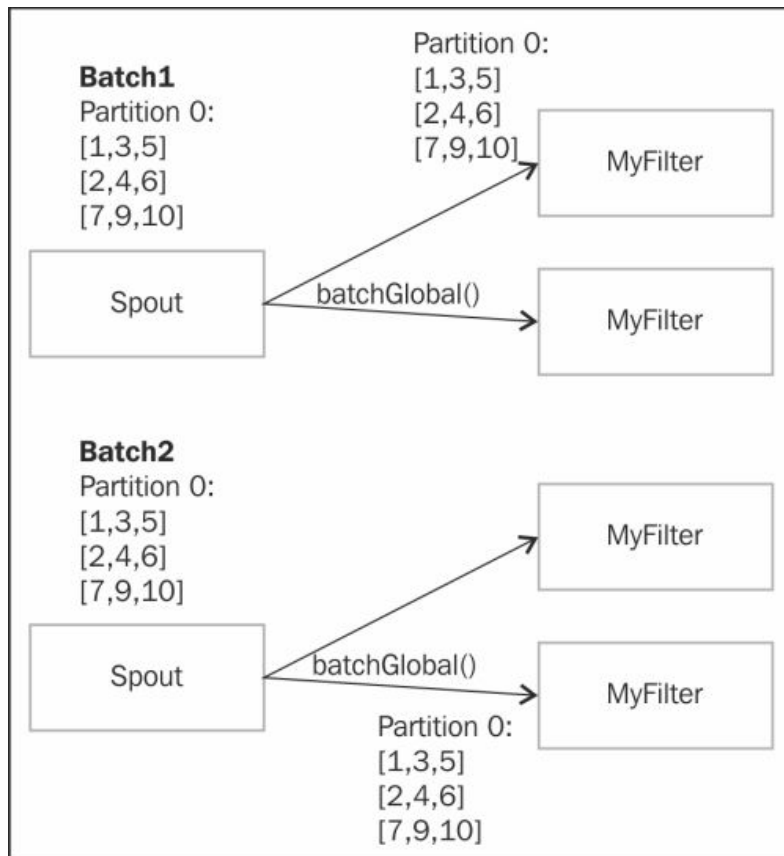
Working of the batchGlobal repartitioning operation

The following piece of code shows how we can use the batchGlobal operation:

mystream.batchGlobal().each(new Fields("a","b"), new myFilter()).parallelismHint(2)

The partition operation

If none of the preceding repartitioning operations fit your use case, you can define your own custom repartition function by implementing the backtype.storm.grouping.CustomStreamGrouping interface. The following is a sample custom repartition that partitions the stream on the basis of the values of the country field:

```
public class CountryRepartition implements CustomStreamGrouping, Serializable {

    private static final long serialVersionUID = 1L;
    private static final Map<String, Integer> countries = ImmutableMap.of(
        "India", 0,
        "Japan", 1,
        "United State", 2,
        "China", 3,
```

```
    "Brazil", 4
  );

  private int tasks = 0;

  public void prepare(WorkerTopologyContext context, GlobalStreamId stream, List<Integer>
targetTasks){
    tasks = targetTasks.size();
  }

  public List<Integer> chooseTasks(int taskId, List<Object> values) {
    String country = (String) values.get(0);
    return ImmutableList.of(countries.get(country) % tasks);
  }
}
```

The CountryRepartition class implements the backtype.storm.grouping.CustomStreamGrouping interface. The chooseTasks() method contains the repartitioning logic to identify the next task in the topology for the input tuple. The prepare() method calls at the start and performs the initialization activity.

Trident aggregators

The Trident's aggregator is used to perform aggregation operations on an input batch or partition or stream. For example, let's say a user wants to count the number of tuples present in each batch, then he/she can use the count aggregator to count the number of tuples in each batch. The output of the Aggregator interface completely replaces the value of the input tuple. There are three types of aggregators available in Trident:

- The partition aggregate
- The aggregate
- The persistence aggregate

Let's understand each type of aggregator in detail.

The partition aggregate

As the name suggests, the partition aggregate works on each partition instead of the entire batch. The output of the partition aggregate completely replaces the input tuple. Also, the output of the

partition aggregate contains a single field tuple. The following is the piece of code that shows how we can use the partitionAggregate method:

mystream.partitionAggregate(new Fields("x"), new Count(), new Fields("count"))

For example, we have an input stream that contains the x and y fields, and we will apply a partitionAggregate function on each partition; the output tuples contain a single field called count. The count field represents the number of tuples present in the input partition. The following is a diagram that shows the working of the partitionAggregate function:
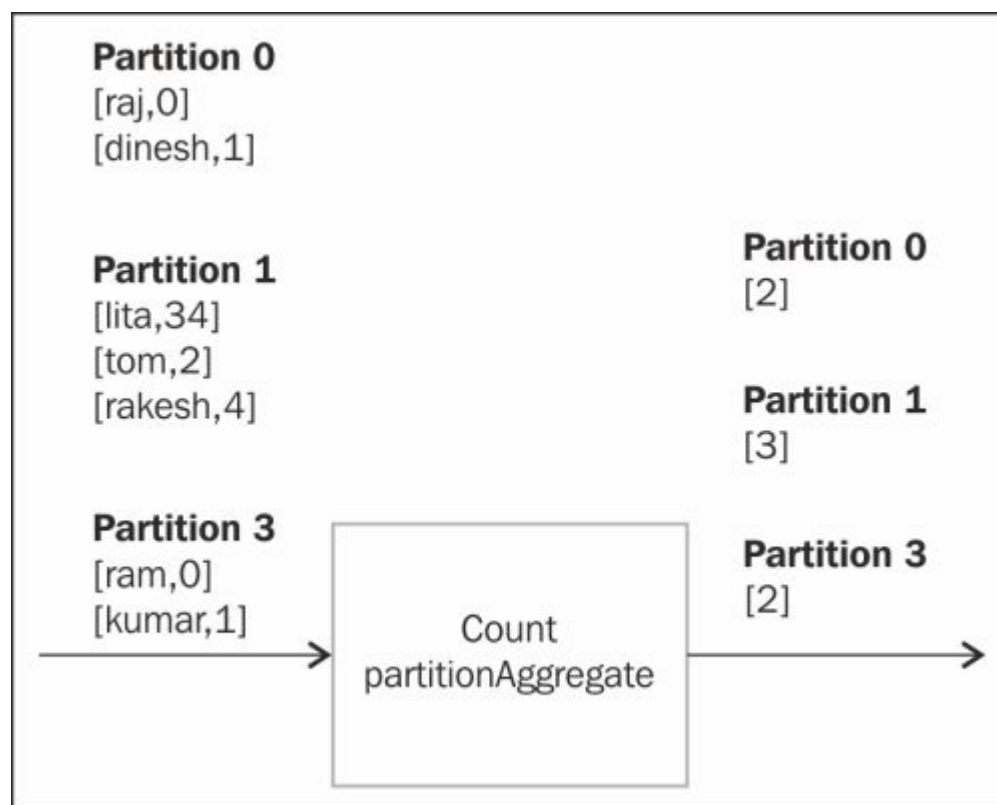
Working of the partition aggregate

The aggregate

An aggregate works on each batch. During the aggregate process, the tuples are first repartitioned using the global operation to combine all partitions of the same batch into a single partition. Then, we run the aggregation function on each batch. The following is the piece of code that shows how we can use the aggregate function:

mystream.aggregate(new Fields("x"), new Count() ,new Fields("count"))

Three types of the Aggregator interface are available in Trident:

- ReducerAggregator
- Aggregator
- CombinerAggregator

The preceding three Aggregator interfaces can also be used with the partition aggregate.

The ReducerAggregator interface

The ReducerAggregator interface first runs the global repartitioning operation on the input stream to combine all the partitions of the same batch into a single partition, and then runs the aggregation function on each batch. The ReducerAggregator<T> interface contains the following methods:

- init(): This method returns the initial value
- reduce(T curr, TridentTuple tuple): This method iterates over the input tuples and emits a single tuple with a single value

The following example code shows how we can implement a Sum class using the ReducerAggregator interface:

```
public static class Sum implements ReducerAggregator<Long> {

  private static final long serialVersionUID = 1L;
  //return the initial value zero
  public Long init() {
    return 0L;
  }
  //Iterates on the input tuples, calculate the sum and
  //produce the single tuple with single field as output
  public Long reduce(Long curr, TridentTuple tuple) {
    return curr+tuple.getLong(0);
  }

}
```

The Aggregator interface

The Aggregator interface first runs the global repartitioning operation on the input stream to combine all the partitions of the same batch into a single partition, and then runs the

aggregation function on each batch. By definition, the Aggregator interface looks very similar to the ReduceAggregator interface. The BaseAggregator<State> interface contains the following methods:

- init(Object batchId, TridentCollector collector): The init() method is called before starting the processing of the batch. This method returns the State object, which we will use to save the state of the batch. This object is used by the aggregate() and complete() methods.
- aggregate(State s, TridentTuple tuple, TridentCollector collector): This method iterates over each tuple of the given batch. It also updates the state in the State object after processing each tuple.
- complete(State state, TridentCollector tridentCollector): This method is called at the end if all tuples of the given batch are processed. This method returns a single tuple corresponding to each batch.

The following is an example that shows how we can implement the SumAsAggregator class using the BaseAggregator interface:

```
public static class SumAsAggregator extends BaseAggregator<SumAsAggregator.State> {

    private static final long serialVersionUID = 1L;
    // state class
    static class State {
        long count = 0;
    }
    // Initialize the state
    public State init(Object batchId, TridentCollector collector) {
        return new State();
    }
    // Maintain the state of sum into count variable.
    public void aggregate(State state, TridentTuple tridentTuple, TridentCollector tridentCollector) {
        state.count = tridentTuple.getLong(0) + state.count;
    }
    // return a tuple with single value as output
    // after processing all the tuples of given batch.
    public void complete(State state, TridentCollector tridentCollector) {
        tridentCollector.emit(new Values(state.count));
    }

}
```

The CombinerAggregator interface

The CombinerAggregator interface first runs the partition aggregate on each partition, then runs the global repartitioning operation to combine all the partitions of the same batch into a single partition, and then reruns the aggregator on the final partition to emit the desired output. The network transfer in the case of the CombinerAggregator interface is less compared to the other two aggregators. Hence, the overall performance of the CombinerAggregator interface is better compared to the Aggregator and ReduceAggregator interfaces. The CombinerAggregator<T> interface contains the following methods:

- init(): This method runs on each input tuple to retrieve the field values from the tuples.
- combine(T val1, T val2): This method combines the values of tuples. It emits a single tuple with a single field as output.
- zero(): This method returns a zero value if the input partition contains no tuple.

The following example code shows how we can implement the Sum class using the CombinerAggregator interface:

```
public class Sum implements CombinerAggregator<Number> {

  private static final long serialVersionUID = 1L;

  public Number init(TridentTuple tridentTuple) {
    return (Number) tridentTuple.getValue(0);
  }

  public Number combine(Number number1, Number number2) {
    return Numbers.add(number1, number2);
  }

  public Number zero() {
    return 0;
  }

}
```

The persistent aggregate

The persistent aggregate works on all tuples across all the batches in a stream and persists the aggregate result to the source of the state (Memory, Memcached, Cassandra, or some other database). The following piece of code shows how we can use the persistentAggregate function:

mystream.persistentAggregate(new   MemoryMapState.Factory(),   new   Fields("select"),new Count(),new Fields("count"));

We will discuss more on this in the *Maintaining the topology state with Trident* section.

Aggregator chaining

Trident provides us with a feature to apply multiple aggregators on the same input stream, and this process is called **aggregator chaining**. The following piece of code shows how we can use aggregator chaining:

mystream.chainedAgg().partitionAggregate(new Fields("b"), new Average(), new
Fields("average")).partitionAggregate(new Fields("b"), new Sum(), new
Fields("sum")).chainEnd();

We have applied the Average() and Sum() aggregators on each partition. The output of the chainedAgg() function contains a single tuple corresponding to each input partition. The output tuple contains two fields, sum and average.

The following diagram shows how aggregator chaining works:
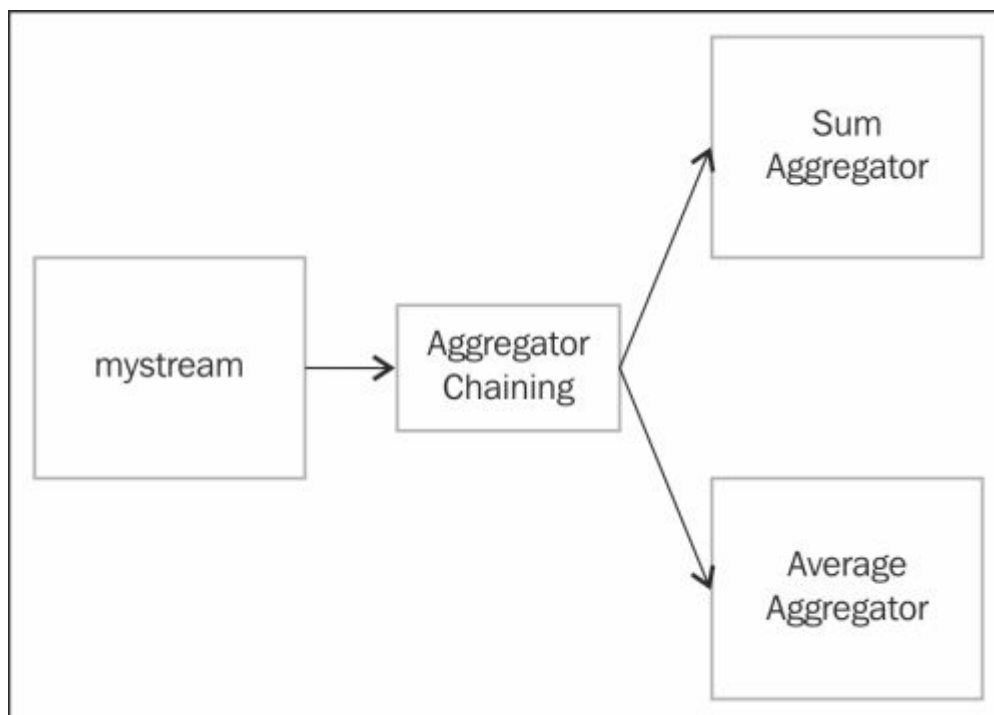
Utilizing the groupBy operation

The groupBy operation doesn't involve any repartitioning. The groupBy operation converts the input stream into a grouped stream. The main function of the groupBy operation is to modify the behavior of the subsequent aggregate function. The following diagram shows how the groupBy operation groups the tuples of a single partition:
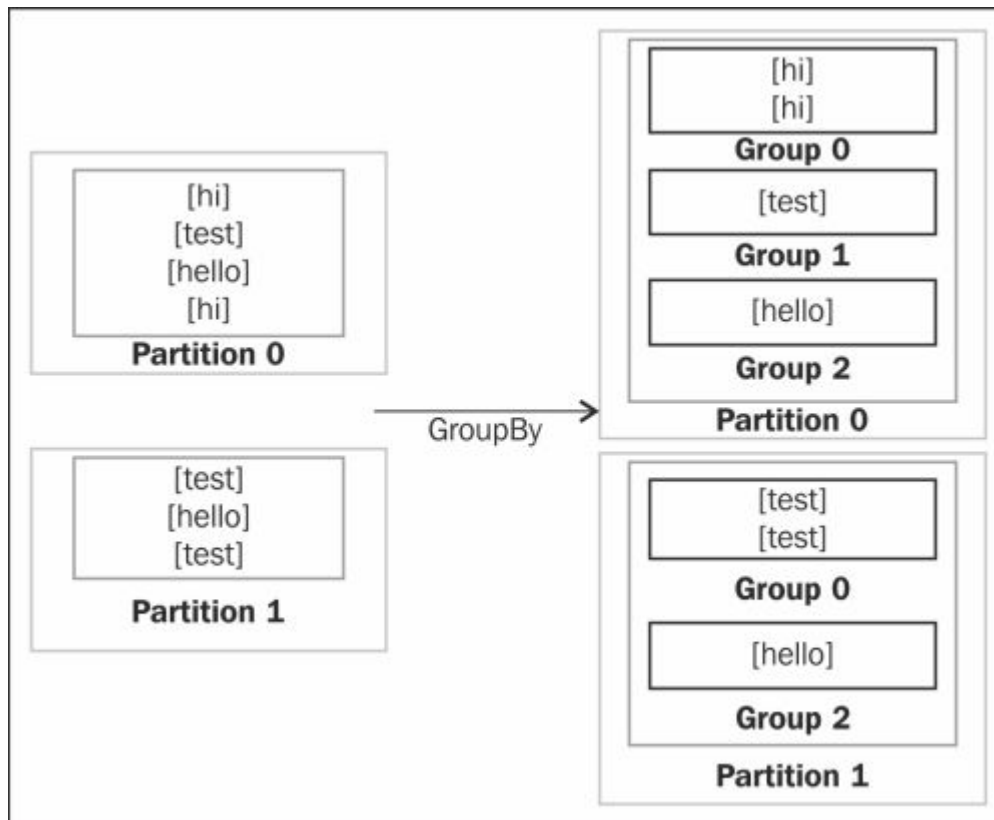
Working of the groupBy operation

- If the groupBy operation is used before the partition aggregate, then the partition aggregate will run the aggregate on each group created within the partition.
- If the groupBy operation is used before the aggregate, then in that case, tuples of the same batch are first repartitioned into a single partition and then the groupBy operation is applied on each single partition. At the end, it will perform the aggregate operation on each group.

So far, we have covered the basics of the Trident APIs. In the following section, we will cover how to write a non-transactional topology in Trident.

A non-transactional topology

In a non-transactional topology, a spout emits a batch of tuples and doesn't guarantees about what is in each batch. By processing behavior, we can divide the pipeline into two categories:

- **At-most-one-processing**: In this type of topology, failed tuples are not retried. Hence, the spout does not wait for an acknowledgment.
- **At-least-once-processing**: The failed tuples are re-entered into the processing pipeline. Hence, this type of topology guarantees that every tuple entered in to the processing pipeline must be processed at least once. The retried logic is handled at the spout end because the spout is the source of tuples in the Trident topology.

Let's understand how we can write a non-transactional spout by implementing the storm.trident.spout.IBatchSpout interface:

```
public class FakeTweetSpout implements IBatchSpout{

  private static final long serialVersionUID = 10L;
  private int batchSize;
  private HashMap<Long, List<List<Object>>> batchesMap = new HashMap<Long,
List<List<Object>>>();
  public FakeTweetSpout(int batchSize) {
    this.batchSize = batchSize;
  }

  private static final Map<Integer, String> TWEET_MAP = new HashMap<Integer, String>();
  static {
    TWEET_MAP.put(0, " Adidas #FIFA World Cup Chant Challenge ");
    TWEET_MAP.put(1, "#FIFA worldcup");
    TWEET_MAP.put(2, "#FIFA worldcup");
    TWEET_MAP.put(3, " The Great Gatsby is such a good #movie ");
    TWEET_MAP.put(4, "#Movie top 10");
  }

  private static final Map<Integer, String> COUNTRY_MAP = new HashMap<Integer,
String>();
  static {
    COUNTRY_MAP.put(0, "United State");
    COUNTRY_MAP.put(1, "Japan");
    COUNTRY_MAP.put(2, "India");
    COUNTRY_MAP.put(3, "China");
```

```java
    COUNTRY_MAP.put(4, "Brazil");
  }

  private List<Object> recordGenerator() {
    final Random rand = new Random();
    int randomNumber = rand.nextInt(5);
    int randomNumber2 = rand.nextInt(5);
    return                                                  new
Values(TWEET_MAP.get(randomNumber),COUNTRY_MAP.get(randomNumber2));
  }

  @Override
  public void ack(long batchId) {
    this.batchesMap.remove(batchId);

  }

  @Override
  public void close() {
    /*This method is used to destroy or close all the connection opened in open method.*/

  }

  @Override
  public void emitBatch(long batchId, TridentCollector collector){
    List<List<Object>> batches = this.batchesMap.get(batchId);
    if(batches == null) {
      batches = new ArrayList<List<Object>>();
      for (int i=0;i < this.batchSize;i++) {
        batches.add(this.recordGenerator());
      }
      this.batchesMap.put(batchId, batches);
    }
    for(List<Object> list : batches){
      collector.emit(list);
    }

  }
```

```java
    @Override
    public Map getComponentConfiguration() {
        /* This method is use to set the spout configuration like defining the parallelism, etc.*/
        return null;
    }

    @Override
    public Fields getOutputFields() {

        return new Fields("text","Country");
    }

    @Override
    public void open(Map arg0, TopologyContext arg1) {
        /*This method is used to initialize the variable, open the connection with external source,
etc. */

    }

}
```

The FakeTweetSpout class implements the storm.trident.spout.IBatchSpout interface. The construct of the FakeTweetSpout(int batchSize) method takes batchSize as an argument; if batchSize is 3, then every batch emitted by the FakeTweetSpout class contains three tuples. The recordGenerator() method contains logic to generate the fake tweet. The following is a sample fake tweet:

["Adidas #FIFA World Cup Chant Challenge", "Brazil"]
["The Great Gatsby is such a good movie","India"]

The getOutputFields() method returns two fields, text and country. The emitBatch(long batchId, TridentCollector collector) method uses the batchSize variable to decide the number of tuples in each batch and emits a batch to the processing pipeline.

The batchesMap collection contains batchId as the key and the batch of tuples as the value. All batches emitted by emitBatch(long batchId, TridentCollector collector) will be added to the batchesMap collection.

The ack(long batchId) method receives batchId as an acknowledgment and will remove the corresponding batch from the batchesMap collection.

A sample Trident topology

This section explains how you can write a Trident topology. We will perform the following steps to create a sample Trident topology:

1. Create a Maven project using com.learningstorm as the group ID and trident-example as the artifact ID.
2. Add the following dependencies and repositories in the pom.xml file:
3. <dependencies>
4.   <dependency>
5.   <groupId>junit</groupId>
6.   <artifactId>junit</artifactId>
7.   <version>3.8.1</version>
8.   <scope>test</scope>
9.   </dependency>
10.   <dependency>
11.   <groupId>storm</groupId>
12.   <artifactId>storm</artifactId>
13.   <version>0.9.0.1</version>
14.   <scope>provided</scope>
15.   </dependency>
16. </dependencies>
17.
18. <repositories>
19.   <repository>
20.   <id>clojars.org</id>
21.   <url>http://clojars.org/repo</url>
22.   </repository>
  </repositories>

23. Create a TridentUtility class in the com.learningstorm.trident_example package. This class contains a Trident filter and function:
24.   public class TridentUtility {
25.   /* Get the comma separated value as input, split the field by comma, and then emits multiple tuple as output.*/
26.   public static class Split extends BaseFunction {

```java
27.
28.        private static final long serialVersionUID = 2L;
29.
30.        public void execute(TridentTuple tuple, TridentCollector collector) {
31.            String countries = tuple.getString(0);
32.            for (String word : countries.split(",")) {
33.                collector.emit(new Values(word));
34.            }
35.        }
36.    }
37.
38.    /* This class extends BaseFilter and contain isKeep method which emits only those
    tuple which has #FIFA in text field.*/
39.    public static class TweetFilter extends BaseFilter {
40.
41.        private static final long serialVersionUID = 1L;
42.
43.        public boolean isKeep(TridentTuple tuple) {
44.            if (tuple.getString(0).contains("#FIFA")) {
45.                return true;
46.            } else {
47.                return false;
48.            }
49.        }
50.
51.    }
52.
53.    /* This class extends BaseFilter and contain isKeep method which will print the input
    tuple.*/
54.    public static class Print extends BaseFilter {
55.
56.        private static final long serialVersionUID = 1L;
57.
58.        public boolean isKeep(TridentTuple tuple) {
59.            System.out.println(tuple);
60.            return true;
61.        }
62.
```

63.   }
    }

The TridentUtility class contains the following three inner classes:

- The Split class extends the storm.trident.operation.BaseFunction class and contains the execute(TridentTuple tuple, TridentCollector collector) method. The execute() method takes a comma-separated value as the input, splits the input value, and emits multiple tuples as the output.
- The TweetFilter class extends the storm.trident.operation.BaseFilter class and contains the isKeep(TridentTuple tuple) method. The isKeep() method takes the tuple as the input and checks whether the input tuple contains the #FIFA value in the text field or not. If the tuple contains #FIFA in the text field, then the method returns true; otherwise, it returns false.
- The Print class extends the storm.trident.operation.BaseFilter class and contains the isKeep(TridentTuple tuple) method. The isKeep() method prints the input tuple and returns true.

2. Create a TridentHelloWorldTopology class in the com.learningstorm.trident_example package. This class defines the sample Trident topology; its code is as follows:

3.   public class TridentHelloWorldTopology {
4.
5.   public static void main(String[] args) throws Exception {
6.   Config conf = new Config();
7.   conf.setMaxSpoutPending(20);
8.   if (args.length == 0) {
9.   LocalCluster cluster = new LocalCluster();
10.   cluster.submitTopology("Count", conf, buildTopology());
11.   } else {
12.   conf.setNumWorkers(3);
13.   StormSubmitter.submitTopology(args[0], conf, buildTopology());
14.   }
15.   }
16.
17.   public static StormTopology buildTopology() {
18.
19.   FakeTweetSpout spout = new FakeTweetSpout(10);
20.   TridentTopology topology = new TridentTopology();
21.

```
22.        topology.newStream("faketweetspout",    spout).shuffle().each(new    Fields("text",
   "Country"),              new               TridentUtility.TweetFilter()).groupBy(new
   Fields("Country")).aggregate(new     Fields("Country"),     new     Count(),     new
   Fields("count")).each(new                    Fields("count"),                    new
   TridentUtility.Print()).parallelismHint(2);
23.
24.        return topology.build();
25.    }
   }
```

Let's understand the code line by line. Firstly, we are creating an object of the TridentTopology class to define the Trident computation.

The TridentTopology class contains a method called newStream() that will take the input source as an argument. In this example, we are using the FakeTweetSpout class created in the *A non-transactional topology* section as an input source. Like Storm, Trident also maintains the state of each input source in ZooKeeper. Here, the faketweetspout string specifies the node name in ZooKeeper where Trident maintains the metadata.

The spout emits a stream which has two fields, text and country.

We are repartitioning the batch of tuples emitted by the input source using the shuffle() operation. The next line of the topology definition applies the TweetFilter class on each tuple. The TweetFilter class filters out all those tuples that do not contain the #FIFA keyword.

The output of the TweetFilter class is grouped by the country field. Then, we will apply the count aggregator to count the number of tweets for each country. Finally, we will apply a Print class to print the output of the aggregate method.

The following is the console output of the TridentHelloWorldTopology class topology:

```
3141 [Thread-9] INFO  backtype.storm.daemon.executor - Loading executor spout0:[7 7]
3142 [Thread-9] INFO  backtype.storm.daemon.executor - Loaded executor tasks spout0:[7 7]
3143 [Thread-9] INFO  backtype.storm.daemon.executor - Finished loading executor spout0:[7 7]
3143 [Thread-26-spout0] INFO  backtype.storm.daemon.executor - Preparing bolt spout0:(7)
3144 [Thread-26-spout0] INFO  backtype.storm.daemon.executor - Prepared bolt spout0:(7)
3147 [Thread-9] INFO  backtype.storm.daemon.executor - Loading executor __system:[-1 -1]
3148 [Thread-9] INFO  backtype.storm.daemon.executor - Loaded executor tasks __system:[-1 -1]
3149 [Thread-9] INFO  backtype.storm.daemon.executor - Finished loading executor __system:[-1 -1]
3149 [Thread-28-__system] INFO  backtype.storm.daemon.executor - Preparing bolt __system:(-1)
3151 [Thread-28-__system] INFO  backtype.storm.daemon.executor - Prepared bolt __system:(-1)
3154 [Thread-9] INFO  backtype.storm.daemon.executor - Loading executor $mastercoord-bg0:[1 1]
3155 [Thread-9] INFO  backtype.storm.daemon.executor - Loaded executor tasks $mastercoord-bg0:[1 1]
3159 [Thread-9] INFO  backtype.storm.daemon.executor - Finished loading executor $mastercoord-bg0:[1 1]
3159 [Thread-30-$mastercoord-bg0] INFO  backtype.storm.daemon.executor - Opening spout $mastercoord-bg0:(1)
3160 [Thread-9] INFO  backtype.storm.daemon.worker - Launching receive-thread for b1bfbee5-be0d-4b1f-b436-ccc230d9ac9e:4
3161 [Thread-30-$mastercoord-bg0] INFO  com.netflix.curator.framework.imps.CuratorFrameworkImpl - Starting
3166 [Thread-9] INFO  backtype.storm.daemon.worker - Worker has topology config {"storm.id" "Count-1-1395506684", "dev.zookee
3167 [Thread-9] INFO  backtype.storm.daemon.worker - Worker b9fd29d4-e9c0-4e84-8026-6a7b22248e5e for storm Count-1-1395506684
3182 [Thread-30-$mastercoord-bg0] INFO  com.netflix.curator.framework.imps.CuratorFrameworkImpl - Starting
3192 [Thread-30-$mastercoord-bg0] INFO  backtype.storm.daemon.executor - Opened spout $mastercoord-bg0:(1)
3194 [Thread-30-$mastercoord-bg0] INFO  backtype.storm.daemon.executor - Activating spout $mastercoord-bg0:(1)
[2]
[1]
[1]
[1]
[2]
[2]
[2]
[4]
[1]
```

The output of the sample Trident topology

The following diagram shows the execution of the sample Trident topology:
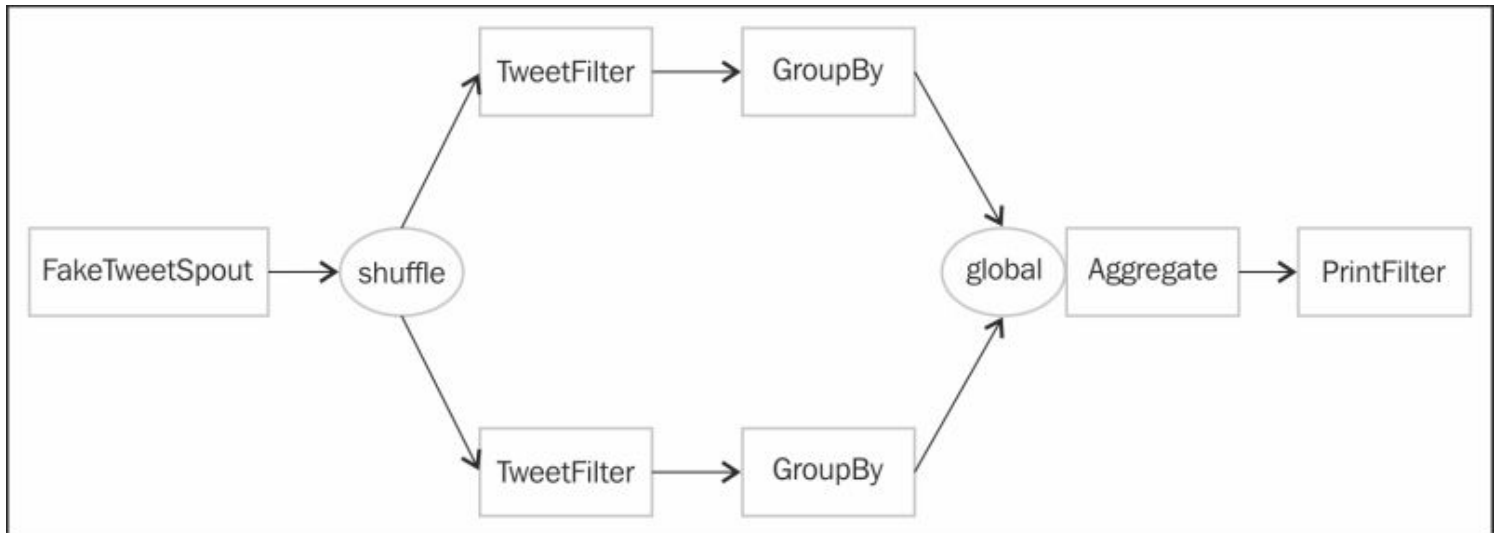
The high-level view of the sample Trident topology

Maintaining the topology state with Trident

Trident provides an abstraction for reading from and writing results to stateful sources. We can maintain the state either internal to the topology (memory) or can store this in external sources (Memcached or Cassandra).

Let's consider that we are maintaining the output of the preceding sample Trident topology in a database. Every time you process a tuple, the count of country present in a tuple increases in the database. However, by maintaining the count in the database, we can't achieve exactly one processing. The reason is that if any tuple fails during processing, then the failed tuple is retried. This creates a problem while updating the state because we are not sure whether the state of this tuple has been updated previously or not. If the tuple has failed before updating the state, then retrying the tuple will make the state consistent. However, if the tuple has failed after updating the state, then retrying the same tuple will again increase the count in the database and make the state inconsistent. Hence, maintaining only the count in the database, an application has no idea whether this tuple is processed earlier or not. You will require more details to make the right decision.

You will need to perform the following steps to achieve the exactly once processing semantics:

1. Process the tuples in small batches.
2. Assign a unique ID to each batch (transactional ID). If the batch is retried, it is given the same unique ID.
3. The state updates are ordered among batches. For example, the state update of the second batch will not be possible until the state update for the first batch has completed.

If we create a topology using the preceding three semantics, then we can easily make a decision whether the tuple is processed earlier or not.

The following section will explain how you can write a transactional topology using Trident.

A transactional topology

As mentioned in the definition of the non-transactional topology, Trident processes tuples in a batch, but this doesn't define what's in each batch. In the case of a transactional topology, a transactional spout guarantees what's in each batch. A transactional spout has the following characteristics:

- Each batch is assigned a unique transactional ID (txid). In the case of failure, the entire batch is replayed. Hence, replays of the failed batch will contain the same set of tuples as the first time the batch was emitted. The txid transactional ID of the failed batch remains the same as the first time.

- Tuples of one batch are not mixed with tuples of another batch. Hence, overlaps of tuples between batches are not allowed.

Let's consider the previous sample Trident topology example and see how we can write a transactional topology. Suppose the sample Trident topology computes the country field's count and stores the counts in a key/value store (Memory Map, Cassandra, Memcached, and so on). The key will be the country's name, and the value will contain the count of the country field so far. As mentioned in the *Maintaining the topology state with Trident* section, just storing a count in the database, we can't guarantee that the tuple is processed earlier or not. You will need to store the transactional ID along with the count to make the decision whether the tuple is processed the first time or already processed. If we are storing the count in the database, then while updating the count in the database, we will first compare the txid parameter of the current tuple with txid already stored in the database. If the txid parameter of the current tuple is greater than the already stored txid, then we will update the database, otherwise, we will escape the tuple without making any change in the database. This entire process works successfully because the transactional spout guarantees that the failed tuples will contain the same set of tuples as the first batch, and state updates are ordered among batches.

For example, we will process txid that is set to 5 and contains the following set of tuples:

[India]
[Japan]
[China]

The current state of the key/value pairs in the database is as follows:

India => [count=7, txid=4]
Japan => [count=10, txid=5]
China => [count=12, txid=4]

As mentioned, the txid parameter of the current batch is 5 and the txid parameter associated with India is 4. The txid parameter of the current batch is greater than the txid parameter of the already stored batch. This means the updates of the current tuple are not present in the database. Hence, we will increment the count by 1 and update the txid parameter from 4 to 5 for India. Similarly, we will increase the count of China and update the txid parameter to 5. On the other hand, the txid parameter of Japan is the same as the txid parameter of the current batch. Hence, we will skip the update of Japan. After performing all the updates, the database will have the following values:

India => [count=8, txid=5]

Japan => [count=10, txid=5]

China => [count=13, txid=5]

In the construction of a transactional topology, a spout plays a key role because it guarantees that the replay tuples will contain the same set of tuples as the first time that the batch was emitted. As we know, the spout reads the data from the external source (Kafka, Twitter, Queue, and so on). Let's consider that a spout is reading data from a distributed queue and emits the batch. If any batch fails, the spout has to read the same set of tuples again from the distributed queue. Now assume that at the same time some nodes of the distributed queue are down. Hence, the spout will not be able to reconstruct the same batch till all the nodes of the distributed queue come up. Hence, the entire pipeline has nothing to process at that time. This concludes that the transactional spouts are not very fault tolerant if the input data source (distributed queue) is not fault tolerant. The data sources, such as Kafka (as of Version 0.8), do guarantee fault tolerance via their partition replication feature; so, using Kafka 0.8 with a transactional topology does give a good fault tolerance.

You can download the implementation of the transaction spout for Kafka from https://github.com/nathanmarz/storm-contrib/blob/master/storm-kafka/src/jvm/storm/kafka/trident/TransactionalTridentKafkaSpout.java.

The following section covers the overview of the opaque transactional topology.

The opaque transactional topology

The opaque transactional topology has overcome the limitation of the transactional topology, and the opaque transactional spout is fault tolerant even if the data source nodes are down. The opaque transactional spout has the following characteristics:

- Every tuple is processed in exactly one batch.
- If a tuple is not processed in one batch, it would be processed in the next batch. But, the second batch doesn't have the same set of tuples as the first processed batch.

In the case of a transactional topology, we would maintain both the txid and count parameters to make the decision whether the tuple was processed earlier or not. On the other hand, in the case of an opaque transactional topology, we would need to store the txid, count, and previous count parameters to maintain the consistency of the database.

For example, we are processing a txid 5 which contains the following set of tuples:

[India]

[India]
[Japan]
[China]

The current state of the key/value in the database is as follows:

India => [count=7, txid=4, previous=5]
Japan => [count=10, txid=4, previous=9]
China => [count=12, txid=4, previous=10]

The current txid parameter is 5, which is greater than the stored txid. Hence, the fifth batch was not processed earlier. The stored value of count is copied to the previous values of count, the value of count for India is incremented by 2, and the txid parameter is updated from 4 to 5. Similarly, we will increase the count value of Japan and China. After processing the txid value to 5, the state of the database will look like the following tuples:

India => [count=9, txid=5, previous=7]
Japan => [count=11, txid=5, previous=10]
China => [count=13, txid=5, previous=12]

Distributed RPC

Distributed RPC is used to query on and retrieve the result from the Trident topology on the fly. Storm has an in-built distributed RPC server. The distributed RPC server receives the RPC request from the client and passes it to the topology. The topology processes the request and sends the result to the distributed RPC server, which is redirected by the distributed RPC server to the client.

We can configure the distributed RPC server by setting the following properties in the storm.yaml file:

drpc.servers:
   - "nimbus-node"

Here, nimbus-node is the IP address of the distributed RPC server.

Now, run the following command on the nimbus-node machine to start the distributed RPC server:

bin/storm drpc

Let's consider that we are storing the count aggregation of the sample Trident topology in the database and want to retrieve the count for the given country on the fly. Then, we will need to use the distributed RPC feature to achieve this. The following example code shows how we can incorporate the distributed RPC server in the sample Trident topology created in the previous section. We will create a DistributedRPC class that contains the buildTopology() method, as mentioned in the following code:

```java
public class DistributedRPC {

    public static void main(String[] args) throws Exception {
        Config conf = new Config();
        conf.setMaxSpoutPending(20);
        LocalDRPC drpc = new LocalDRPC();
        if (args.length == 0) {

            LocalCluster cluster = new LocalCluster();
            cluster.submitTopology("CountryCount", conf, buildTopology(drpc));
            Thread.sleep(2000);
            for(int i=0; i<100; i++) {
                System.out.println(drpc.execute("Count", "Japan,India,Europe"));
                Thread.sleep(1000);
            }
        } else {
            conf.setNumWorkers(3);
            StormSubmitter.submitTopology(args[0], conf, buildTopology(null));
            Thread.sleep(2000);
            DRPCClient client = new DRPCClient("RRPC-Server", 1234);
            System.out.println(client.execute("Count", "Japan,India,Europe"));
        }
    }

    public static StormTopology buildTopology(LocalDRPC drpc) {

        FakeTweetSpout spout = new FakeTweetSpout(10);
        TridentTopology topology = new TridentTopology();
        TridentState countryCount = topology.newStream("spout1", spout).shuffle().each(new
Fields("text","Country"), new TridentUtility.TweetFilter()).groupBy(new
Fields("Country")).persistentAggregate(new MemoryMapState.Factory(),new Fields("Country"),
new Count(), new Fields("count")).parallelismHint(2);
```

```
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
    }

    topology.newDRPCStream("Count", drpc).each(new Fields("args"), new
TridentUtility.Split(), new Fields("Country")).stateQuery(countryCount, new Fields("Country"),
new MapGet(), new Fields("count")).each(new Fields("count"), new FilterNull());

    return topology.build();
  }
}
```

Let's understand the code line by line. We are using the FakeTweetSpout class as an input source and the TridentTopology class to define the Trident computation.

In the next line, we are using the persistentAggregate function, which will store the count aggregation of all the batches ever emitted to the Trident state. We are using the MemoryMapState.Factory() method to maintain the count state. The persistentAggregate function knows how to store and update the aggregation in the source state:

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

The memory mapstate is an in-memory Java map and stores the country's name as the key and the aggregation count as the value, as shown in the following tuples:

India -> 124
United State -> 145
Japan -> 130
Brazil -> 155
China -> 100

The persistentAggregate function transforms the stream into the TridentState object. In this case, the countryCount variable represents the count of each country so far.

The next part of the topology defines a distributed query to get the count of each country on the fly. The distributed RPC query takes the comma-separated list of countries as input and returns

the count for each country. The following is the piece of code that defines the distributed query portion:

```
topology.newDRPCStream("Count", drpc).each(new Fields("args"), new TridentUtility.Split(),
new Fields("Country")).stateQuery(countryCount, new Fields("Country"), new MapGet(), new
Fields("count")).each(new Fields("count"), new FilterNull());
```

The Split function is used to split the comma-separated list of countries. We have used a stateQuery() method to query the TridentState object, which is defined in the first part of the topology. The stateQuery() method takes in a source of the state, in this case, the countries count computed by the first part of the topology, and a function to query that state. We are using a MapGet() function, which gets the count for each country. Finally, the count of each country is returned as the query output.

The following piece of code shows how we can pass input to a local distributed RPC:

```
System.out.println(drpc.execute("Count", "Japan,India,Europe"));
```

To run the topology on the local mode, we have created an instance of the backtype.storm.LocalDRPC class to simulate the distributed RPC server.

If you are running the distributed RPC server, then we would need to create an instance of the distributed RPC client to execute the query. The following piece of code shows how we can pass the input to the distributed RPC server:

```
DRPCClient client = new DRPCClient("RRPC-Server", 1234);
System.out.println(client.execute("Count", "Japan,India,Europe"));
```

Trident's distributed RPC query executes like the normal RPC query, except these queries are run in parallel. The following screenshot is of the console output of the DistributedRPC class:

```
4895 [Thread-7] INFO  backtype.storm.daemon.worker - Worker 27b04604-09c0-4153-b20d-7794f304a008 for storm CountryCount-
1-1397038903 on de777e4a-caea-435d-b9a3-697673008979:1 has finished loading
4916 [Thread-41-$mastercoord-bg0] INFO  com.netflix.curator.framework.imps.CuratorFrameworkImpl - Starting
4926 [Thread-41-$mastercoord-bg0] INFO  backtype.storm.daemon.executor - Opened spout $mastercoord-bg0:(1)
4926 [Thread-41-$mastercoord-bg0] INFO  backtype.storm.daemon.executor - Activating spout $mastercoord-bg0:(1)
[["Japan,India,Europe","Japan",39]]
[["Japan,India,Europe","Japan",63]]
[["Japan,India,Europe","India",85]]
[["Japan,India,Europe","Japan",121]]
[["Japan,India,Europe","India",133]]
[["Japan,India,Europe","India",169]]
[["Japan,India,Europe","India",198]]
[["Japan,India,Europe","Japan",227]]
[["Japan,India,Europe","India",250]]
[["Japan,India,Europe","India",280]]
[["Japan,India,Europe","India",300]]
[["Japan,India,Europe","India",324]]
[["Japan,India,Europe","Japan",363]]
[["Japan,India,Europe","Japan",397]]
```

Output of the distributed RPC topology

When to use Trident

As in many use cases, we have required exactly one processing, which we can achieve by writing a transactional topology in Trident. On the other hand, it will be difficult to achieve exactly one processing in the case of Vanilla Storm. Hence, Trident will be useful for those use cases where we require exactly once processing.

Trident is not fit for all use cases, especially high-performance use cases, because Trident adds complexity on Storm and manages the state.

Summary

In this chapter, we mainly concentrated on high-level abstraction over Storm with Trident and learned about Trident filters, functions, aggregators, repartitioning operations, and the non-transactional topology. We also walked through how we can define a Trident topology. We also covered how we can query on the fly on the Trident topology using distributed RPC.

In the next chapter, we will explain how we can combine batch processing and real-time processing tools to solve real-world use cases.

Chapter 6. Integration of Storm with Batch Processing Tools

So far, we have seen how Storm can be used to develop real-time stream processing applications. In general, these real-time applications are seldom used in isolation. They are more often than not used in combination with other batch processing operations.

The most common platform to develop batch jobs is Apache Hadoop. In this chapter, we will see how applications built with Apache Storm can be deployed over existing Hadoop clusters with the help of the Storm-YARN framework for optimized use and management of resources.

In this chapter, we will cover the following topics:

- An overview of Apache Hadoop and its various components
- Setting up a Hadoop cluster
- An overview of Storm-YARN
- Deploying Storm-YARN on Hadoop
- Running a Storm application on Storm-YARN

Exploring Apache Hadoop

Apache Hadoop is an open source platform to develop and deploy Big Data applications. It was initially developed at Yahoo! based on the MapReduce and Google File System papers published by Google. Over the past few years, Hadoop has become the flagship Big Data platform.

The following are the key components of a Hadoop cluster:

- **Hadoop Distributed File System** (**HDFS**)
- **Yet Another Resource Negotiator** (**YARN**)

Both HDFS and YARN are based on a set of libraries called **Hadoop Common**. It provides an abstraction for OS and filesystem operations so that Hadoop can be deployed on a variety of platforms. Now let's have a deeper look into HDFS and YARN.

Understanding HDFS

Commonly known as HDFS, Hadoop Distributed File System is a scalable, distributed, fault-tolerant filesystem. HDFS acts as the storage layer of the Hadoop ecosystem. It allows sharing and storage of data and application code among the various nodes in a Hadoop cluster.

The following were the key assumptions made while designing HDFS:

- It should be deployable on a cluster of commodity hardware.
- Hardware failures are expected, and it should be tolerant to those.
- It should be scalable to thousands of nodes.
- It should be optimized for high throughput, even at the cost of latency.
- Most of the files will be large in size, so optimize for big files.
- Storage is cheap, so use replication for reliability.
- It should be locality aware so that the computations requested on data can be performed on the physical node where it actually resides. This will result in less data movement, and hence lower network congestion.

The following diagram illustrates the key components of an HDFS cluster and the ways in which they interact with each other:
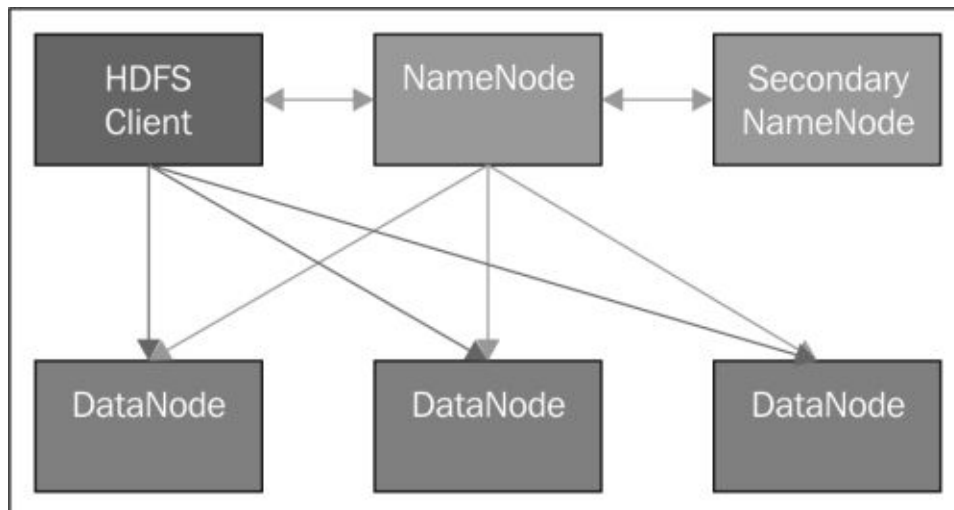
An HDFS cluster

Now, let's have a detailed look at each of the HDFS components:

- **NameNode**: This is the master node in an HDFS cluster. It is responsible for managing the filesystem metadata and operations. It does not store any user data but only the filesystem tree of all files in the cluster. It also keeps track of the physical locations of the blocks that are part of a file.

  Since the NameNode keeps all the data in RAM, it should be deployed on a machine with a large amount of RAM. Also, no other processes should be hosted on the machine hosting the NameNode so that all the resources are dedicated to it.

  The NameNode is the single point of failure in an HDFS cluster. If the NameNode dies, no operations can take place on an HDFS cluster.

- **DataNode**: This is responsible for storing user data in an HDFS cluster. There can be multiple DataNodes in an HDFS cluster. A DataNode stores data on the physical disks attached to the system hosting the DataNode. It is not recommended to store DataNode data on disks in the RAID configuration as HDFS achieves data protection by replicating data across DataNodes.
- **An HDFS client**: An HDFS client is a client library that can be used to interact with an HDFS cluster. It usually talks to the NameNode to do meta operations, such as creating new files, deleting files, and so on, while the DataNodes serve the actual data read and write requests.
- **Secondary NameNode**: This is one of the poorly named components of HDFS. Despite its name, it is not a standby for the NameNode. To understand its function, we need to delve deep into how the NameNode works.

  The NameNode keeps the filesystem metadata in the main memory. For durability, it also writes this metadata to a local disk in the form of the fsimage file. When a NameNode starts, it reads this fsimage snapshot file to recreate the in-memory data structure to hold filesystem data. Any updates on the filesystem are applied to the in-memory data structure but not to the fsimage file. These changes are written to the disk in separate files called **edit logs**. When a NameNode starts, it merges these edit logs into the fsimage file so that the next restart will be quick. In production, the edit logs can grow very large as the NameNode is not restarted frequently. This could result in a very long boot time for the NameNode whenever it is restarted.

  The Secondary NameNode is responsible for merging the edit logs of the NameNode with the fsimage file so that the NameNode starts faster the next time. It takes the fsimage snapshot file and edit logs from the NameNode, merges them, and then puts the updated fsimage snapshot file on the NameNode machine. This process runs periodically and

reduces the amount of merging that is required by a NameNode on restarts, thus reducing the time to boot for the NameNode.

The following diagram illustrates the working of the Secondary NameNode:
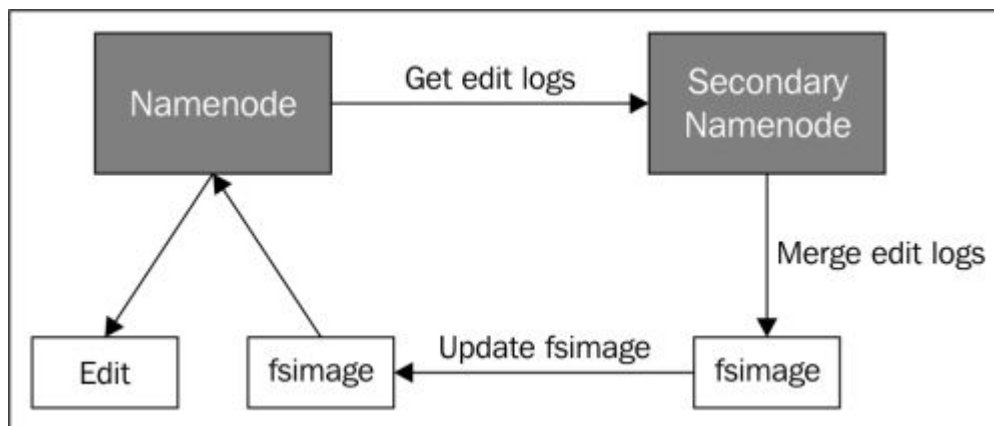
The Secondary NameNode's functioning

So far, we have seen the storage side of Hadoop; next, we will look into the processing components.

Understanding YARN

Yet Another Resource Negotiator (**YARN**) is a cluster resource management framework that enables users to submit a variety of jobs to a Hadoop cluster and manages the scalability, fault tolerance, and scheduling of jobs. As HDFS provides a storage layer for large amounts of data, the YARN framework gives you the plumbing required to write Big Data processing applications.

The following are the major components of a YARN cluster:

- **ResourceManager** (**RM**): This is the entry point for applications in the YARN cluster. It is the master process in the cluster that is responsible for managing all the resources in the cluster. It is also responsible for the scheduling of various jobs submitted to the cluster. This scheduling policy is pluggable and can be customized by users if they want to support new kinds of applications.
- **NodeManager** (**NM**): A NodeManager agent is deployed on each of the processing nodes on the cluster. It is the counterpart to the ResourceManager on the node level. It communicates with the RM to update the node state and to receive any job requests from it. It is also responsible for the life cycle management and reporting of various node metrics to the RM.

- **ApplicationMaster** (**AM**): Once a job is scheduled by the RM, it no longer keeps track of the job's status and progress. This results in the fact that a ResourceManager can support a completely different kind of application in the cluster without worrying about the internal communication and logic of the application.

  Whenever an application is submitted, the RM creates a new ApplicationMaster for that application, which is then responsible for negotiating resources from the RM and communicating with the NM for the resources. The NM gives resources in the form of resource containers that are abstractions of resource allocation, where you can tell how much CPU, memory, and so on are required.

  Once the application starts running on various nodes in the cluster, the AM keeps track of the status of various jobs and in the event of failures, reruns those jobs. On completion of the job, it releases the resources to the RM.

  The following diagram illustrates the various components in a YARN cluster:
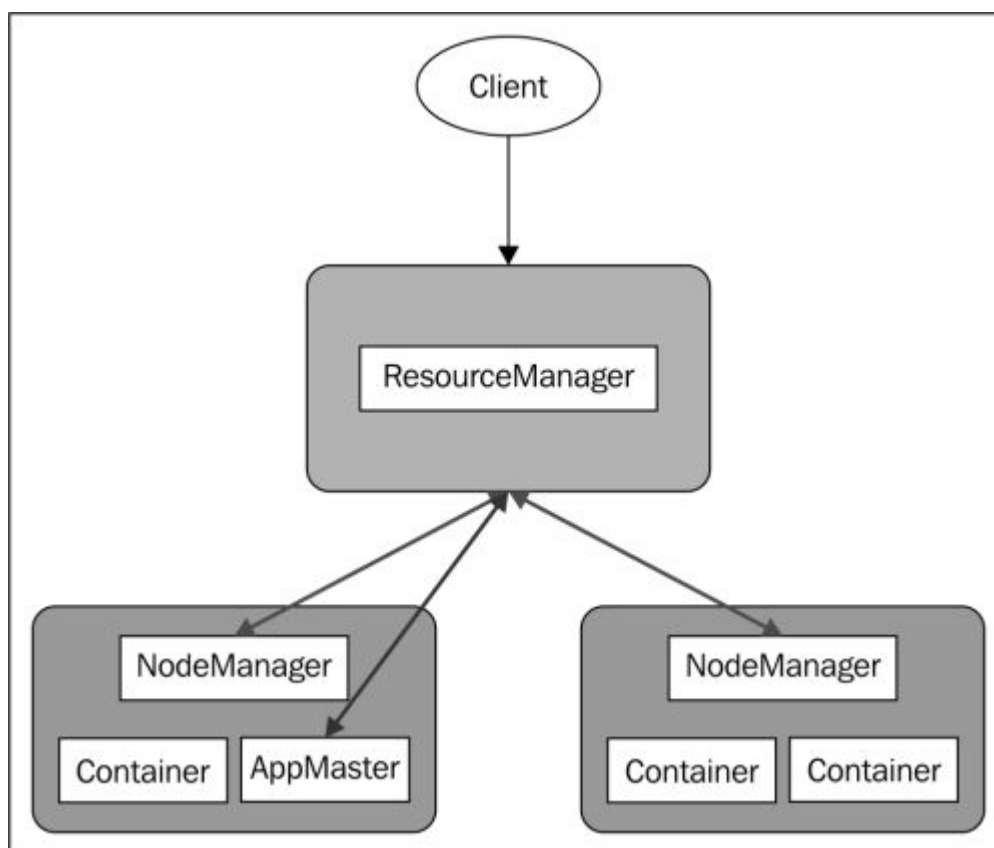
YARN components

Installing Apache Hadoop

Now that we have seen both the storage and processing parts of a Hadoop cluster, let's get started with the installation of Hadoop. We will use Hadoop 2.2.0 in this chapter.

Note

Hadoop 2.2.0 is not compatible with Hadoop 1.X versions.

We will be setting up a cluster on a single node. Before starting, please make sure that you have the following software installed on your system:

- **JDK 1.7**: We need JDK to run Hadoop as it is written in Java
- **ssh-keygen**: This is used to generate SSH keys that are used to set password-less SSH required for Hadoop

If you don't have ssh-keygen, install it with the following command:

yum install openssh-clients

Next, we will need to set up password-less SSH on this machine as it is required for Hadoop.

Setting up password-less SSH

In a Hadoop cluster, executing commands on one of the machines in turn can execute further commands on some nodes in the cluster. For example, when starting HDFS, the DataNode is started on each of the machines. This is done automatically by the scripts provided with your Hadoop distribution. Password-less SSH between all the machines in a Hadoop cluster is a mandatory requirement for these scripts to run without any user intervention. The following are the steps for setting up password-less SSH:

1. Generate your ssh key pair by executing the following command:
2. **ssh-keygen -t rsa -P "**

   The following information is displayed:

   Generating public/private rsa key pair.
   Enter file in which to save the key (/home/anand/.ssh/id_rsa):
   Your identification has been saved in /home/anand/.ssh/id_rsa.
   Your public key has been saved in /home/anand/.ssh/id_rsa.pub.
   The key fingerprint is:

b7:06:2d:76:ed:df:f9:1d:7e:5f:ed:88:93:54:0f:24anand@localhost.localdoma in

The key's randomart image is:

```
+--[ RSA 2048]----+
|                 |
|            E .  |
|             o   |
|        . .  o   |
|       S + ..o   |
|      . = o.   o |
|         o... .o |
|      .  oo.+*   |
|          ..ooX  |
+-----------------+
```

3. Next, we need to copy the generated public key to the list of authorized keys for the current user. To do this, execute the following command:

4. **cp ~/.ssh/id_rsa.pub ~/.ssh/authorized_keys**

5. Now, we can check whether password-less SSH is working by connecting to localhost with ssh using the following command:

6. **ssh localhost**

   The following output is displayed:

   Last login: Wed Apr 2 09:12:17 2014 from localhost

Since we are able to SSH into localhost without a password, our setup is working now, and we will now proceed with the Hadoop setup.

Getting the Hadoop bundle and setting up environment variables

The following are the steps to set up Hadoop:

1. Download Hadoop 2.2.0 from the Apache website at http://hadoop.apache.org/releases.html#Download.

2. Untar the archive at a location where you want to install Hadoop using the following commands. We will call this location $HADOOP_HOME:

3. **tar xzf hadoop-2.2.0.tar.gz**

4. **cd hadoop-2.2.0**

5. Next, we need to set up the environment variables and path for Hadoop. Add the following entries in your ~/.bashrc file. Please make sure that you provide the paths for Java and Hadoop as per your system using the following commands:

6. **export JAVA_HOME=/usr/java/jdk1.7.0_45**

7. **export HADOOP_HOME=/home/anand/opt/hadoop-2.2.0**

8. **export HADOOP_COMMON_HOME=/home/anand/opt/hadoop-2.2.0**

9. **export HADOOP_HDFS_HOME=$HADOOP_COMMON_HOME**

10. **export HADOOP_MAPRED_HOME=$HADOOP_COMMON_HOME**

11. **export HADOOP_YARN_HOME=$HADOOP_COMMON_HOME**

12. **export HADOOP_CONF_DIR=$HADOOP_COMMON_HOME/etc/hadoop**

13. **export HADOOP_COMMON_LIB_NATIVE_DIR=$HADOOP_COMMON_HOME/lib /native**

14. **export HADOOP_OPTS="-Djava.library.path=$HADOOP_COMMON_HOME/lib"**

15. **export PATH=$PATH:$JAVA_HOME/bin:$HADOOP_COMMON_HOME/bin:$HADOOP_COMMON_HOME/sbin**

16. Refresh your ~/.bashrc file with the following command:

17. **source ~/.bashrc**

18. Now, let's check whether the paths are properly configured with the following command:

19. **hadoop version**

The following information is displayed:

Hadoop 2.2.0
Subversion https://svn.apache.org/repos/asf/hadoop/common -r 1529768
Compiled by hortonmu on 2013-10-07T06:28Z
Compiled with protoc 2.5.0
From source with checksum 79e53ce7994d1628b240f09af91e1af4
This command was run using /home/anand/opt/hadoop-2.2.0/share/hadoop/common/hadoop-common-2.2.0.jar

Using the preceding steps, the paths are properly set. Now we will set up HDFS on our system.

Setting up HDFS

Please perform the following steps to set up HDFS:

1. Make directories to hold the NameNode and DataNode data as follows:
2. **mkdir -p ~/mydata/hdfs/namenode**
3. **mkdir -p ~/mydata/hdfs/datanode**
4. Specify the NameNode port in the core-site.xml file at the $HADOOP_CONF_DIR directory by adding the following property inside the <configuration> tag:
5.   <property>
6.   <name>fs.default.name</name>
7.   <value>hdfs://localhost:19000</value>
8.   <!-- The default port for HDFS is 9000, but we are using 19000 Storm-Yarn uses port 9000 for its application master -->
</property>

9. Specify the NameNode and data directory in the hdfs-site.xml file at $HADOOP_CONF_DIR by adding the following property inside the <configuration> tag:
10.   <property>
11.   <name>dfs.replication</name>
12.   <value>1</value>
13.   <!-- Since we have only one node, we have replication factor=1 -->
14.   </property>
15.   <property>
16.   <name>dfs.namenode.name.dir</name>
17.   <value>file:/home/anand/hadoop-data/hdfs/namenode</value>
18.   <!-- specify absolute path of the namenode directory -->
19.   </property>
20.   <property>
21.   <name>dfs.datanode.data.dir</name>
22.   <value>file:/home/anand/hadoop-data/hdfs/datanode</value>
23.   <!-- specify absolute path of the datanode directory -->
  </property>

24. Now, we will format the NameNode. This is a one-time process and needs to be done only while setting up HDFS using the following command:
25. **hdfs namenode -format**

The following output is displayed:

14/04/02 09:03:06 INFO namenode.NameNode: STARTUP_MSG:
/***

STARTUP_MSG: Starting NameNode

STARTUP_MSG: host = localhost.localdomain/127.0.0.1

STARTUP_MSG: args = [-format]

STARTUP_MSG: version = 2.2.0

... ...

14/04/02 09:03:08 INFO namenode.NameNode: SHUTDOWN_MSG:

/**

SHUTDOWN_MSG: Shutting   down   NameNode   at
localhost.localdomain/127.0.0.1

**/

26. Now, we are done with the configuration and will start HDFS with the following command:
27. **start-dfs.sh**

    The following information is displayed:

    **14/04/02 09:27:13 WARN util.NativeCodeLoader: Unable to load
    native-hadoop library for your platform... using builtin-java classes where
    applicable**
    Starting namenodes on [localhost]
    **localhost: starting namenode, logging to
    /home/anand/opt/hadoop-2.2.0/logs/hadoop-anand-namenode-localhost.lo
    caldomain.out**
    **localhost: starting datanode, logging to
    /home/anand/opt/hadoop-2.2.0/logs/hadoop-anand-datanode-localhost.loc
    aldomain.out**
    Starting secondary namenodes [0.0.0.0]
    **0.0.0.0: starting secondarynamenode, logging to
    /home/anand/opt/hadoop-2.2.0/logs/hadoop-anand-secondarynamenode-l
    ocalhost.localdomain.out**
    **14/04/02 09:27:32 WARN util.NativeCodeLoader: Unable to load
    native-hadoop library for your platform... using builtin-java classes where
    applicable**

28. Now, execute the jps command to see whether all the processes are running fine:
29. **jps**

    We will get the following output:

    50275 NameNode

50547 SecondaryNameNode

50394 DataNode

51091 Jps

Here, we can see that all the expected processes are running.

30. Now, you can check the status of HDFS using the NameNode Web UI by opening http://localhost:50070 in your browser. You should see something similar to the following screenshot:

The NameNode Web UI

31. You can interact with HDFS using the hdfs dfs command. Get all the options by running the hdfs dfs command on the console or refer to the documentation at http://hadoop.apache.org/docs/r2.2.0/hadoop-project-dist/hadoop-common/FileSystemShell.html. Most of the commands mirror the filesystem interaction commands that you'll find on any Linux system. For example, to copy a file on HDFS, use the following command:

32. **hdfs dfs -cp /user/hadoop/file1 /user/hadoop/file2**

Now that HDFS is deployed, we will set up YARN next.

Setting up YARN

The following are the steps to set up YARN:

1. Create the mapred-site.xml file from mapred-site.xml.template using the following command:
2. **cp $HADOOP_CONF_DIR/mapred-site.xml.template $HADOOP_CONF_DIR/mapred-site.xml**
3. Specify that we are using the YARN framework by adding the following property in the mapred-site.xml file located in the $HADOOP_CONF_DIR directory in the <configuration> tag:
4. <property>
5.   <name>mapreduce.framework.name</name>
6.   <value>yarn</value>

</property>

7. Configure the following properties in the yarn-site.xml file:
8. <property>
9.   <name>yarn.nodemanager.aux-services</name>
10.   <value>mapreduce_shuffle</value>
11. </property>
12.
13. <property>
14.   <!-- Minimum amount of memory allocated for containers in MBs.-->
15.   <name>yarn.scheduler.minimum-allocation-mb</name>
16.   <value>1024</value>
17. </property>
18.
19. <property>
20.   <!--Total memory that can be allocated to containers in MBs. -->
21.   <name>yarn.nodemanager.resource.memory-mb</name>
22.   <value>4096</value>
23. </property>
24.
25. <property>
26.   <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
27.   <value>org.apache.hadoop.mapred.ShuffleHandler</value>
28. </property>

29.   <property>

30.   <!-- This is ratio of physical memory to virtual memory used when setting memory requirements for containers. If you don't have enough RAM, increase this value. -->

31.   <name>yarn.nodemanager.vmem-pmem-ratio</name>

32.   <value>8</value>

</property>

33. Start the YARN processes with the following command:

34. **start-yarn.sh**

The following information is displayed:

starting yarn daemons
starting( )resourcemanager, logging to
/home/anand/opt/hadoop-2.2.0/logs/yarn-anand-resourcemanager-localho
st.localdomain.out
localhost: starting nodemanager, logging to
/home/anand/opt/hadoop-2.2.0/logs/yarn-anand-nodemanager-localhost.lo
caldomain.out

35. Now, execute the jps command to see whether all the processes are running fine:

36. **jps**

We will get the following output:

50275 NameNode
50547 SecondaryNameNode
50394 DataNode
51091 Jps
50813 NodeManager
50716 ResourceManager

Here, we can see that all the expected processes are running.

37. Now, you can check the status of YARN using the ResourceManager Web UI by opening http://localhost:8088/cluster in your browser. You should see something similar to the following screenshot:
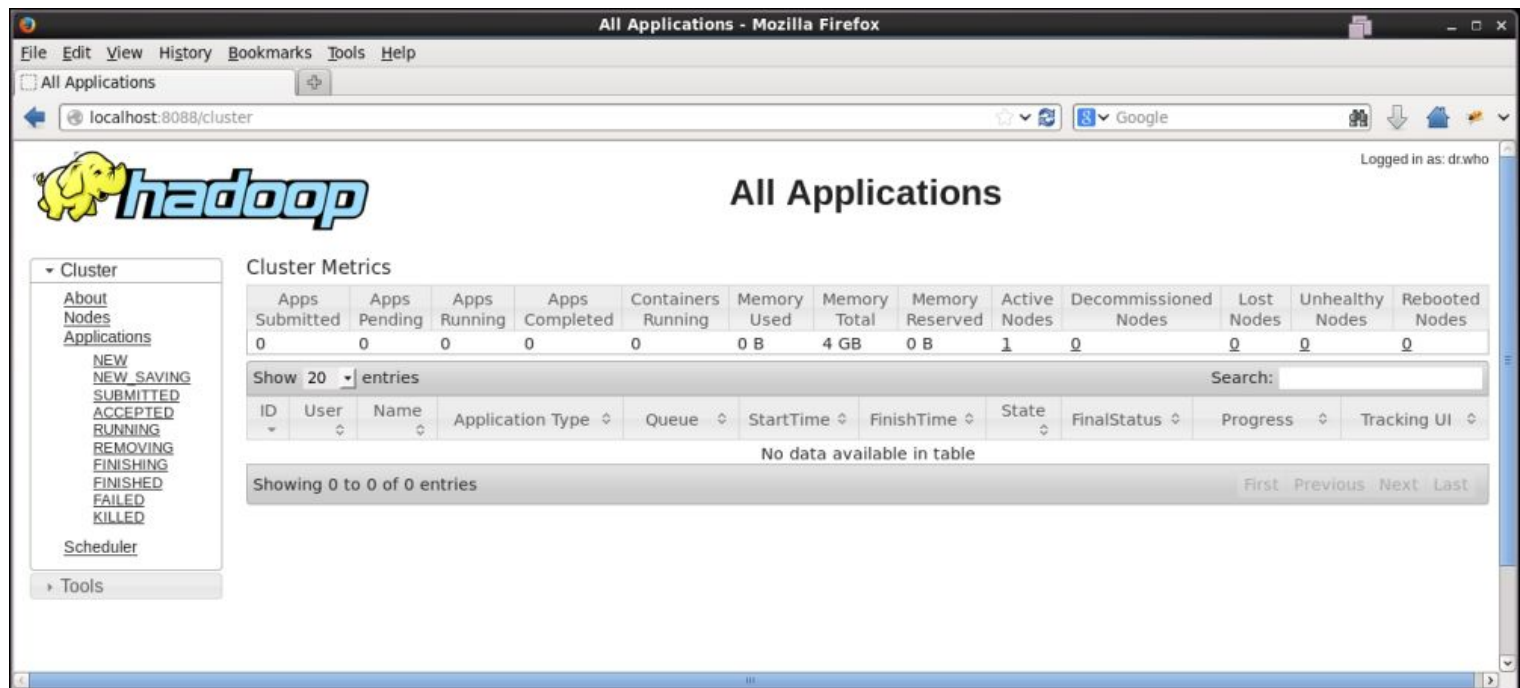
The ResourceManager Web UI

38. You can interact with YARN using the yarn command. Get all the options by running the yarn command on your console, or refer to the documentation at http://hadoop.apache.org/docs/r2.2.0/hadoop-yarn/hadoop-yarn-site/YarnCommands.html. To get all the applications currently running on YARN, run the following command:

39. **yarn application -list**

The following information is displayed:

14/04/02 11:41:42 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
14/04/02 11:41:42 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
Total number of applications (application-types: [] and states: [SUBMITTED, ACCEPTED, RUNNING]):0
Application-Id   Application-Name   Application-Type   User   Queue   State   Final-State   Progress   Tracking-URL

With this, we have completed the deployment of a Hadoop cluster on a single node. Next, we will see how to run Storm topologies on this cluster.

Integration of Storm with Hadoop

The probability that the organizations developing and operating Big Data applications already have a Hadoop cluster deployed is very high. Also, there is a high possibility that they also have real-time stream processing applications deployed to go along with the batch applications running on Hadoop.

It would be great if we can leverage the already deployed YARN cluster to also run Storm topologies. This will reduce the operational cost of maintenance by giving us only one cluster to manage instead of two.

Storm-YARN is a project developed by Yahoo! that enables the deployment of Storm topologies over YARN clusters. It enables the deployment of Storm processes on nodes managed by YARN.

The following diagram illustrates how the Storm processes are deployed on YARN:
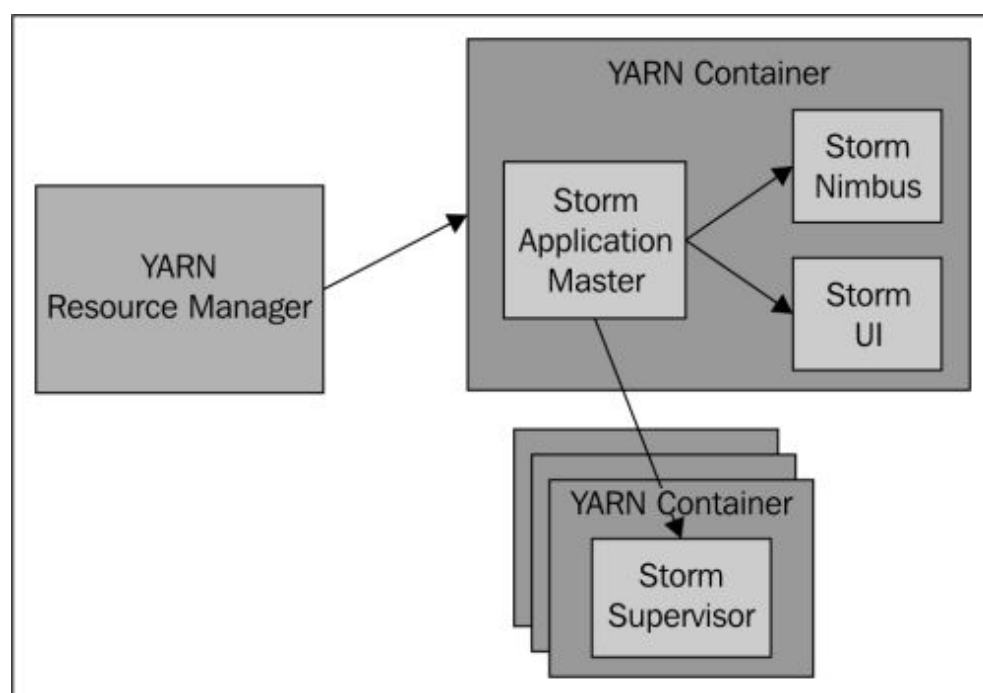
Storm processes on YARN

In the following section, we will see how to set up Storm-YARN.

Setting up Storm-YARN

Since Storm-YARN is still in alpha, we will proceed with the master branch of the Git repository. The master branch is the branch where all the development for Git repositories takes place. It is

equivalent to the trunk in SVN repositories. Make sure you have Git installed on your system. If not, then run the following command:

yum install git-core

Also make sure that you have Apache ZooKeeper and Apache Maven installed on your system. Refer to previous chapters for their setup instructions.

The following are the steps for deploying Storm-YARN:

1. Clone the Storm-YARN repository with the following commands:
2. **cd ~/opt**
3. **git clone https://github.com/yahoo/storm-yarn.git**
4. **cd storm-yarn**
5. Build Storm-YARN by running the following Maven command:
6. **mvn package**

    We will get the following output:

    [INFO] Scanning for projects...
    [INFO]
    [INFO] --
    [INFO] Building storm-yarn 1.0-alpha
    [INFO] --
    ...
    [INFO] --
    [INFO] BUILD SUCCESS
    [INFO] --
    [INFO] Total time: 32.049s
    [INFO] Finished at: Fri Apr 04 09:45:06 IST 2014
    [INFO] Final Memory: 14M/152M
    [INFO] --

7. Copy the storm.zip file from storm-yarn/lib to HDFS using the following commands:
8. **hdfs dfs -mkdir -p   /lib/storm/0.9.0-wip21**
9. **hdfs dfs -put lib/storm.zip /lib/storm/0.9.0-wip21/storm.zip**

    The exact version might be different in your case from 0.9.0-wip21.

10. Create a directory to hold our Storm configuration:
11. **mkdir -p ~/storm-data**

12. **cp lib/storm.zip ~/storm-data/**
13. **cd ~/storm-data/**
14. **unzip storm.zip**
15. Add the following configuration in the storm.yaml file located at ~/storm-data/storm-0.9.0-wip21/conf:
16.   storm.zookeeper.servers:
17.   - "localhost"
18.
19.   nimbus.host: "localhost"
20.
21.   master.initial-num-supervisors: 2

    master.container.size-mb: 128

    If required, change the values as per your setup.

22. Add the storm-yarn/bin folder to your path by adding the following code to the ~/.bashrc file:

    export PATH=$PATH:/home/anand/storm-data/storm-0.9.0-wip21/bin:/home/anand/opt/storm-yarn/bin

23. Refresh the ~/.bashrc file with the following command:
24. **source ~/.bashrc**
25. Make sure ZooKeeper is running on your system. If not, start ZooKeeper by running the following command:
26. **~/opt/zookeeper-3.4.5/bin/zkServer.sh start**
27. Launch Storm-YARN using the following command:
28. **storm-yarn launch ~/storm-data/storm-0.9.0-wip21/conf/storm.yaml**

    We will get the following output:

    14/04/15 10:14:49 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
    14/04/15 10:14:49 INFO yarn.StormOnYarn: Copy App Master jar from local filesystem and add to local environment
    ... ...
    14/04/15 10:14:51 INFO impl.YarnClientImpl: Submitted application application_1397537047058_0001 to ResourceManager at /0.0.0.0:8032
    application_1397537047058_0001

The Storm-YARN application has been submitted with the
application_1397537047058_0001 application ID.

29. We can retrieve the status of our application using the following yarn command:
30. **yarn application -list**

We will get the status of our application as follows:

14/04/15 10:23:13 INFO client.RMProxy: Connecting to ResourceManager at
/0.0.0.0:8032
Total number of applications (application-types: [] and states: [SUBMITTED,
ACCEPTED, RUNNING]):1
        Application-Id Application-Name Application-Type
User Queue   State   Final-State   Progress
Tracking-URL
application_1397537047058_0001 Storm-on-Yarn
YARN and default   RUNNING   UNDEFINED
50%   N/A

31. We can also see Storm-YARN running on the ResourceManager Web UI at
http://localhost:8088/cluster/. You should be able to see something similar to the
following screenshot:

Storm-YARN on the ResourceManager Web UI

You can explore the various metrics exposed by clicking on various links on the UI.

32. Nimbus should also be running now, and you should be able to see it through the Nimbus Web UI at http://localhost:7070/. You should be able to see something similar to the following screenshot:
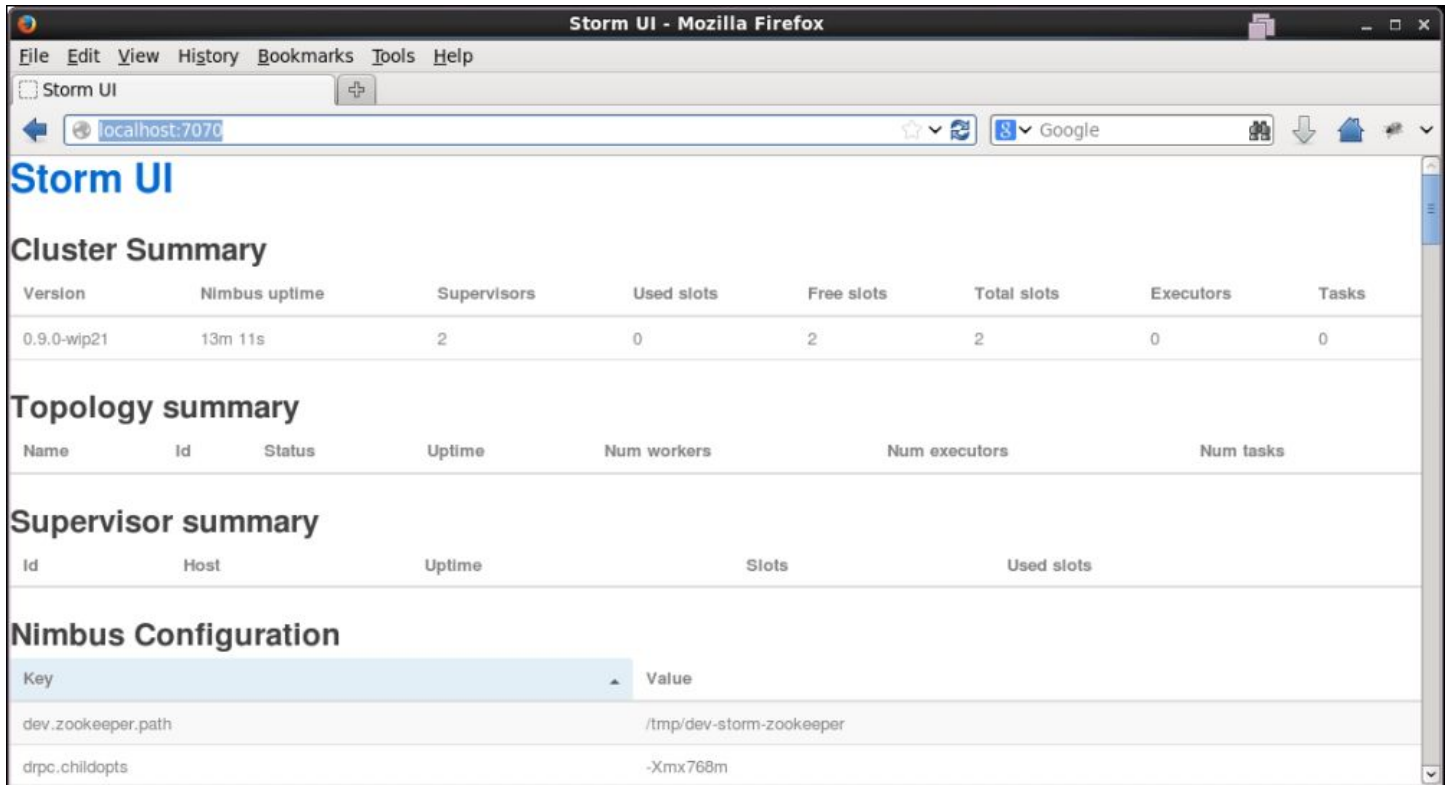
The Nimbus Web UI running on YARN

33. Now, we need to get the Storm configuration that will be used when deploying topologies on this Storm cluster deployed over YARN. To do so, execute the following commands:

34. **mkdir ~/.storm**

35. **storm-yarn getStormConfig --appId application_1397537047058_0001 --output ~/.storm/storm.yaml**

We will get the following output:

14/04/15 10:32:01 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
14/04/15 10:32:02 INFO yarn.StormOnYarn: application report for application_1397537047058_0001 :localhost.localdomain:9000
14/04/15 10:32:02 INFO yarn.StormOnYarn: Attaching to localhost.localdomain:9000 to talk to app master application_1397537047058_0001

14/04/15 10:32:02 INFO yarn.StormMasterCommand: storm.yaml downloaded into /home/anand/.storm/storm.yaml

36. Please make sure that you are passing the correct application ID as retrieved in step 9 to the –appId parameter.

Now that we have successfully deployed Storm-YARN, we will see how to run our topologies on this Storm cluster.

Deploying Storm-Starter topologies on Storm-YARN

In this section, we will see how to deploy Storm-Starter topologies on Storm-YARN. Storm-Starter is a set of example topologies that comes with Storm. Perform the following steps to run the topologies on Storm-YARN:

1. Clone the Storm-Starter project with the following commands:
2. **git clone https://github.com/nathanmarz/storm-starter**
3. **cd storm-starter**
4. Package the topologies with the following mvn command:
5. **mvn package –DskipTests**
6. Deploy WordCountTopology on Storm-YARN with the following command:
7. **storm jar target/storm-starter-0.0.1-SNAPSHOT.jar storm.starter.WordCountTopology word-count-topology**

   The following information is displayed:

   545 [main] INFO backtype.storm.StormSubmitter - Jar not uploaded to master yet. Submitting jar...
   558 [main] INFO backtype.storm.StormSubmitter - Uploading topology jar target/storm-starter-0.0.1-SNAPSHOT.jar to assigned location: storm-local/nimbus/inbox/stormjar-9ab704ff-29f3-4b9d-b0ac-e9e41d4399dd.jar
   609 [main] INFO backtype.storm.StormSubmitter - Successfully uploaded topology jar to assigned location: storm-local/nimbus/inbox/stormjar-9ab704ff-29f3-4b9d-b0ac-e9e41d4399dd.jar

609 [main] INFO   backtype.storm.StormSubmitter - Submitting topology word-cout-topology in distributed mode with conf {"topology.workers":3,"topology.debug":true}
937 [main] INFO   backtype.storm.StormSubmitter - Finished submitting topology: word-cout-topology

8. Now, we can see the deployed topology on the Nimbus Web UI at http://localhost:7070/, as shown in the following screenshot:
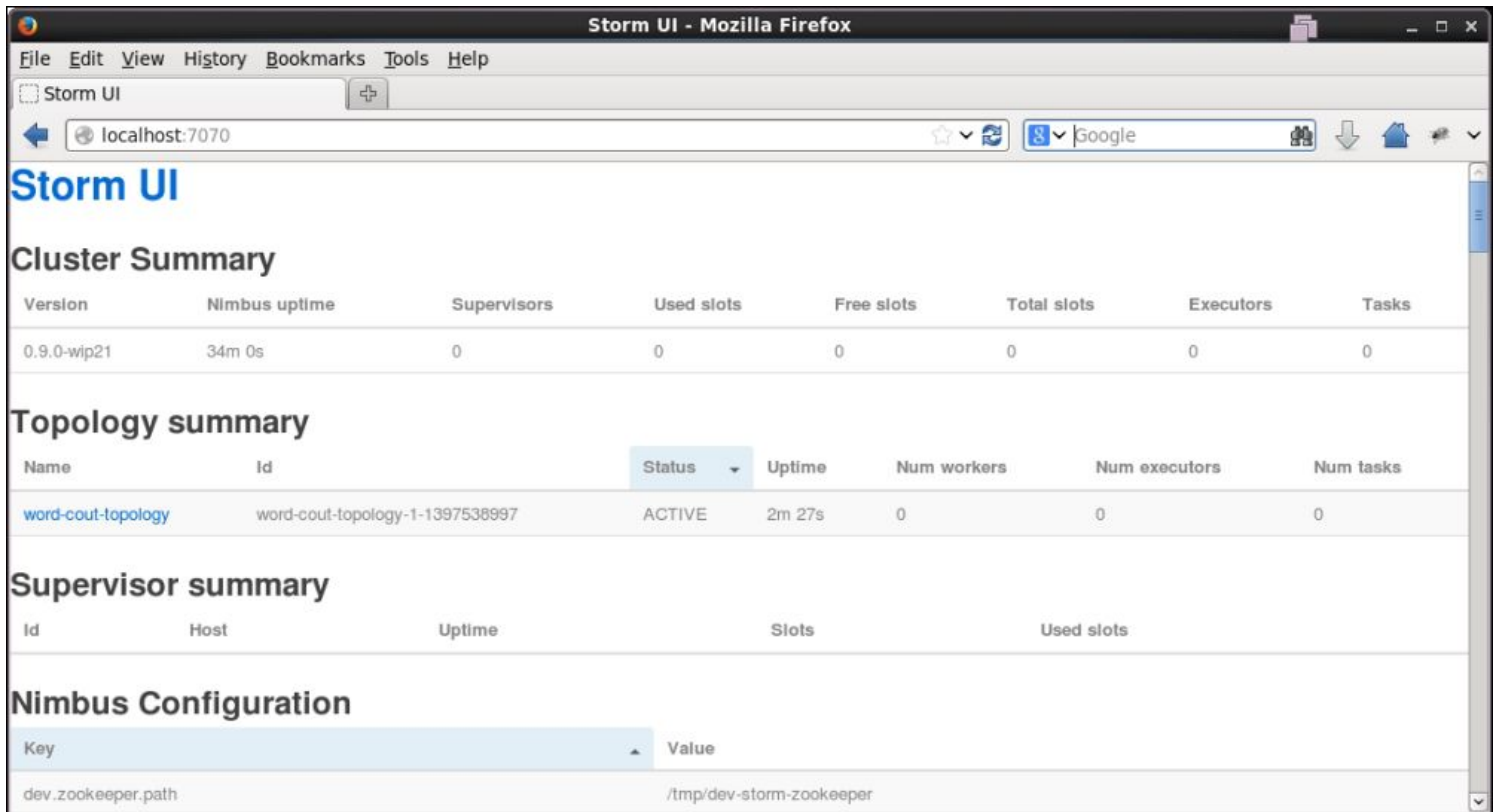
The Nimbus Web UI showing the word-count topology on YARN

9. To see how you can interact with topologies running on Storm-YARN, run the following command:

10. **storm-yarn help**

It will list all the options for interacting with various Storm processes and starting new supervisors. The following operations are supported:

- launch
- shutdown
- addSupervisors
- startSupervisors
- stopSupervisors

- startNimbus
- stopNimbus
- getStormConfig
- setStormConfig
- startUI
- stopUI

In this section, we built a Storm-Started topology and ran it over Storm-YARN.

Summary

In this chapter, we explored Apache Hadoop in depth and covered its components, such as HDFS, YARN, and so on, that are part of a Hadoop cluster. We also learned about the subcomponents of an HDFS cluster and a YARN cluster and the ways in which they interact with each other. Then, we walked through setting up a single-node Hadoop cluster.

We also introduced Storm-YARN, which was the object of this chapter. Storm-YARN enables you to run Storm topologies on a Hadoop cluster. This helps from the manageability and operations points of view. Finally, we learned how to deploy a topology on Storm running over YARN.

In the next chapter, we will see how Storm can integrate with other Big Data technologies, such as HBase and Redis.

Chapter 7. Integrating Storm with JMX, Ganglia, HBase, and Redis

In the previous chapter, we covered an overview of Apache Hadoop and its various components, overview of Storm-YARN and deploying Storm-YARN on Apache Hadoop.

In this chapter, we will explain how you can monitor the Storm cluster using well-known monitoring tools such as **Java Managements Extensions** (**JMX**) and Ganglia.

We will also cover sample examples that will demonstrate how you can store the process data into databases and a distributed cache.

In this chapter, we will cover the following topics:

- Monitoring Storm using JMX
- Monitoring Storm using Ganglia
- Integrating Storm with HBase
- Integrating Storm with Redis

Monitoring the Storm cluster using JMX

In Chapter 3, *Monitoring the Storm Cluster*, we learned how to monitor a Storm cluster using the Storm UI or Nimbus thrift API. This section will explain how you can monitor the Storm cluster using JMX. JMX is a set of specifications used to manage and monitor applications running in the JVM. We can collect or display the Storm metrics such as heap size, non-heap size, number of threads, number of loaded classes, heap and non-heap memory, and virtual machine arguments, and manage objects on the JMX console. The following are the steps we need to perform to monitor the Storm cluster using JMX:

1. We will need to add the following line in the storm.yaml file of each supervisor node to enable JMX on each of them:

   supervisor.childopts: -verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.port=12346

Here, 12346 is the port number used to collect the supervisor **Java Virtual Machine (JVM)** metrics through JMX.

2. Add the following line in the storm.yaml file of the Nimbus machine to enable JMX on the Nimbus node:

nimbus.childopts: -verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.port=12345

Here, 12345 is the port number used to collect the Nimbus JVM metrics through JMX.

3. Also, you can collect the JVM metrics of worker processes by adding the following line in the storm.yaml file of each supervisor node:

worker.childopts: -verbose:gc -XX:+PrintGCTimeStamps -XX:+PrintGCDetails -Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false -Dcom.sun.management.jmxremote.port=2%ID%

Here, %ID% denotes the port number of the worker processes. If the port of the worker process is 6700, then its JVM metrics are published on port number 26700 (2%ID%).

4. Now, run the following commands on any machine where Java is installed to start the JConsole:
5. **cd $JAVA_HOME**
6. **./bin/jconsole**

The following screenshot shows how we can connect to the supervisor JMX port using the JConsole:
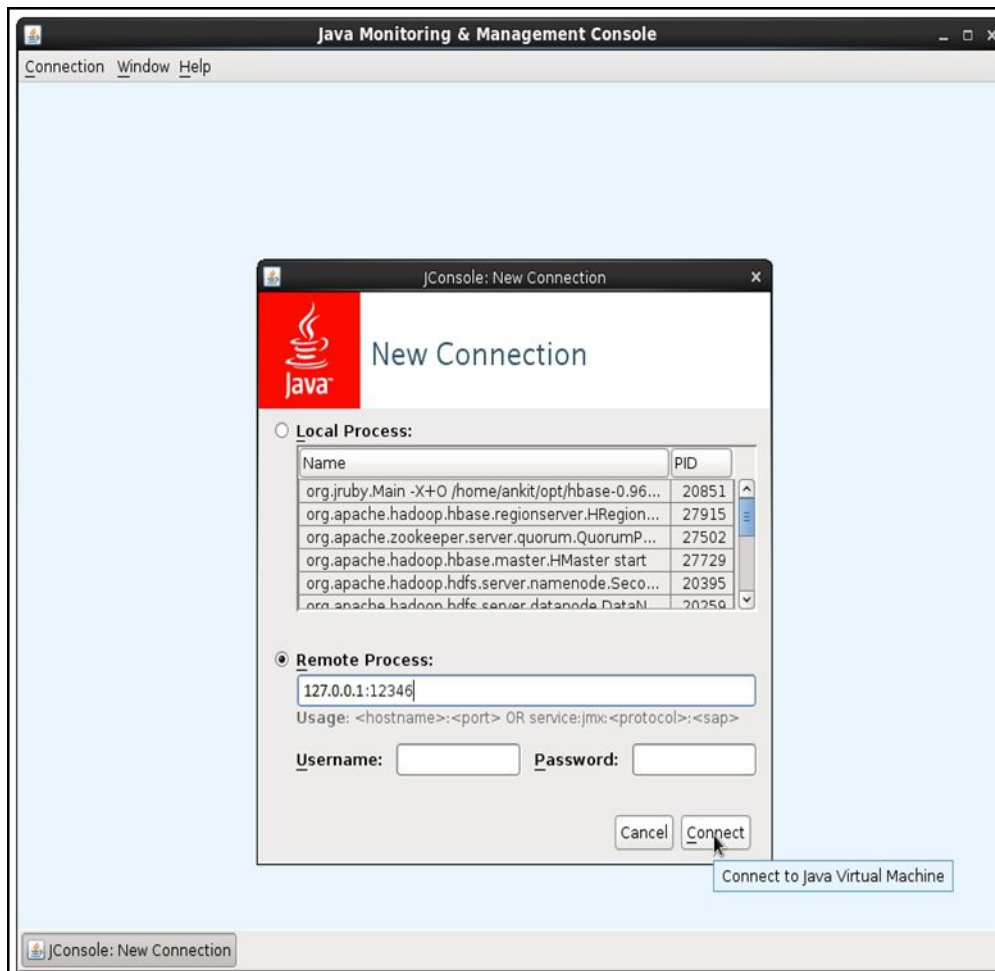
The JMX connection page

If you open the JMX console on a machine other than the supervisor machine, then you need to use the IP address of the supervisor machine in the preceding screenshot instead of 127.0.0.1.

Now, click on the **Connect** button to view the metrics of the supervisor node. The following screenshot shows what the metrics of the Storm supervisor node looks like on the JMX console:
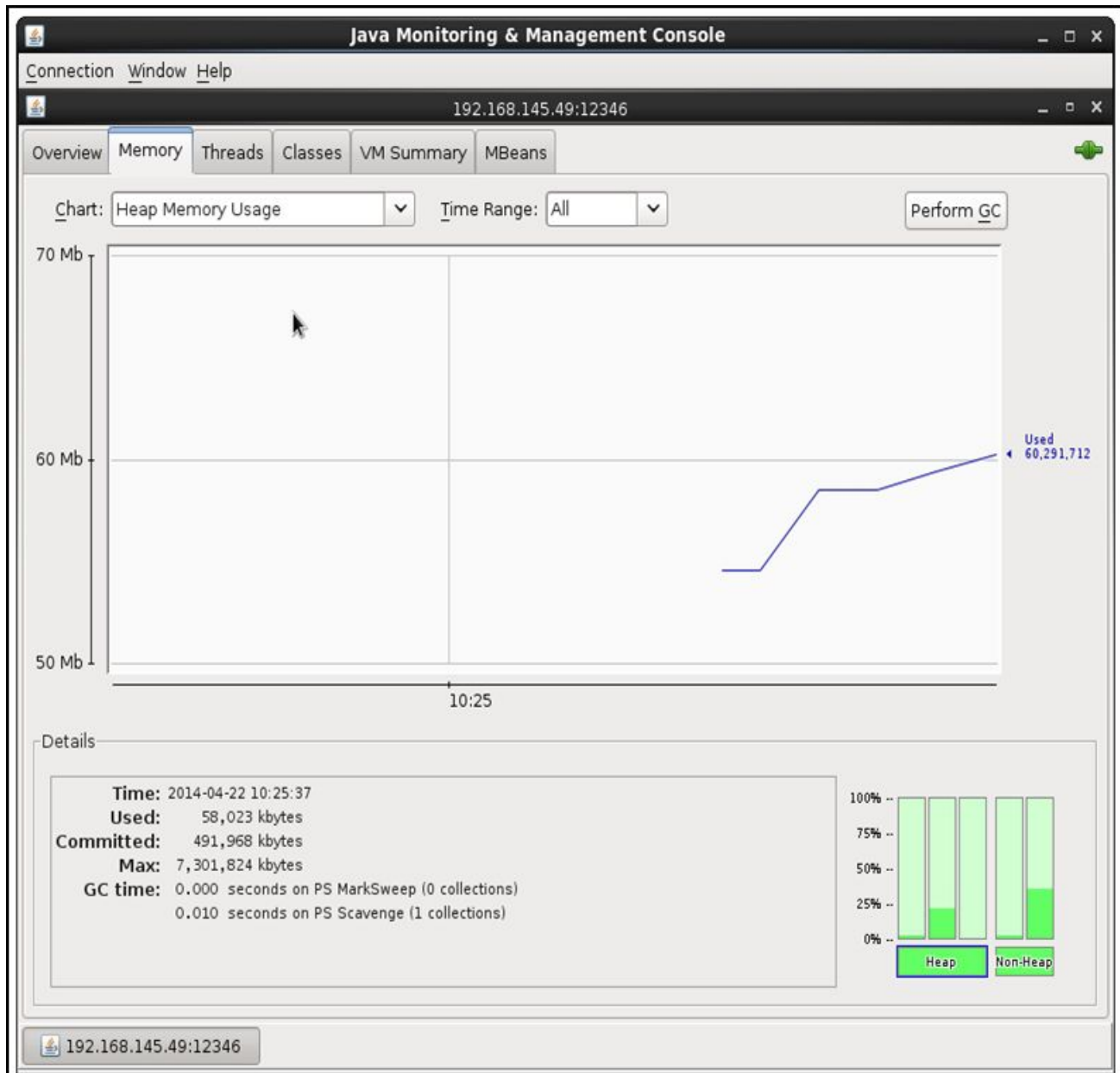
The JMX console

Similarly, you can collect the JVM metrics of the Nimbus node by specifying the IP address and the JMX port of the Nimbus machine on the JMX console.

The following section will explain how you can display the Storm cluster metrics on Ganglia.

Monitoring the Storm cluster using Ganglia

Ganglia is a monitoring tool that is used to collect the metrics of different types of processes that run on a cluster. In most of the applications, Ganglia is used as the centralized monitoring tool

to display the metrics of all the processes that run on a cluster. Hence, it is essential that you enable the monitoring of the Storm cluster through Ganglia.

Ganglia has three important components:

- **Gmond**: This is a monitoring daemon of Ganglia that collects the metrics of nodes and sends this information to the Gmetad server. To collect the metrics of each Storm node, you will need to install the Gmond daemon on each of them.
- **Gmetad**: This gathers the metrics from all the Gmond nodes and stores them in the round-robin database.
- **The Ganglia web interface**: This displays the metrics information in a graphical form.

Storm doesn't have built-in support to monitor the Storm cluster using Ganglia. However, with jmxtrans, you can enable Storm monitoring using Ganglia. The **jmxtrans** tool allows you to connect to any JVM and fetches its JVM metrics without writing a single line of code. The JVM metrics exposed via JMX can be displayed on Ganglia using jmxtrans. Hence, jmxtrans acts as a bridge between Storm and Ganglia.

The following diagram shows how jmxtrans are used between the Storm node and Ganglia:
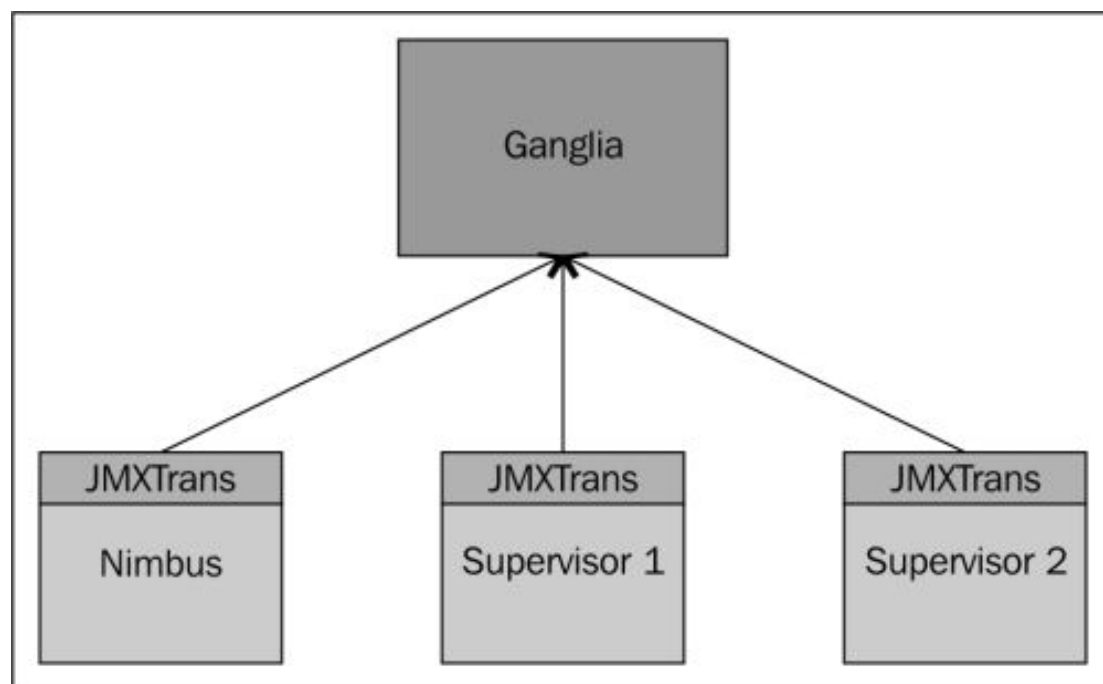
Integrating Ganglia with Storm

Perform the following steps to set up jmxtrans and Ganglia:

1. Run the following commands to download and install the jmxtrans tool on each Storm node:
2. **wget https://jmxtrans.googlecode.com/files/jmxtrans-239-0.noarch.rpm**

3. **sudo rpm -i jmxtrans-239-0.noarch.rpm**

4. Run the following commands to install the Ganglia Gmond and Gmetad packages on any machine in a network. You can deploy the Gmetad and Gmond processes on a machine that will not be a part of the Storm cluster.

5. **sudo yum -q -y install rrdtool**

6. **sudo yum −q −y install ganglia-gmond**

7. **sudo yum −q −y install ganglia-gmetad**

8. **sudo yum −q −y install ganglia-web**

9. Edit the following line in the gmetad.conf configuration file, which is located at /etc/ganglia in the Gmetad process. We are editing this file to specify the name of the data source and the IP address of the Ganglia Gmetad machine.

   data_source "stormcluster" 127.0.0.1

   # Note

   You can replace 127.0.0.1 with the IP address of the Ganglia Gmetad machine.

10. Edit the following line in the gmond.conf configuration file, which is located at /etc/ganglia, in the Gmond process:

11. cluster {

12.   name = "stormcluster"

13.   owner = "clusterOwner"

14.   latlong = "unspecified"

15.   url = "unspecified"

16. }

17. host {

18.   location = "unspecified"

19. }

20. udp_send_channel {

21.   host = 127.0.0.1

22.   port = 8649

23.   ttl = 1

24. }

25. udp_recv_channel {

26.   port = 8649

   }

Here, 127.0.0.1 is the IP address of the Storm node. You need to replace 127.0.0.1 with the actual IP address of the machine. We have mainly edited the following entries in the Gmond configuration file:

- The cluster name
- The host address of the head Gmond node in the udp_send channel
- The port in the udp_recv channel

2. Edit the following line in the ganglia.conf file, which is located at /etc/httpd/conf.d. We are editing the ganglia.conf file to enable access on the Ganglia UI from all machines.

Alias /ganglia /usr/share/ganglia<Location /ganglia>Allow from all</Location>

Note

The ganglia.conf file can be found on the node where the Ganglia web frontend application is installed. In our case, the Ganglia web interface and the Gmetad server are installed on the same machine.

3. Run the following commands to start the Ganglia Gmond, Gmetad, and Web UI processes:
4. **sudo service gmond start**
5.
6. **setsebool -P httpd_can_network_connect 1**
7. **sudo service gmetad start**
8.
9. **sudo service httpd stop**
10. **sudo service httpd start**
11. Now, go to http://127.0.0.1/ganglia to verify the installation of Ganglia.

Note

Replace 127.0.0.1 with the IP address of the Ganglia web interface machine.

12. Now, you will need to write a supervisor.json file on each supervisor node to collect the JVM metrics of the Storm supervisor node using jmxtrans and publish them on Ganglia using the com.googlecode.jmxtrans.model.output.GangliaWriter OutputWriters class. The com.googlecode.jmxtrans.model.output.GangliaWriter OutputWriters class is used to process the input JVM metrics and convert them into the format used by Ganglia. The following is the content for the supervisor.json JSON file:
13. {

```
14.     "servers" : [ {
15.        "port" : "12346",
16.        "host" : "IP_OF_SUPERVISOR_MACHINE",
17.        "queries" : [ {
18.          "outputWriters": [{
19.            "@class":"com.googlecode.jmxtrans.model.output.GangliaWriter",
20.            "settings": {
21.               "groupName": "supervisor",
22.               "host": "IP_OF_GANGLIA_GMOND_SERVER",
23.               "port": "8649" }
24.          }],
25.          "obj": "java.lang:type=Memory",
26.          "resultAlias": "supervisor",
27.          "attr": ["ObjectPendingFinalizationCount"]
28.        },
29.        {
30.          "outputWriters": [{
31.            "@class":"com.googlecode.jmxtrans.model.output.GangliaWriter",
32.            "settings": {
33.               "groupName": " supervisor ",
34.               "host": "IP_OF_GANGLIA_GMOND_SERVER",
35.               "port": "8649"
36.            }
37.          }],
38.          "obj": "java.lang:name=Copy,type=GarbageCollector",
39.          "resultAlias": " supervisor ",
40.          "attr": [
41.            "CollectionCount",
42.            "CollectionTime"
43.          ]
44.        },
45.        {
46.          "outputWriters": [{
47.            "@class": "com.googlecode.jmxtrans.model.output.GangliaWriter",
48.            "settings": {
49.               "groupName": "supervisor ",
50.               "host": "IP_OF_GANGLIA_GMOND_SERVER",
51.               "port": "8649"
```

```
52.          }
53.        }],
54.        "obj": "java.lang:name=Code Cache,type=MemoryPool",
55.        "resultAlias": "supervisor ",
56.        "attr": [
57.          "CollectionUsageThreshold",
58.          "CollectionUsageThresholdCount",
59.          "UsageThreshold",
60.          "UsageThresholdCount"
61.        ]
62.      },
63.      {
64.        "outputWriters": [{
65.          "@class": "com.googlecode.jmxtrans.model.output.GangliaWriter",
66.          "settings": {
67.            "groupName": "supervisor ",
68.            "host": "IP_OF_GANGLIA_GMOND_SERVER",
69.            "port": "8649"
70.          }
71.        }],
72.        "obj": "java.lang:type=Runtime",
73.        "resultAlias": "supervisor",
74.        "attr": [
75.          "StartTime",
76.          "Uptime"
77.        ]
78.      }],
79.      "numQueryThreads" : 2
80.    }]
    }
```

Here, 12346 is the JMX port of the supervisor specified in the storm.yaml file.

You need to replace the IP_OF_SUPERVISOR_MACHINE value with the IP address of the supervisor machine. If you have two supervisors in a cluster, then the supervisor.json file of node 1 contains the IP address of node 1, and the supervisor.json file of node 2 contains the IP address of node 2.

You need to replace the IP_OF_GANGLIA_GMOND_SERVER value with the IP address of the Ganglia Gmond server.

81. Create nimbus.json JSON file on the Nimbus node. Using jmxtrans, collect the Storm Nimbus process JVM metrics and publish them on Ganglia using the com.googlecode.jmxtrans.model.output.GangliaWriter OutputWriters class. The following is the contents of the nimbus.json JSON file:

82. {
83.   "servers" : [{
84.   "port" : "12345",
85.   "host" : "IP_OF_NIMBUS_MACHINE",
86.   "queries" : [
87.   { "outputWriters": [{
88.   "@class": "com.googlecode.jmxtrans.model.output.GangliaWriter",
89.   "settings": {
90.   "groupName": "nimbus",
91.   "host": "IP_OF_GANGLIA_GMOND_SERVER",
92.   "port": "8649"
93.   }
94.   }],
95.   "obj": "java.lang:type=Memory",
96.   "resultAlias": "nimbus",
97.   "attr": ["ObjectPendingFinalizationCount"]
98.   },
99.   {
100.   "outputWriters": [{
101.   "@class": "com.googlecode.jmxtrans.model.output.GangliaWriter",
102.   "settings": {
103.   "groupName": "nimbus",
104.   "host": "IP_OF_GANGLIA_GMOND_SERVER",
105.   "port": "8649"
106.   }
107.   }],
108.   "obj": "java.lang:name=Copy,type=GarbageCollector",
109.   "resultAlias": "nimbus",
110.   "attr": [
111.   "CollectionCount",
112.   "CollectionTime"
113.   ]

```
114.        },
115.        {
116.          "outputWriters": [{
117.            "@class": "com.googlecode.jmxtrans.model.output.GangliaWriter",
118.            "settings": {
119.               "groupName": "nimbus",
120.               "host": "IP_OF_GANGLIA_GMOND_SERVER",
121.               "port": "8649"
122.            }
123.          }],
124.          "obj": "java.lang:name=Code Cache,type=MemoryPool",
125.          "resultAlias": "nimbus",
126.          "attr": [
127.            "CollectionUsageThreshold",
128.            "CollectionUsageThresholdCount",
129.            "UsageThreshold",
130.            "UsageThresholdCount"
131.          ]
132.        },
133.        {
134.          "outputWriters": [{
135.            "@class": "com.googlecode.jmxtrans.model.output.GangliaWriter",
136.            "settings": {
137.               "groupName": "nimbus",
138.               "host": "IP_OF_GANGLIA_GMOND_SERVER",
139.               "port": "8649"
140.            }
141.          }],
142.          "obj": "java.lang:type=Runtime",
143.          "resultAlias": "nimbus",
144.          "attr": [
145.            "StartTime",
146.            "Uptime"
147.          ]
148.        }]
149.       "numQueryThreads" : 2
150.     } ]
      }
```

Here, 12345 is the JMX port of the Nimbus machine specified in the storm.yaml file.

You need to replace the IP_OF_NIMBUS_MACHINE value with the IP address of the Nimbus machine.

You need to replace the IP_OF_GANGLIA_GMOND_SERVER value with the IP address of the Ganglia Gmond server.

151. Run the following commands on each Storm node to start the jmxtrans process:

152. **cd /usr/share/jmxtrans/**

153. **sudo ./jmxtrans.sh start PATH_OF_JSON_FILES**

Here, PATH_OF_JSON_FILE is the location of the supervisor.json and nimbus.json files.

154. Now, go to the Ganglia page at http://127.0.0.1/ganglia to view the Storm metrics. The following screenshot shows what the Storm metrics look like:
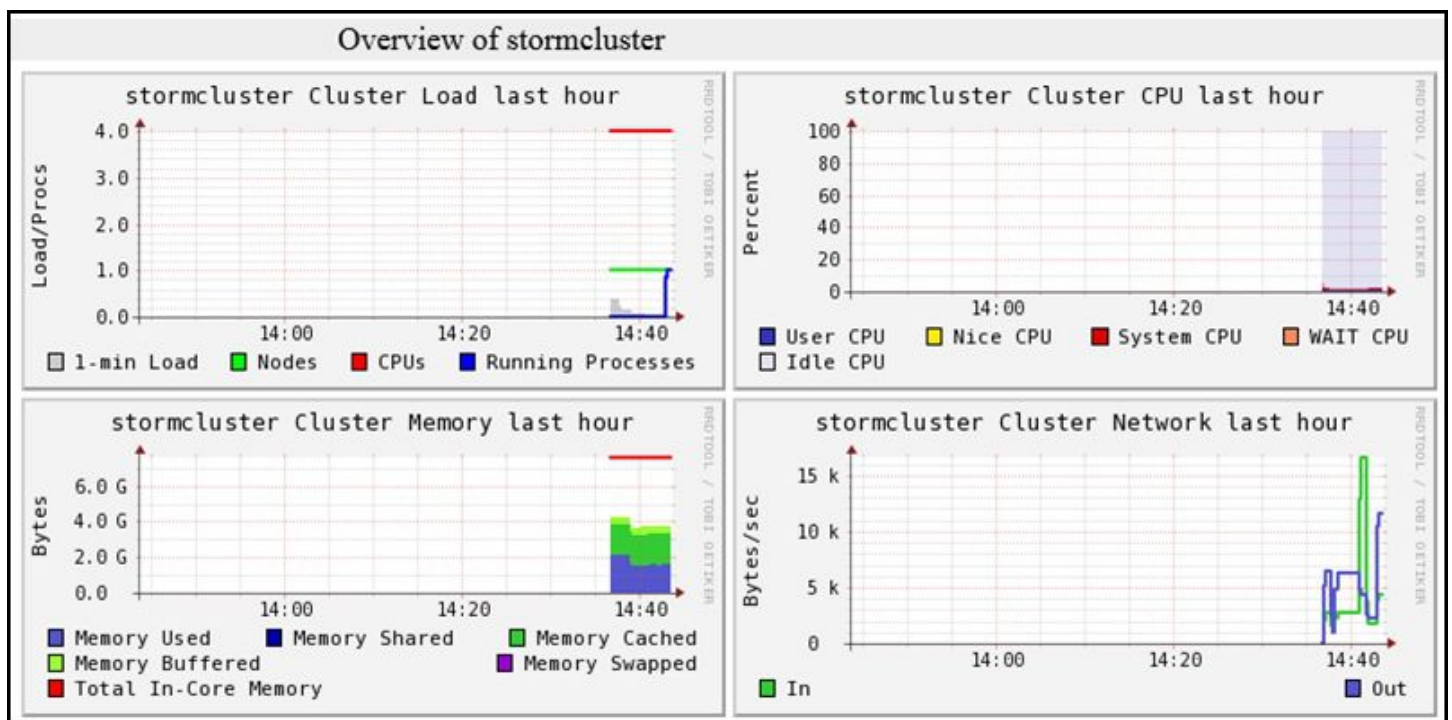
The Ganglia home page

155. Perform the followings steps to view the metrics of Storm Nimbus and the supervisor processed on the Ganglia UI:

0. Open the Ganglia page.
1. Now click on the stormCluster link to view the metrics of the Storm cluster.

2. The following screenshot shows the metrics of the Storm supervisor node:

Supervisor metrics

3. The following screenshot shows the metrics of the Storm Nimbus node:
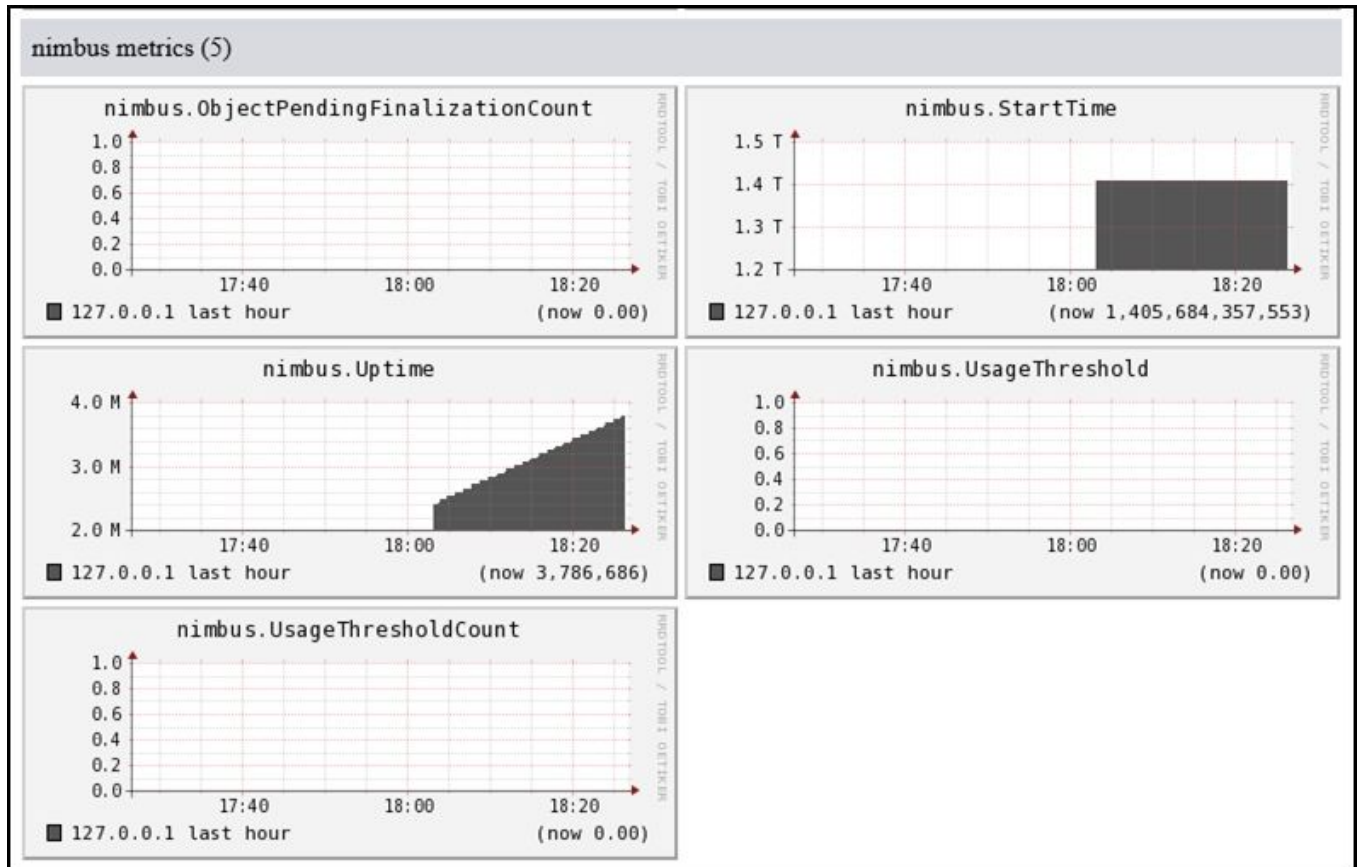
Nimbus metrics

In the following section, we will explain how you can store the data processed by Storm on the HBase database.

Integrating Storm with HBase

As explained in earlier chapters, Storm is meant for real-time data processing. However, in most cases, you will need to store the processed data in a data store so that you can use the stored data for further analysis and can execute the analysis query on the data stored. This section explains how you can store the data processed by Storm in HBase.

HBase is a NoSQL, multidimensional, sparse, horizontal scalable database modeled after Google BigTable. HBase is built on top Hadoop, which means it relies on Hadoop and integrates with the MapReduce framework very well. Hadoop provides the following benefits to HBase.

- A distributed data store that runs on top of commodity hardware
- Fault tolerance

We will assume that you have HBase installed and running on your system. You can refer to the blog on HBase installation at http://ankitasblogger.blogspot.in/2011/01/installing-hbase-in-cluster-complete.html.

We will create a sample Storm topology that explains how you can store the data processed by Storm to HBase using the following steps:

1. Create a Maven project using com.learningstorm for the Group ID and storm-hbase for the Artifact ID.
2. Add the following dependencies and repositories to the pom.xml file:
3.   <repositories>
4.   <repository>
5.   <id>clojars.org</id>
6.   <url>http://clojars.org/repo</url>
7.   </repository>
8.   </repositories>
9.   <dependencies>
10.   <dependency>
11.   <groupId>storm</groupId>
12.   <artifactId>storm</artifactId>
13.   <version>0.9.0.1</version>
14.   <exclusions>
15.   <exclusion>
16.   <artifactId>log4j-over-slf4j</artifactId>
17.   <groupId>org.slf4j</groupId>
18.   </exclusion>
19.   </exclusions>
20.   </dependency>
21.   <dependency>
22.   <groupId>org.apache.hadoop</groupId>
23.   <artifactId>hadoop-core</artifactId>
24.   <version>1.1.1</version>
25.   </dependency>
26.   <dependency>
27.   <groupId>org.slf4j</groupId>
28.   <artifactId>slf4j-api</artifactId>
29.   <version>1.7.7</version>
30.   </dependency>
31.   <dependency>

```
32.          <groupId>org.apache.hbase</groupId>
33.          <artifactId>hbase</artifactId>
34.          <version>0.94.5</version>
35.          <exclusions>
36.            <exclusion>
37.              <artifactId>zookeeper</artifactId>
38.              <groupId>org.apache.zookeeper</groupId>
39.            </exclusion>
40.          </exclusions>
41.        </dependency>
42.        <dependency>
43.          <groupId>junit</groupId>
44.          <artifactId>junit</artifactId>
45.          <version>4.10</version>
46.        </dependency>
      </dependencies>
```

47. Create an HBaseOperations class in the com.learningstorm.storm_hbase package. The HBaseOperations class contains two methods:

- createTable(String tableName, List<String> ColumnFamilies): This method takes the name of the table and the HBase column family list as input to create a table in HBase.
- insert(Map<String, Map<String, Object>> record, String rowId): This method takes the record and its rowID parameter as input and inserts the input record to HBase. The following is the structure of the input record:
- 　　　　{
- 　　　　　"columnfamily1":
- 　　　　　{
- 　　　　　　"column1":"abc",
- 　　　　　　"column2":"pqr"
- 　　　　　},
- 　　　　　"columnfamily2":
- 　　　　　{
- 　　　　　　"column3":"bc",
- 　　　　　　"column4":"jkl"
- 　　　　　}
- 　　　　}

Here, columnfamily1 and columnfamily2 are the names of HBase column families, and column1, column2, column3, and column4 are the names of columns.

The rowId parameter is the HBase table row key that is used to uniquely identify each record in HBase.

The following is the source code of the HBaseOperations class:

```java
public class HBaseOperations implements Serializable{

    private static final long serialVersionUID = 1L;

    // Instance of Hadoop Configuration class
    Configuration conf = new Configuration();
    HTable hTable = null;

    public HBaseOperations(String tableName, List<String> ColumnFamilies,
    List<String> zookeeperIPs, int zkPort) {
        conf = HBaseConfiguration.create();
        StringBuffer zookeeperIP = new StringBuffer();
        // Set the zookeeper nodes
        for (String zookeeper : zookeeperIPs) {
            zookeeperIP.append(zookeeper).append(",");
        }
        zookeeperIP.deleteCharAt(zookeeperIP.length() - 1);

        conf.set("hbase.zookeeper.quorum", zookeeperIP.toString());

        // Set the zookeeper client port
        conf.setInt("hbase.zookeeper.property.clientPort", zkPort);
        // call the createTable method to create a table into HBase.
        createTable(tableName, ColumnFamilies);
        try {
            // initialize the HTable.
            hTable = new HTable(conf, tableName);
        } catch (IOException e) {
            System.out.println("Error occurred while creating instance of HTable class :
" + e);
        }
```

```java
        }

    /**
     * This method create a table into HBase
     *
     * @param tableName
     *                Name of the HBase table
     * @param ColumnFamilies
     *                List of column families
     *
     */
    public void createTable(String tableName, List<String> ColumnFamilies) {
        HBaseAdmin admin = null;
        try {
            admin = new HBaseAdmin(conf);
            // Set the input table in HTableDescriptor
            HTableDescriptor tableDescriptor = new
HTableDescriptor(Bytes.toBytes(tableName));
            for (String columnFamaliy : ColumnFamilies) {
                HColumnDescriptor columnDescriptor = new
HColumnDescriptor(columnFamaliy);
                // add all the HColumnDescriptor into HTableDescriptor
                tableDescriptor.addFamily(columnDescriptor);
            }
            /*  execute  the  creaetTable(HTableDescriptor  tableDescriptor)  of
HBaseAdmin
             * class to createTable into HBase.
             */
            admin.createTable(tableDescriptor);
            admin.close();

        }catch (TableExistsException tableExistsException) {
            System.out.println("Table already exist : " + tableName);
            if(admin != null) {
                try {
                    admin.close();
                } catch (IOException ioException) {
```

```java
            System.out.println("Error occurred while closing the HBaseAdmin
connection : " + ioException);
            }
        }

    }catch (MasterNotRunningException e) {
        throw new RuntimeException("HBase master not running, table creation
failed : ");
    } catch (ZooKeeperConnectionException e) {
        throw new RuntimeException("Zookeeper not running, table creation
failed : ");
    } catch (IOException e) {
        throw new RuntimeException("IO error, table creation failed : ");
    }
}

/**
 * This method insert the input record into HBase.
 *
 * @param record
 *              input record
 * @param rowId
 *              unique id to identify each record uniquely.
 */
public void insert(Map<String, Map<String, Object>> record, String rowId) {
    try {
        Put put = new Put(Bytes.toBytes(rowId));
        for (String cf : record.keySet()) {
            for (String column: record.get(cf).keySet()) {
                put.add(Bytes.toBytes(cf),                      Bytes.toBytes(column),
Bytes.toBytes(record.get(cf).get(column).toString()));
            }
        }
        hTable.put(put);
    }catch (Exception e) {
        throw new RuntimeException("Error occurred while storing record into
HBase");
    }
```

```java
        }

        public static void main(String[] args) {
            List<String> cFs = new ArrayList<String>();
            cFs.add("cf1");
            cFs.add("cf2");

            List<String> zks = new ArrayList<String>();
            zks.add("127.0.0.1");
            Map<String, Map<String, Object>> record = new HashMap<String,
    Map<String,Object>>();

            Map<String, Object> cf1 = new HashMap<String, Object>();
            cf1.put("aa", "1");

            Map<String, Object> cf2 = new HashMap<String, Object>();
            cf2.put("bb", "1");
            record.put("cf1", cf1);
            record.put("cf2", cf2);

            HBaseOperations hbaseOperations = new HBaseOperations("tableName", cFs,
    zks, 2181);
            hbaseOperations.insert(record, UUID.randomUUID().toString());

        }
    }
```

2. Create a SampleSpout class in the com.learningstorm.storm_hbase package. This class generates random records and passes them to the next action (bolt) in the topology. The following is the format of the record generated by the SampleSpout class:

["john","watson","abc"]

The following is the source code of the SampleSpout class:

```java
public class SampleSpout extends BaseRichSpout {
    private static final long serialVersionUID = 1L;
    private SpoutOutputCollector spoutOutputCollector;
```

```java
    private static final Map<Integer, String> FIRSTNAMEMAP = new HashMap<Integer,
String>();
    static {
        FIRSTNAMEMAP.put(0, "john");
        FIRSTNAMEMAP.put(1, "nick");
        FIRSTNAMEMAP.put(2, "mick");
        FIRSTNAMEMAP.put(3, "tom");
        FIRSTNAMEMAP.put(4, "jerry");
    }

    private static final Map<Integer, String> LASTNAME = new HashMap<Integer,
String>();
    static {
        LASTNAME.put(0, "anderson");
        LASTNAME.put(1, "watson");
        LASTNAME.put(2, "ponting");
        LASTNAME.put(3, "dravid");
        LASTNAME.put(4, "lara");
    }

    private static final Map<Integer, String> COMPANYNAME = new HashMap<Integer,
String>();
    static {
        COMPANYNAME.put(0, "abc");
        COMPANYNAME.put(1, "dfg");
        COMPANYNAME.put(2, "pqr");
        COMPANYNAME.put(3, "ecd");
        COMPANYNAME.put(4, "awe");
    }

    public void open(Map conf, TopologyContext context,
        SpoutOutputCollector spoutOutputCollector) {
        // Open the spout
        this.spoutOutputCollector = spoutOutputCollector;
    }

    public void nextTuple() {
```

```
        // Storm cluster repeatedly call this method to emit the continuous //
        // stream of tuples.
        final Random rand = new Random();
        // generate the random number from 0 to 4.
        int randomNumber = rand.nextInt(5);
        spoutOutputCollector.emit (new
    Values(FIRSTNAMEMAP.get(randomNumber),LASTNAME.get(randomNumber),COMP
    ANYNAME.get(randomNumber)));
    }

    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // emits the field firstName, lastName and companyName.
        declarer.declare(new Fields("firstName","lastName","companyName"));
    }
}
```

3. Create a StormHBaseBolt class in the com.learningstorm.storm_hbase package. This bolt received the tuples emitted by SampleSpout and then calls the insert() method of the HBaseOperations class to insert the record into HBase. The following is the source code of the StormHBaseBolt class:

4.   public class StormHBaseBolt implements IBasicBolt {

5.

6.   private static final long serialVersionUID = 2L;

7.   private HBaseOperations hbaseOperations;

8.   private String tableName;

9.   private List<String> columnFamilies;

10.   private List<String> zookeeperIPs;

11.   private int zkPort;

12.   /**

13.   * Constructor of StormHBaseBolt class

14.   *

15.   * @param tableName

16.   *   HBaseTableNam

17.   * @param columnFamilies

18.   *   List of column families

19.   * @param zookeeperIPs

20.   *   List of zookeeper nodes

21.   * @param zkPort

22.   *   Zookeeper client port

```java
23.    */
24.    public    StormHBaseBolt(String    tableName,    List<String>    columnFamilies,
    List<String> zookeeperIPs, int zkPort) {
25.        this.tableName =tableName;
26.        this.columnFamilies = columnFamilies;
27.        this.zookeeperIPs = zookeeperIPs;
28.        this.zkPort = zkPort;
29.
30.    }
31.
32.    public void execute(Tuple input, BasicOutputCollector collector) {
33.        Map<String, Map<String, Object>> record = new HashMap<String, Map<String,
    Object>>();
34.        Map<String, Object> personalMap = new HashMap<String, Object>();
35.
36.        personalMap.put("firstName", input.getValueByField("firstName"));
37.        personalMap.put("lastName", input.getValueByField("lastName"));
38.
39.        Map<String, Object> companyMap = new HashMap<String, Object>();
40.        companyMap.put("companyName", input.getValueByField("companyName"));
41.
42.        record.put("personal", personalMap);
43.        record.put("company", companyMap);
44.        // call the inset method of HBaseOperations class to insert record into
45.        // HBase
46.        hbaseOperations.insert(record, UUID.randomUUID().toString());
47.    }
48.    public void declareOutputFields(OutputFieldsDeclarer declarer) {
49.
50.    }
51.
52.    @Override
53.    public Map<String, Object> getComponentConfiguration() {
54.        // TODO Auto-generated method stub
55.        return null;
56.    }
57.
58.    @Override
```

```
59.    public void prepare(Map stormConf, TopologyContext context) {
60.        // create the instance of HBaseOperations class
61.        hbaseOperations = new HBaseOperations(tableName, columnFamilies,
62.        zookeeperIPs, zkPort);
63.    }
64.
65.    @Override
66.    public void cleanup() {
67.        // TODO Auto-generated method stub
68.
69.    }
    }
```

The constructor of the StormHBaseBolt class takes the HBase table name, column families list, ZooKeeper IP address, and ZooKeeper port as an argument and sets the class level variables. The prepare() method of the StormHBaseBolt class will create an instance of the HBaseOperatons class.

The execute() method of the StormHBaseBolt class takes an input tuple as an argument and converts it into the HBase structure format. It also uses the java.util.UUID class to generate the HBase row ID.

70. Create a Topology class in the com.learningstorm.storm_hbase package. This class creates an instance of the spout and bolt classes and chains them together using a TopologyBuilder class. The following is the implementation of the main class:

```
71.  public class Topology {
72.      public static void main(String[] args) throws AlreadyAliveException,
    InvalidTopologyException {
73.          TopologyBuilder builder = new TopologyBuilder();
74.          List<String> zks = new ArrayList<String>();
75.          zks.add("127.0.0.1");
76.
77.          List<String> cFs = new ArrayList<String>();
78.          cFs.add("personal");
79.          cFs.add("company");
80.
81.          // set the spout class
82.          builder.setSpout("spout", new SampleSpout(), 2);
83.          // set the bolt class
```

```
84.      builder.setBolt("bolt",    new    StormHBaseBolt("user",    cFs,    zks,    2181),
         2).shuffleGrouping("spout");
85.      Config conf = new Config();
86.      conf.setDebug(true);
87.      // create an instance of LocalCluster class for
88.      // executing topology in local mode.
89.      LocalCluster cluster = new LocalCluster();
90.
91.      // StormHBaseTopology is the name of submitted topology.
92.      cluster.submitTopology("StormHBaseTopology", conf, builder.createTopology());
93.      try {
94.         Thread.sleep(60000);
95.      } catch (Exception exception) {
96.         System.out.println("Thread interrupted exception : " + exception);
97.      }
98.      System.out.println("Stopped Called : ");
99.      // kill the StormHBaseTopology
100.     cluster.killTopology("StormHBaseTopology");
101.     // shutdown the storm test cluster
102.     cluster.shutdown();
103.
104.  }
    }
```

In the following section, we will cover how you can integrate Storm with an in-memory cache called Redis.

Integrating Storm with Redis

Redis is a key value data store. The key values can be strings, lists, sets, hashes, and so on. It is extremely fast because the entire dataset is stored in the memory. The following are the steps to install Redis:

1. First, you will need to install make, gcc, and cc to compile the Redis code using the following command:
2. **sudo yum -y install make gcc cc**
3. Download, unpack, and make Redis, and copy it to /usr/local/bin using the following commands:
4. **cd /home/$USER**

Here, $USER is the name of the Linux user.

http://download.redis.io/releases/redis-2.6.16.tar.gz
tar -xvf redis-2.6.16.tar.gz
cd redis-2.6.16
make
sudo cp src/redis-server /usr/local/bin
sudo cp src/redis-cli /usr/local/bin

5. Execute the following commands to make Redis as a service:
6.   **sudo mkdir -p /etc/redis**
7.   **sudo mkdir -p /var/redis**
8.   **cd /home/$USER/redis-2.6.16/**
9.   **sudo cp utils/redis_init_script /etc/init.d/redis**
10.   **wget**
    https://bitbucket.org/ptylr/public-stuff/raw/41d5c8e87ce6adb34aa16cd571c
    3f04fb4d5e7ac/etc/init.d/redis
11.   **sudo cp redis /etc/init.d/redis**
12.   **cd /home/$USER/redis-2.6.16/**
13.   **sudo cp redis.conf /etc/redis/redis.conf**
14. Now, run the following commands to add the service to chkconfig, set it to autostart, and actually start the service:
15.   **chkconfig --add redis**
16.   **chkconfig redis on**
17.   **service redis start**
18. Check the installation of Redis with the following command:
19.   **redis-cli ping**

If the result of the test command is PONG, then the installation has been successful.

Now, we will assume that you have the Redis service up and running.

Next, we will create a sample Storm topology that will explain how you can store the data processed by Storm in Redis.

20. Create a Maven project using com.learningstorm for the Group ID and storm-redis for the Artifact ID.
21. Add the following dependencies and repositories in the pom.xml file:
22.   <repositories>
23.   <repository>

24.   <id>clojars.org</id>
25.   <url>http://clojars.org/repo</url>
26.   </repository>
27.   </repositories>
28.   <dependencies>
29.   <dependency>
30.   <groupId>storm</groupId>
31.   <artifactId>storm</artifactId>
32.   <version>0.9.0.1</version>
33.   </dependency>
34.   <dependency>
35.   <groupId>junit</groupId>
36.   <artifactId>junit</artifactId>
37.   <version>3.8.1</version>
38.   <scope>test</scope>
39.   </dependency>
40.   <dependency>
41.   <groupId>redis.clients</groupId>
42.   <artifactId>jedis</artifactId>
43.   <version>2.4.2</version>
44.   </dependency>
45.   <dependency>
46.   <groupId>com.fasterxml.jackson.core</groupId>
47.   <artifactId>jackson-core</artifactId>
48.   <version>2.1.1</version>
49.   </dependency>
50.   <dependency>
51.   <groupId>com.fasterxml.jackson.core</groupId>
52.   <artifactId>jackson-databind</artifactId>
53.   <version>2.1.1</version>
54.   </dependency>
     </dependencies>

55. Create a RedisOperations class in the com.learningstorm.storm_redis package. The RedisOperations class contains the following method:

   • insert(Map<String, Object> record, String id): This method takes the record and ID as input and inserts the input record in Redis. In the insert() method, we will first serialize the record into a string using the Jackson library and then store the

serialized record into Redis. Each record must have a unique ID because it is used to retrieve the record from Redis.

The following is the source code of the RedisOperations class:

```java
public class RedisOperations implements Serializable {

    private static final long serialVersionUID = 1L;
    Jedis jedis = null;

    public RedisOperations(String redisIP, int port) {
        // Connecting to Redis
        jedis = new Jedis(redisIP, port);
    }
    /* This method takes the record and record id as input. We will first serialize the
record into String using Jackson library and then store the whole record into
Redis.User can use the record id to retrieve the record from Redis*/
    public void insert(Map<String, Object> record, String id) {
        try {
            jedis.set(id, new ObjectMapper().writeValueAsString(record));
        } catch (Exception e) {

            System.out.println("Record not persisted into datastore");
        }

    }
}
```

2. We will use the same SampleSpout class created in the *IntegratingStorm with HBase* section.

3. Create a StormRedisBolt class in the com.learningstorm.storm_redis package. This bolt receives the tuples emitted by the SampleSpout class, converts it to the Redis structure, and then calls the insert() method of the RedisOperations class to insert the record into Redis. The following is the source code of the StormRedisBolt class:

4.   public class StormRedisBolt implements IBasicBolt{

5.

6.   private static final long serialVersionUID = 2L;

7.   private RedisOperations redisOperations = null;

8.   private String redisIP = null;

```
9.      private int port;
10.     public StormRedisBolt(String redisIP, int port) {
11.        this.redisIP = redisIP;
12.        this.port = port;
13.     }
14.
15.     public void execute(Tuple input, BasicOutputCollector collector) {
16.        Map<String, Object> record = new HashMap<String, Object>();
17.        //"firstName","lastName","companyName")
18.        record.put("firstName", input.getValueByField("firstName"));
19.        record.put("lastName", input.getValueByField("lastName"));
20.        record.put("companyName", input.getValueByField("companyName"));
21.        redisOperations.insert(record, UUID.randomUUID().toString());
22.     }
23.
24.     public void declareOutputFields(OutputFieldsDeclarer declarer) {
25.
26.     }
27.
28.     public Map<String, Object> getComponentConfiguration() {
29.        return null;
30.     }
31.
32.     public void prepare(Map stormConf, TopologyContext context) {
33.        redisOperations = new RedisOperations(this.redisIP, this.port);
34.     }
35.
36.     public void cleanup() {
37.
38.     }
39.
   }
```

In the StormRedisBolt class, we are using the java.util.UUID class to generate the Redis key.

40. Create a Topology class in the com.learningstorm.storm_redis package. This class creates an instance of the spout and bolt classes and chains them together using a TopologyBuilder class. The following is the implementation of the main class:

```
41.  public class Topology {
42.  public   static   void   main(String[]   args)   throws   AlreadyAliveException,
     InvalidTopologyException {
43.      TopologyBuilder builder = new TopologyBuilder();
44.  List<String> zks = new ArrayList<String>();
45.   zks.add("127.0.0.1");
46.
47.  List<String> cFs = new ArrayList<String>();
48.  cFs.add("personal");
49.  cFs.add("company");
50.
51.      // set the spout class
52.      builder.setSpout("spout", new SampleSpout(), 2);
53.      // set the bolt class
54.      builder.setBolt("bolt",          new          StormRedisBolt("127.0.0.1",6379),
     2).shuffleGrouping("spout");
55.
56.      Config conf = new Config();
57.      conf.setDebug(true);
58.      // create an instance of LocalCluster class for
59.      // executing topology in local mode.
60.      LocalCluster cluster = new LocalCluster();
61.
62.      // StormRedisTopology   is the name of submitted topology.
63.      cluster.submitTopology("StormRedisTopology", conf, builder.createTopology());
64.      try {
65.        Thread.sleep(10000);
66.      } catch (Exception exception) {
67.        System.out.println("Thread interrupted exception : " + exception);
68.      }
69.      // kill the StormRedisTopology
70.      cluster.killTopology("StormRedisTopology");
71.      // shutdown the storm test cluster
72.      cluster.shutdown();
73.
74.    }
   }
```

In this section, we covered installation of Redis and how we can integrate Storm with Redis.

Summary

In this chapter, we mainly concentrated on monitoring the Storm cluster through JMX and Ganglia. We also covered how we can integrate Storm with Redis and HBase.

In the next chapter, we will cover the Apache log processing case study. We will explain how you can generate business information by processing logfiles through Storm.

Chapter 8. Log Processing with Storm

In the previous chapter, we covered how we can integrate Storm with **Redis** and **HBase**. Also, we learned how to use **Ganglia** and **JMX** to monitor the **Storm Cluster**.

In this chapter, we will cover the most popular use case of Storm, that is, **log processing**.

This chapter covers the following topics:

- Server log-processing elements
- Producing the server log in Kafka
- Splitting the server logfile
- Identifying the country name, the operating system type, and the browser type
- Extracting the searched keyword
- Persisting the process data
- Defining a topology and the Kafka spout
- Deploying a topology
- MySQL queries

Server log-processing elements

Log processing is becoming a need for every organization to collect business information from log data. In this chapter, we are basically going to work on how we can process the server log data to collect business information using Storm.

The following diagram shows the log-processing topology and illustrates all the elements that we will develop in this chapter:
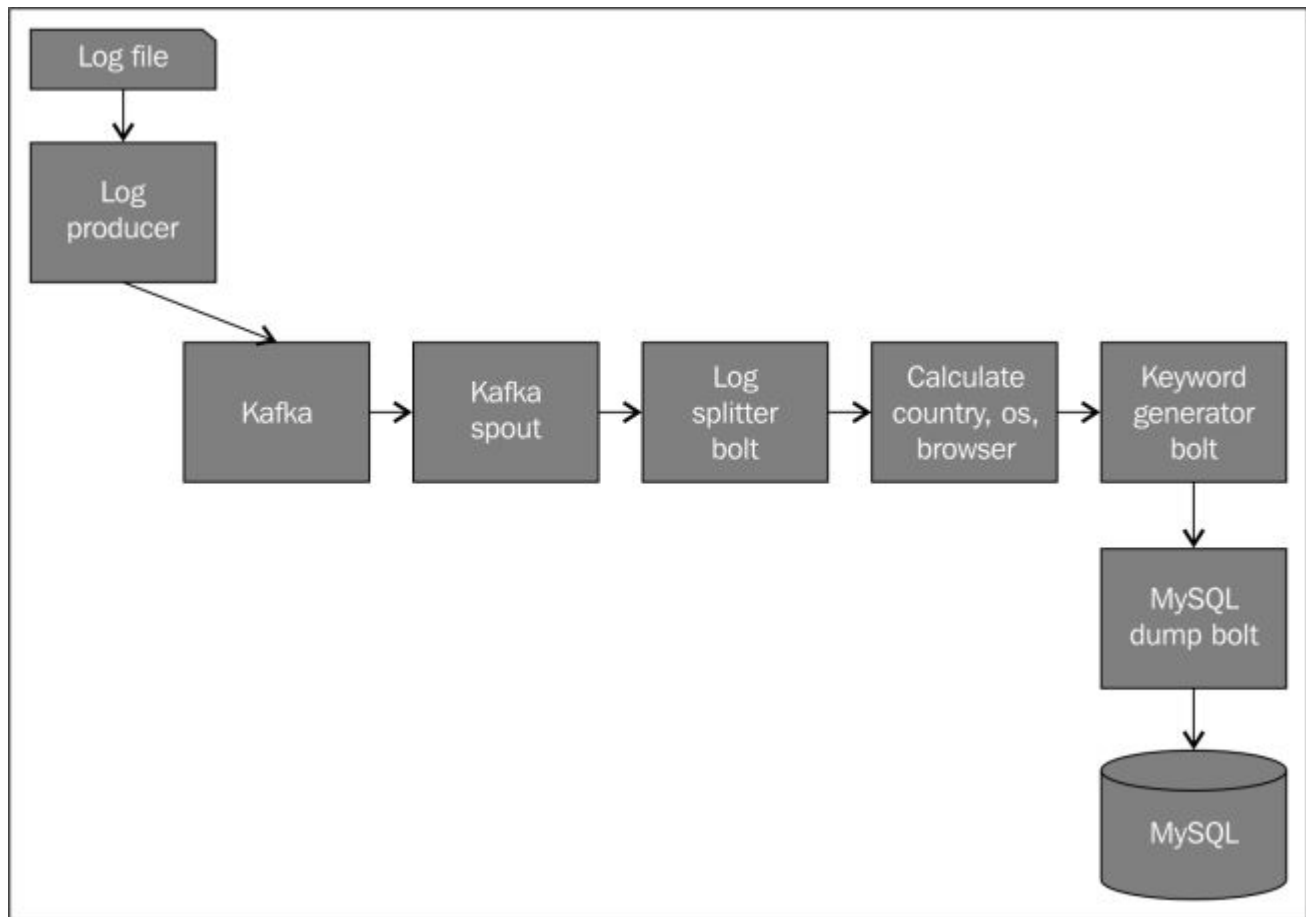
Producing the Apache log in Kafka

As explained in Chapter 4, *Storm and Kafka Integration*, Kafka is a distributed messaging queue and can integrate with Storm very well. In this section, you'll see how to write a Kafka producer that will read the server logfile and produce the log in Kafka.

As we all know, Storm provides guaranteed message processing, which means every message that enters the Storm topology will be processed at least once. In Storm, data loss is possible only at the spout. This happens if the processing capacity of the Storm spout is less than the producing capacity of the data publisher. Hence, to avoid data loss at the Storm spout, we will generally publish the data into a messaging queue, and the Storm spout will use that messaging queue as the data source.

We will create a Maven project that will publish the server log into a Kafka broker. Perform the following steps to create the server log producer:

1. Create a new Maven project with com.learningstorm for groupId and kafkaLogProducer for artifactId.
2. Add the following dependencies for Kafka in pom.xml:

```
3.    <dependency>
4.      <groupId>org.apache.kafka</groupId>
5.      <artifactId>kafka_2.10</artifactId>
6.      <version>0.8.0</version>
7.      <exclusions>
8.        <exclusion>
9.          <groupId>com.sun.jdmk</groupId>
10.         <artifactId>jmxtools</artifactId>
11.       </exclusion>
12.       <exclusion>
13.         <groupId>com.sun.jmx</groupId>
14.         <artifactId>jmxri</artifactId>
15.       </exclusion>
16.     </exclusions>
17.   </dependency>
18.   <dependency>
19.     <groupId>org.apache.logging.log4j</groupId>
20.     <artifactId>log4j-slf4j-impl</artifactId>
21.     <version>2.0-beta9</version>
22.   </dependency>
23.   <dependency>
24.     <groupId>org.apache.logging.log4j</groupId>
25.     <artifactId>log4j-1.2-api</artifactId>
26.     <version>2.0-beta9</version>
      </dependency>
```

27. Add the following build plugins to pom.xml. These plugins will let us execute the producer using Maven:

```
28.  <build>
29.    <plugins>
30.      <plugin>
31.        <groupId>org.codehaus.mojo</groupId>
32.        <artifactId>exec-maven-plugin</artifactId>
33.        <version>1.2.1</version>
34.        <executions>
35.          <execution>
36.            <goals>
37.              <goal>exec</goal>
38.            </goals>
```

39.   </execution>
40.   </executions>
41.   <configuration>
42.   <executable>java</executable>
43.   <includeProjectDependencies>true</includeProjectDependencies>
44.   <includePluginDependencies>false</includePluginDependencies>
45.   <classpathScope>compile</classpathScope>
46.   <mainClass>com.learningstorm.kafka.WordsProducer</mainClass>
47.   </configuration>
48.   </plugin>
49.   </plugins>
    </build>

50. Now, we will create the ApacheLogProducer class in the com.learningstorm.kafkaLogProducer package. This class will read the server logfile and produce each log line in the apache_log topic in Kafka as a single message. The following is the code for the ApacheLogProducer class with its explanation:

51.   public class KafkaProducer {

52.

53.   public static void main(String[] args) {

54.   // Build the configuration required

55.   // for connecting to Kafka

56.   Properties props = new Properties();

57.

58.   // List of kafka brokers.

59.   // The complete list of brokers is not required as

60.   // the producer will auto discover

61.   //the rest of the brokers.

62.   props.put("metadata.broker.list", "localhost:9092");

63.

64.   // Serializer used for sending data to kafka.

65.   // Since we are sending string,

66.   // we are using StringEncoder.

67.   props.put("serializer.class", "kafka.serializer.StringEncoder");

68.

69.   // We want acknowledgement from Kafka that

70.   // the messages have been properly received.

71.   props.put("request.required.acks", "1");

72.

```
73.      // Create the producer instance
74.      ProducerConfig config = new ProducerConfig(props);
75.      Producer<String, String> producer = new Producer<String, String>(config);
76.      try {
77.      FileInputStream                fstream                =                new
   FileInputStream("./src/main/resources/apache_test.log");
78.      BufferedReader br = new BufferedReader(new InputStreamReader(fstream));
79.      String strLine;
80.      /* read log line by line */
81.      while ((strLine = br.readLine()) != null) {
82.       KeyedMessage<String,    String>    data    =    new    KeyedMessage<String,
   String>("apache_log", strLine);
83.          producer.send(data);
84.      }
85.      br.close();
86.      fstream.close();
87.      }catch (Exception e) {
88.       throw new RuntimeException("Error occurred while persisting records : ");
89.      }
90.
91.      // close the producer
92.      producer.close();
93.   }
94.
   }
```

Replace localhost of the preceding ApacheLogProducer class with the IP address of the broker machine.

Also, replace ./src/main/resources/apache_test.log (the server log path) with the path of your logfile.

95. The preceding ApacheLogProducer class will directly produce the log data in the apache_log topic in Kafka. Hence, you need to create the apache_log topic in Kafka before you run the ApacheLogProducer producer. To do so, go to the home directory of Kafka and execute the following command:

96. **bin/kafka-create-topic.sh --zookeeper localhost:2181 --replica 1 --partition 1 --topic apache_log**

97. **creation succeeded!**

98. Now, you can run ApacheLogProducer by executing the following Maven command. The ApacheLogProducer needs to be run on a machine where the server logs are generated:

99. **mvn compile exec:java**

100. Now, run the Kafka console consumer to check whether the messages are successfully produced in Kafka. Run the following command to start the Kafka console consumer:

101. **bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic apache_log --from-beginning**

The following information is displayed:

4.19.162.143 - - [4-03-2011:06:20:31 -0500] "GET / HTTP/1.1" 200 864 "http://www.adeveloper.com/resource.html" "Mozilla/5.0 (Windows; U; Windows NT 5.1; hu-HU; rv:1.7.12) Gecko/20050919 Firefox/1.0.7"
4.19.162.152 - - [4-03-2011:06:20:31 -0500] "GET / HTTP/1.1" 200 864 "http://www.adeveloper.com/resource.html" "Mozilla/5.0 (Windows; U; Windows NT 5.1; hu-HU; rv:1.7.12) Gecko/20050919 Firefox/1.0.7"
4.20.73.15 - - [4-03-2011:06:20:31 -0500] "GET / HTTP/1.1" 200 864 "http://www.adeveloper.com/resource.html" "Mozilla/5.0 (Windows; U; Windows NT 5.1; hu-HU; rv:1.7.12) Gecko/20050919 Firefox/1.0.7"
4.20.73.32 - - [4-03-2011:06:20:31 -0500] "GET / HTTP/1.1" 200 864 "http://www.adeveloper.com/resource.html" "Mozilla/5.0 (Windows; U; Windows NT 5.1; hu-HU; rv:1.7.12) Gecko/20050919 Firefox/1.0.7"

Splitting the server log line

Now, we will create a new Storm topology that will read the data from Kafka using the KafkaSpout spout, process the server logfiles, and store the process data in MySQL for further analysis.

In this section, we will write a bolt, ApacheLogSplitterBolt, which has logic to fetch the IP address, status code, referrer, bytes sent, and other such information from the server log line. We will create a new Maven project for this use case:

1. Create a new Maven project with com.learningstorm for groupId and stormlogprocessing for artifactId.

2. Add the following dependencies to the pom.xml file:

3.   **<!-- Dependency for Storm -->**

4.   **<dependency>**

5.   **<groupId>storm</groupId>**

6.   **<artifactId>storm-core</artifactId>**

7.   **<version>0.9.0.1</version>**

8.   **<scope>provided</scope>**

9.   **</dependency>**

10.   **<dependency>**

11.   **<groupId>com.google.guava</groupId>**

12.   **<artifactId>guava</artifactId>**

13.   **<version>15.0</version>**

14.   **</dependency>**

15.   **<dependency>**

16.   **<groupId>commons-collections</groupId>**

17.   **<artifactId>commons-collections</artifactId>**

18.   **<version>3.2.1</version>**

19.   **</dependency>**

20. Add the following repository to the pom.xml file:

21.   **<repository>**

22.   **<id>clojars.org</id>**

23.   **<url>http://clojars.org/repo</url>**

24.   **</repository>**

25. Create an ApacheLogSplitter class in the com.learningstorm.stormlogprocessing package and add the following content. This class contains logic to fetch different elements such as ip, referrer, user-agent, and so on from the Apache log line:

26.   /**

27.   * This class contains logic to Parse an Apache logfile

28.   * with Regular Expressions

29.   */

30.   public class ApacheLogSplitter {

31.

32.   public Map<String,Object> logSplitter(String apacheLog) {

33.

34.   String logEntryLine = apacheLog;

35.   // Regex pattern to split fetch

36.   // the different properties from log lines.

37.   String   logEntryPattern   =   "^([\\d.]+)   (\\S+)   (\\S+) \\[([\\w-:/]+\\s[+\\-]\\d{4})\\] \"(.+?)\" (\\d{3}) (\\d+) \"([^\"]+)\" \"([^\"]+)\"";

38.   Pattern p = Pattern.compile(logEntryPattern);

39.   Matcher matcher = p.matcher(logEntryLine);

40.   Map<String,Object> logMap = new HashMap<String, Object>();

```
41.        if (!matcher.matches() || 9 != matcher.groupCount()) {
42.            System.err.println("Bad log entry (or problem with RE?):");
43.            System.err.println(logEntryLine);
44.            return logMap;
45.        }
46.        // set the ip, dateTime, request, etc into map.
47.        logMap.put("ip", matcher.group(1));
48.        logMap.put("dateTime", matcher.group(4));
49.        logMap.put("request", matcher.group(5));
50.        logMap.put("response", matcher.group(6));
51.        logMap.put("bytesSent", matcher.group(7));
52.        logMap.put("referrer", matcher.group(8));
53.        logMap.put("useragent", matcher.group(9));
54.        return logMap;
       }
```

The input for the logSplitter(String apacheLog) method is as follows:

98.83.179.51 - - [18/May/2011:19:35:08 -0700] \"GET /css/main.css HTTP/1.1\" 200 1837 \"http://www.safesand.com/information.htm\" \"Mozilla/5.0 (Windows NT 6.0; WOW64; rv:2.0.1) Gecko/20100101 Firefox/4.0.1\"

The output of the logSplitter(String apacheLog) method is as follows:

{response=200, referrer=http://www.safesand.com/information.htm, bytesSent=1837, useragent=Mozilla/5.0 (Windows NT 6.0; WOW64; rv:2.0.1) Gecko/20100101 Firefox/4.0.1, dateTime=18/May/2011:19:35:08 -0700, request=GET /css/main.css HTTP/1.1, ip=98.83.179.51}

55. Now, we will create an ApacheLogSplitterBolt class in the com.learningstorm.stormlogprocessing package. The ApacheLogSplitterBolt class extends the backtype.storm.topology.base.BaseBasicBolt class. The execute() method of the ApacheLogSplitterBolt class receives the tuples (server log lines) from KafkaSpout. Then, it internally calls the logSplitter(String apachelog) method of the ApacheLogSplitter class to process the server log lines. After this, the process data is emitted to the next bolt in the topology. The following is the source code of the ApacheLogSplitterBolt class:

```
56.    /**
57.     * This class calls the ApacheLogSplitter class and
```

```java
58.    * passes the set of fields (ip, referrer, user-agent,
59.    * and so on) to the next bolt in the topology.
60.    */
61.
62.  public class ApacheLogSplitterBolt extends BaseBasicBolt {
63.
64.    private static final long serialVersionUID = 1L;
65.    // Create the instance of the ApacheLogSplitter class.
66.    private static final ApacheLogSplitter apacheLogSplitter = new ApacheLogSplitter();
67.    private static final List<String> LOG_ELEMENTS = new ArrayList<String>();
68.    static {
69.      LOG_ELEMENTS.add("ip");
70.      LOG_ELEMENTS.add("dateTime");
71.      LOG_ELEMENTS.add("request");
72.      LOG_ELEMENTS.add("response");
73.      LOG_ELEMENTS.add("bytesSent");
74.      LOG_ELEMENTS.add("referrer");
75.      LOG_ELEMENTS.add("useragent");
76.    }
77.
78.    public void execute(Tuple input, BasicOutputCollector collector) {
79.      // Get the Apache log from the tuple
80.      String log = input.getString(0);
81.
82.      if (StringUtils.isBlank(log)) {
83.        // Ignore blank lines
84.        return;
85.      }
86.      // Call the logSplitter(String apachelog) method // of the ApacheLogSplitter class.
87.      Map<String, Object> logMap = apacheLogSplitter.logSplitter(log);
88.      List<Object> logdata = new ArrayList<Object>();
89.      for (String element : LOG_ELEMENTS) {
90.        logdata.add(logMap.get(element));
91.      }
92.      // emits set of fields (ip, referrer, user-agent,// bytesSent, and so on.)
93.      collector.emit(logdata);
94.
95.    }
```

```
96.    public void declareOutputFields(OutputFieldsDeclarer declarer) {
97.        // Specify the name of output fields.
98.        declarer.declare(new Fields("ip", "dateTime", "request", "response", "bytesSent",
    "referrer", "useragent"));
99.    }
    }
```

The output of the ApacheLogSplitterBolt class contains seven fields. These fields are ip, dateTime, request, response, bytesSent, referrer, and useragent.

Identifying the country, the operating system type, and the browser type from the logfile

This section explains how you can identify a user's country name, the operating system type, and the browser type by analyzing the server log line. By identifying the country name, we can easily identify the locations from where our site is attracting more attention and where it is getting less attention. Let's perform the following steps to identify the country name, operating system type, and browser type from the Apache log line:

1. We will use the open source geoip library to identify the country name from the IP address. Add the following dependencies to the pom.xml file:
2.   <dependency>
3.   <groupId>org.geomind</groupId>
4.   <artifactId>geoip</artifactId>
5.   <version>1.2.8</version>
        </dependency>

6. Add the following repository to the pom.xml file:
7.   <repository>
8.   <id>geoip</id>
9.   <url>http://snambi.github.com/maven/</url>
        </repository>

10. We will create an IpToCountryConverter class in the com.learningstorm.stormlogprocessing package. This class contains the parameterized constructor that will take the location of the GeoLiteCity.dat file. You can find the GeoLiteCity.dat file in the Resources folder of the stormlogprocessing project. The

location of the GeoLiteCity.dat file must be the same in all Storm nodes. The GeoLiteCity.dat file is the database we will use to identify the country name when the IP address is given. The following is the source code of the IpToCountryConverter class:

```
11.  /**
12.   * This class contains logic to identify
13.   * the country name from the IP address
14.  */
15.  public class IpToCountryConverter {
16.
17.    private static LookupService cl = null;
18.
19.    /**
20.     * A parameterized constructor which would take
21.     * the location of the GeoLiteCity.dat file as input.
22.     *
23.     * @param pathTOGeoLiteCityFile
24.     */
25.    public IpToCountryConverter(String pathTOGeoLiteCityFile) {
26.      try {
27.        cl = new LookupService("pathTOGeoLiteCityFile",
28.            LookupService.GEOIP_MEMORY_CACHE);
29.      } catch (Exception exception) {
30.        throw new RuntimeException(
31.            "Error occurred while initializing IpToCountryConverter class: ");
32.      }
33.    }
34.
35.    /**
36.     * This method takes the IP address of the input and
37.     * converts it into a country name.
38.     *
39.     * @param ip
40.     * @return
41.     */
42.    public String ipToCountry (String ip) {
43.      Location location = cl.getLocation(ip);
44.      if (location == null) {
45.        return "NA";
```

```
46.        }
47.        if (location.countryName == null) {
48.            return "NA";
49.        }
50.        return location.countryName;
51.    }
   }
```

52. Now, download the UserAgentTools class from https://code.google.com/p/ndt/source/browse/branches/applet_91/Applet/src/main/java/edu/internet2/ndt/UserAgentTools.java?r=856.

    This class contains the logic to identify the operating system and the browser type from the user agent class. You can also find the UserAgentTools class in the stormlogprocessing project.

53. Let's write the UserInformationGetterBolt class to the com.learningstorm.stormlogprocessing package as follows. This bolt uses the UserAgentTools and IpToCountryConverter classes to identify the country name, the operating system type, and the browser type:

```
54.  /**
55.   * This class uses the IpToCountryConverter and
56.   * UserAgentTools classes to identify
57.   * the country, os, and browser from log line.
58.   */
59.  public class UserInformationGetterBolt extends BaseRichBolt {
60.
61.      private static final long serialVersionUID = 1L;
62.      private IpToCountryConverter ipToCountryConverter = null;
63.      private UserAgentTools userAgentTools = null;
64.      public OutputCollector collector;
65.      private String pathTOGeoLiteCityFile;
66.
67.      public UserInformationGetterBolt(String pathTOGeoLiteCityFile) {
68.          // set the path of the GeoLiteCity.dat file.
69.          this.pathTOGeoLiteCityFile = pathTOGeoLiteCityFile;
70.      }
71.
72.      public void declareOutputFields(OutputFieldsDeclarer declarer) {
```

```
73.      declarer.declare(new Fields("ip", "dateTime", "request", "response",
74.          "bytesSent", "referrer", "useragent", "country", "browser",
75.          "os"));
76.    }
77.    public void prepare(Map stormConf, TopologyContext context,
78.        OutputCollector collector) {
79.      this.collector = collector;
80.      this.ipToCountryConverter = new IpToCountryConverter(
81.          this.pathTOGeoLiteCityFile);
82.      this.userAgentTools = new UserAgentTools();
83.
84.    }
85.
86.    public void execute(Tuple input) {
87.
88.      String ip = input.getStringByField("ip").toString();
89.
90.      // Identify the country using the IP Address
91.      Object country = ipToCountryConverter.ipToCountry(ip);
92.
93.      // Identify the browser using useragent.
94.      Object browser = userAgentTools.getBrowser(input.getStringByField(
95.          "useragent").toString())[1];
96.
97.      // Identify the os using useragent.
98.      Object                            os                            =
    userAgentTools.getOS(input.getStringByField("useragent").toString())[1];
99.      collector.emit(new        Values(input.getString(0),        input.getString(1),
    input.getString(2),      input.getString(3),      input.getString(4),      input.getString(5),
    input.getString(6), country, browser, os));
100.
101.   }
    }
```

The output of the UserInformationGetterBolt class contains ten fields. These fields are ip, dateTime, request, response, bytesSent, referrer, useragent, country, browser, and os.

Extracting the searched keyword

This section explains how you can extract the searched keyword from the referrer URL. Suppose a referrer URL is https://www.google.co.in/#q=learning+storm. We will pass this referrer URL to our KeywordGenerator class and the output will be learning storm. By extracting the keyword to be searched, we can easily identify the search keyword that users are using to reach our site. Let's perform the following steps to extract the keyword from the referrer URL:

1. We will create a KeywordGenerator class in the com.learningstorm.stormlogprocessing package. This class contains the logic to generate the keyword from the referrer URL. The following is the source code of the KeywordGenerator class:

```
2.    /**
3.      * This class takes the referrer URL as the input,
4.      * analyzes the URL and returns the
5.      * keyword to be searched as the output.
6.    */
7.    public class KeywordGenerator {
8.       public String getKeyword(String referer) {
9.
10.          String[] temp;
11.          Pattern pat = Pattern.compile("[?&#]q=([^&]+)");
12.          Matcher m = pat.matcher(referer);
13.          if (m.find()) {
14.             String searchTerm = null;
15.             searchTerm = m.group(1);
16.             temp = searchTerm.split("\\+");
17.             searchTerm = temp[0];
18.             for (int i = 1; i < temp.length; i++) {
19.                searchTerm = searchTerm + " " + temp[i];
20.             }
21.             return searchTerm;
22.          } else {
23.             pat = Pattern.compile("[?&#]p=([^&]+)");
24.             m = pat.matcher(referer);
25.             if (m.find()) {
26.                String searchTerm = null;
27.                searchTerm = m.group(1);
28.                temp = searchTerm.split("\\+");
29.                searchTerm = temp[0];
30.                for (int i = 1; i < temp.length; i++) {
31.                   searchTerm = searchTerm + " " + temp[i];
```

```java
32.          }
33.          return searchTerm;
34.      } else {
35.          //
36.          pat = Pattern.compile("[?&#]query=([^&]+)");
37.          m = pat.matcher(referer);
38.          if (m.find()) {
39.              String searchTerm = null;
40.              searchTerm = m.group(1);
41.              temp = searchTerm.split("\\+");
42.              searchTerm = temp[0];
43.              for (int i = 1; i < temp.length; i++) {
44.                  searchTerm = searchTerm + " " + temp[i];
45.              }
46.              return searchTerm;
47.          } else {
48.              return "NA";
49.          }
50.      }
51.  }
52. }
  }
```

The input for the KeywordGenerator class is as follows:

https://in.search.yahoo.com/search;_ylt=AqH0NZe1hgPCzVap0PdKk7GuitI F?p=india+live+score&toggle=1&cop=mss&ei=UTF-8&fr=yfp-t-704

Then, the output of the KeywordGenerator class is as follows:

india live score

53. We will create a KeyWordIdentifierBolt class in the com.learningstorm.stormlogprocessing package. This class calls the KeywordGenerator class that extracts the keyword from the referrer URL. The following is the source code of the KeyWordIdentifierBolt class:

```java
54. /**
55.  * This class uses the KeywordGenerator class
56.  * to extract the keyword from the referrer URL.
57.  */
```

```
58.    public class KeyWordIdentifierBolt extends BaseRichBolt {
59.
60.       private static final long serialVersionUID = 1L;
61.       private KeywordGenerator keywordGenerator = null;
62.       public OutputCollector collector;
63.
64.       public KeyWordIdentifierBolt() {
65.       }
66.
67.       public void declareOutputFields(OutputFieldsDeclarer declarer) {
68.          declarer.declare(new Fields("ip", "dateTime", "request", "response", "bytesSent",
   "referrer", "useragent", "country", "browser", "os", "keyword"));
69.       }
70.
71.       public void prepare(Map stormConf, TopologyContext context, OutputCollector
   collector) {
72.          this.collector = collector;
73.          this.keywordGenerator = new KeywordGenerator();
74.
75.       }
76.
77.       public void execute(Tuple input) {
78.
79.          String referrer = input.getStringByField("referrer").toString();
80.          // Call the getKeyword(String referrer) method
81.          // of the KeywordGenerator class to
82.          // extract the keyword.
83.          Object keyword = keywordGenerator.getKeyword(referrer);
84.          // emits all the field emitted by previous bolt +
85.          // the keyword
86.          collector.emit(new            Values(input.getString(0),          input.getString(1),
   input.getString(2),       input.getString(3),       input.getString(4),       input.getString(5),
   input.getString(6), input.getString(7), input.getString(8), input.getString(9), keyword));
87.
88.       }
   }
```

The output of the KeyWordIdentifierBolt class contains 11 fields. These fields are ip, dateTime, request, response, bytesSent, referrer, useragent, country, browser, os, and keyword.

Persisting the process data

This section will explain how you can persist the process data to the data store. We are using MySQL as the data store for storing the processed data in this use case.

We will assume that you have MySQL installed on your CentOS machine, or you can follow the blog at http://www.rackspace.com/knowledge_center/article/installing-mysql-server-on-centos to install MySQL on a CentOS machine. Let's perform the following steps to persist records to MySQL:

1. Add the following dependency to the pom.xml file of the stormlogprocessing project:
2.   `<dependency>`
3.   `<groupId>mysql</groupId>`
4.   `<artifactId>mysql-connector-java</artifactId>`
5.   `<version>5.1.6</version>`
      `</dependency>`

6. We will create a MySQLConnection class in the com.learningstorm.stormlogprocessing package. This class contains the getMySQLConnection(String ip, String database, String user, String password) function, which returns the MySQL connection. The following is the source code of the MySQLConnection class:
7.   `/**`
8.   `* This class returns the MySQL connection.`
9.   `*/`
10. `public class MySQLConnection {`
11.
12.   `private static Connection connect = null;`
13.
14.   `/**`
15.   `* This method returns the MySQL connection.`
16.   `*`
17.   `* @param ip`
18.   `*`   `IP address of the MySQL server`
19.   `* @param database`

20.   *   name of the database
21.   * @param user
22.   *   name of the user
23.   * @param password
24.   *   password of the given user
25.   * @return MySQL connection
26.   */
27.   public static Connection getMySQLConnection(String ip, String database, String user, String password) {
28.   try {
29.   // This will load the MySQL driver,
30.   // each DB has its own driver
31.   Class.forName("com.mysql.jdbc.Driver");
32.   // Set up the connection with the DB.
33.   connect   =   DriverManager.getConnection("jdbc:mysql://"+   ip +"/"+database+"?"+"user="+user+"&password="+password+"");
34.   return connect;
35.   } catch (Exception e) {
36.   throw new RuntimeException("Error occurred while getting the MySQL connection: ");
37.   }
38.   }
   }

39. Now, we will create a MySQLDump class in the com.learningstorm.stormlogprocessing package. This class has a parameterized constructor that will take the IP address, the database name, the user name, and the password of the MySQL server as arguments. This class calls the getMySQLConnection(ip,database,user,password) method of the MySQLConnection class to get the MySQL connection. The MySQLDump class contains the persistRecord(Tuple tuple) method, and this method persists the tuples into MySQL. The following is the source code of the MySQLDump class:

40. /**
41.   * This class contains logic to persist the record
42.   * into the MySQL database.
43.   */
44. public class MySQLDump {
45.   /**
46.   * Name of database you want to connect
47.   */

```java
48.    private String database;
49.    /**
50.     * Name of MySQL user
51.     */
52.    private String user;
53.    /**
54.     * IP of the MySQL server
55.     */
56.    private String ip;
57.    /**
58.     * Password of the MySQL server
59.     */
60.    private String password;
61.
62.    public MySQLDump(String ip, String database, String user, String password) {
63.        this.ip = ip;
64.        this.database = database;
65.        this.user = user;
66.        this.password = password;
67.    }
68.
69.    /**
70.     * Get the MySQL connection
71.     */
72.    private          Connection          connect          =
    MySQLConnection.getMySQLConnection(ip,database,user,password);
73.
74.    private PreparedStatement preparedStatement = null;
75.
76.    /**
77.     * Persist input tuple.
78.     * @param tuple
79.     */
80.    public void persistRecord(Tuple tuple) {
81.        try {
82.
83.            // preparedStatements can use variables and
84.            // are more efficient
```

```
85.        preparedStatement = connect.prepareStatement("insert into   apachelog values
      (default, ?, ?, ?,?, ?, ?, ?, ? , ?, ?, ?)");
86.
87.        preparedStatement.setString(1, tuple. getStringByField("ip"));
88.        preparedStatement.setString(2, tuple.getStringByField("dateTime"));
89.        preparedStatement.setString(3, tuple.getStringByField("request"));
90.        preparedStatement.setString(4, tuple.getStringByField("response"));
91.        preparedStatement.setString(5, tuple.getStringByField("bytesSent"));
92.        preparedStatement.setString(6, tuple.getStringByField("referrer"));
93.        preparedStatement.setString(7, tuple.getStringByField("useragent"));
94.        preparedStatement.setString(8, tuple.getStringByField("country"));
95.        preparedStatement.setString(9, tuple.getStringByField("browser"));
96.        preparedStatement.setString(10, tuple.getStringByField("os"));
97.        preparedStatement.setString(11, tuple.getStringByField("keyword"));
98.
99.        // Insert record
100.       preparedStatement.executeUpdate();
101.
102.     } catch (Exception e) {
103.       throw new RuntimeException("Error occurred while persisting records in
      MySQL: ");
104.     } finally {
105.       // close prepared statement
106.       if (preparedStatement != null) {
107.         try {
108.           preparedStatement.close();
109.         } catch (Exception exception) {
110.           System.out.println("Error occurred while closing PreparedStatement:");
111.         }
112.       }
113.     }
114.
115.   }
116.   public void close() {
117.     try {
118.     connect.close();
119.     }catch(Exception exception) {
120.       System.out.println("Error occurred while closing the connection");
```

121.   }
122.   }
    }

123. Let's create a PersistenceBolt class in the com.learningstorm.stormlogprocessing package. This class implements the bolt, backtype.storm.topology.IBasicBolt. The PersistenceBolt class has a parameterized constructor that will take the IP address, the database name, the user name, and password of the MySQL server as arguments. The execute() method of the PersistenceBolt class calls the persistRecord(Tuple tuple) method of the MySQLDump class to persist the record into MySQL. The following is the source code of the PersistenceBolt class:

124. /**
125.   * This Bolt calls the getConnectionn(....) method
126.   * of the MySQLDump class to persist
127.   * the record into the MySQL database.
128.   *
129.   * @author Admin
130.   *
131.   */
132. public class PersistenceBolt implements IBasicBolt {
133.
134.   private MySQLDump mySQLDump = null;
135.   private static final long serialVersionUID = 1L;
136.   /**
137.   * Name of the database you want to connect
138.   */
139.   private String database;
140.   /**
141.   * Name of the MySQL user
142.   */
143.   private String user;
144.   /**
145.   * IP address of the MySQL server
146.   */
147.   private String ip;
148.   /**
149.   * Password of the MySQL server
150.   */
151.   private String password;

```
152.
153.    public PersistenceBolt(String ip, String database, String user, String password) {
154.        this.ip = ip;
155.        this.database = database;
156.        this.user = user;
157.        this.password = password;
158.    }
159.
160.    public void declareOutputFields(OutputFieldsDeclarer declarer) {
161.    }
162.
163.    public Map<String, Object> getComponentConfiguration() {
164.        return null;
165.    }
166.
167.    public void prepare(Map stormConf, TopologyContext context) {
168.
169.        // create the instance of the MySQLDump(....) class.
170.        mySQLDump = new MySQLDump(ip, database, user, password);
171.    }
172.    /**
173.     * This method calls the persistRecord(input) method
174.     * of the MySQLDump class to persist records into MySQL.
175.     */
176.    public void execute(Tuple input, BasicOutputCollector collector) {
177.        System.out.println("Input tuple : " + input);
178.        mySQLDump.persistRecord(input);
179.    }
180.
181.    public void cleanup() {
182.        // Close the connection
183.        mySQLDump.close();
184.    }
185.
    }
```

In this section, we covered how to insert the input tuples into the data store.

Defining a topology and the Kafka spout

This section will explain how you can read the server log from a Kafka topic. We will use the Kafka spout integration available on GitHub at https://github.com/wurstmeister/storm-kafka-0.8-plus for consuming the data from Kafka. This section also defines the LogProcessingTopology topology that will chain together all the bolts created in the preceding sections. Let's perform the following steps to consume the data from Kafka and define a topology:

1. Add the following dependency and repository for Kafka in pom.xml:
2.   `<dependency>`
3.   `<groupId>net.wurstmeister.storm</groupId>`
4.   `<artifactId>storm-kafka-0.8-plus</artifactId>`
5.   `<version>0.4.0</version>`
      `</dependency>`

6. Add the following build plugins to pom.xml. These plugins will let us execute LogProcessingTopology using Maven:
7.   `<build>`
8.   `<plugins>`
9.   `<plugin>`
10.   `<groupId>org.apache.maven.plugins</groupId>`
11.   `<artifactId>maven-compiler-plugin</artifactId>`
12.   `<version>2.5.1</version>`
13.   `<configuration>`
14.   `<source>1.6</source>`
15.   `<target>1.6</target>`
16.   `</configuration>`
17.   `</plugin>`
18.
19.   `<plugin>`
20.   `<artifactId>maven-assembly-plugin</artifactId>`
21.   `<version>2.2.1</version>`
22.   `<configuration>`
23.   `<descriptorRefs>`
24.   `<descriptorRef>jar-with-dependencies`
25.   `</descriptorRef>`
26.   `</descriptorRefs>`
27.   `<archive>`

```
28.          <manifest>
29.             <mainClass />
30.          </manifest>
31.        </archive>
32.      </configuration>
33.      <executions>
34.        <execution>
35.          <id>make-assembly</id>
36.          <phase>package</phase>
37.          <goals>
38.            <goal>single</goal>
39.          </goals>
40.        </execution>
41.      </executions>
42.    </plugin>
43.  </plugins>
   </build>
```

44. Let's create a LogProcessingTopology class in the com.learningstorm.stormlogprocessing package. This class uses the backtype.storm.topology.TopologyBuilder class to define the topology. The following is the source code of the LogProcessingTopology class with its explanation:

```
45.  public class LogProcessingTopology {
46.    public static void main(String[] args) throws Exception {
47.
48.      // zookeeper hosts for the Kafka cluster
49.      ZkHosts zkHosts = new ZkHosts("localhost:2181");
50.
51.      // Create the KafkaSpout configuartion
52.      // Second argument is the topic name
53.      // Third argument is the zookeeper root for Kafka
54.      // Fourth argument is consumer group id
55.      SpoutConfig kafkaConfig = new SpoutConfig(zkHosts, "apache_log", "", "id");
56.
57.      // Specify that the kafka messages are String
58.      kafkaConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
59.
60.      // We want to consume all the first messages
61.      // in the topic every time we run the topology
```

```
62.        // to help in debugging. In production, this
63.        // property should be false
64.        kafkaConfig.forceFromStart = true;
65.
66.        // Now we create the topology
67.        TopologyBuilder builder = new TopologyBuilder();
68.
69.        // set the kafka spout class
70.        builder.setSpout("KafkaSpout", new KafkaSpout(kafkaConfig), 1);
71.
72.        // set the LogSplitter, IpToCountry, Keyword,
73.        // and PersistenceBolt bolts
74.        // class.
75.        builder.setBolt("LogSplitter", new ApacheLogSplitterBolt(), 1)
76.            .globalGrouping("KafkaSpout");
77.        builder.setBolt("IpToCountry",                              new
    UserInformationGetterBolt("./src/main/resources/GeoLiteCity.dat"), 1)
78.            .globalGrouping("LogSplitter");
79.        builder.setBolt("Keyword",         new         KeyWordIdentifierBolt(),
    1).globalGrouping("IpToCountry");
80.        builder.setBolt("PersistenceBolt", new PersistenceBolt("localhost", "apachelog",
    "root", "root"), 1).globalGrouping("Keyword");
81.
82.    if (args != null && args.length > 0) {
83.        // Run the topology on remote cluster.
84.        Config conf = new Config();
85.        conf.setNumWorkers(4);
86.        try {
87.            StormSubmitter.submitTopology(args[0], conf, builder.createTopology());
88.        } catch (AlreadyAliveException alreadyAliveException) {
89.            System.out.println(alreadyAliveException);
90.        } catch (InvalidTopologyException invalidTopologyException) {
91.            System.out.println(invalidTopologyException);
92.        }
93.    } else {
94.        // create an instance of the LocalCluster class
95.        // for executing the topology in the local mode.
96.        LocalCluster cluster = new LocalCluster();
```

```
97.         Config conf = new Config();
98.
99.         // Submit topology for execution
100.        cluster.submitTopology("KafkaToplogy", conf, builder.createTopology());
101.
102.        try {
103.          // Wait for some time before exiting
104.          System.out.println("*********************Waiting to consume from kafka");
105.          Thread.sleep(10000);
106.
107.        } catch (Exception exception) {
108.          System.out.println("*****************Thread interrupted exception : " +
      exception);
109.        }
110.
111.        // kill KafkaTopology
112.        cluster.killTopology("KafkaToplogy");
113.
114.        // shut down the storm test cluster
115.        cluster.shutdown();
116.
117.      }
118.
119.    }
      }
```

This section covered how to chain the different types of bolts into a topology. In addition to this, we covered how to consume the data from Kafka. In the next section, we will learn how to deploy the topology.

Deploying a topology

This section will explain how you can deploy the LogProcessingTopology topology. To deploy this topology, perform the following steps:

1. Execute the following command on the MySQL console to define a database schema:
2. **create database apachelog;**
3. **use apachelog;**
4. **create table apachelog(**

5.   **id INT NOT NULL AUTO_INCREMENT,**

6.   **ip VARCHAR(100) NOT NULL,**

7.   **dateTime VARCHAR(200) NOT NULL,**

8.   **request VARCHAR(100) NOT NULL,**

9.   **response VARCHAR(200) NOT NULL,**

10.   **bytesSent VARCHAR(200) NOT NULL,**

11.   **referrer VARCHAR(500) NOT NULL,**

12.   **useragent VARCHAR(500) NOT NULL,**

13.   **country VARCHAR(200) NOT NULL,**

14.   **browser VARCHAR(200) NOT NULL,**

15.   **os VARCHAR(200) NOT NULL,**

16.   **keyword VARCHAR(200) NOT NULL,**

17.   **PRIMARY KEY (id)**

18.   **);**

19. Before running the log-processing use cases, we need to produce some data in Kafka using the KafkaLogProducer project, which was created at the start of this chapter.

20. Go to the home directory of the stormlogprocessing project and run the following command to build the project:

21.   **mvn clean install −DskipTests**

22. Execute the following command to start the log-processing topology in the local mode:

23.   **java   -cp target/stormlogprocessing-0.0.1-SNAPSHOT-jar-with-dependencies.jar:$ST ORM_HOME/storm-core-0.9.0.1.jar:$STORM_HOME/lib/* com.learningstorm.stormlogprocessing.LogProcessingTopology /path/to/GeoLiteCity.dat localhost apachelog root root**

24. Now, go to the MySQL console and check out the rows in the apachelog table.

25.   **select * from apachelog limit 2;**

The following screenshot shows the data in the apachelog table:

```
+----+----------------+-------------------------+----------------+----------+-----------+
| id | ip             | dateTime                | request        | response | bytesSent |
+----+----------------+-------------------------+----------------+----------+-----------+
|  1 | 24.25.135.19   | 1-01-2011:06:20:31 -0500 | GET / HTTP/1.1 | 200      | 864       |
|  2 | 180.183.50.208 | 1-01-2011:06:20:31 -0500 | GET / HTTP/1.1 | 200      | 864       |
+----+----------------+-------------------------+----------------+----------+-----------+
```

In this section, we covered how to deploy the log-processing topology. The next section will explain how you can generate statistics from the data stored in MySQL.

MySQL queries

This section will explain how you can analyze or query the stored data to generate some statistics. We will cover the following types of statistics:

- How to calculate the page hits from each country
- How to calculate the count of each browser
- How to calculate the count of each operating system

Calculating the page hits from each country

Run the following command on the MySQL console to calculate the number of hits on a page from each country:

select country, count(*) from apachelog group by country;

The output for the preceding command is as follows:

```
+--------------------------+----------+
| country                  | count(*) |
+--------------------------+----------+
| Asia/Pacific Region      |        9 |
| Belarus                  |       12 |
| Belgium                  |       12 |
| Bosnia and Herzegovina   |       12 |
| Brazil                   |       36 |
| Bulgaria                 |       12 |
| Canada                   |      218 |
| Europe                   |       24 |
| France                   |       44 |
| Germany                  |       48 |
| Greece                   |       12 |
| Hungary                  |       12 |
| India                    |      144 |
| Indonesia                |       60 |
| Iran, Islamic Republic of |      12 |
| Italy                    |       24 |
| Japan                    |       12 |
| Malaysia                 |       12 |
| Mexico                   |       36 |
| NA                       |       10 |
```

```
| Nepal                    |         24 |
| Netherlands              |        164 |
| Nigeria                  |         24 |
| Puerto Rico              |         72 |
| Russian Federation       |         60 |
| Singapore                |        165 |
| Spain                    |         48 |
| Sri Lanka                |         12 |
| Switzerland              |          7 |
| Taiwan                   |         12 |
| Thailand                 |         12 |
| Ukraine                  |         12 |
| United Kingdom           |         48 |
| United States            |       5367 |
| Vietnam                  |         12 |
| Virgin Islands, U.S.     |        129 |
+--------------------------+----------+
36 rows in set (0.08 sec)
```

Calculating the count for each browser

Run the following command on the MySQL console to calculate the count for each browser:

select browser, count(*) from apachelog group by browser;

The output for the preceding command is as follows:

```
+----------------+----------+
| browser        | count(*) |
+----------------+----------+
| Gecko(Firefox) |     6929 |
+----------------+----------+
1 row in set (0.00 sec)
```

Calculating the count for each operating system

Run the following command on the MySQL console to calculate the count for each operating system:

select os,count(*) from apachelog group by os;

The output for the preceding command is as follows:

```
+-------+----------+
| os    | count(*) |
+-------+----------+
| WinXP |     6929 |
+-------+----------+
1 row in set (0.00 sec)
```

Summary

In this chapter, we learned how to process the Apache logfile, how to identify the country name from the IP address, how to identify a user's operating system and browser by analyzing the logfile, and how to extract the searched keyword by analyzing the referrer URL.

In the next chapter, we will learn how to solve the machine learning problem through Storm.

Chapter 9. Machine Learning

In the previous chapter, you learned how to create a log processing application with Storm and Kafka.

In this chapter, we will cover another important use case of Storm—machine learning.

The following are the major topics covered in this chapter:

- Introduction to machine learning
- Introduction to Trident-ML
- Introduction to the case study
- Producing training dataset into Kafka
- Building a Trident topology to build the clustering model
- Predicting the cluster for the test data

Exploring machine learning

Machine learning is a branch of applied computer science in which we build models of real-world phenomena on the basis of existing data available for analysis, and then using that model, we predict certain characteristics of data never seen before by the model. Machine learning techniques are one of the important ways in which decisions are made in applications. As most of the applications operate in real time, using machine learning with Storm is a great way to implement decision making in real-time applications.

Graphically, the process of machine learning can be represented by the following diagram:
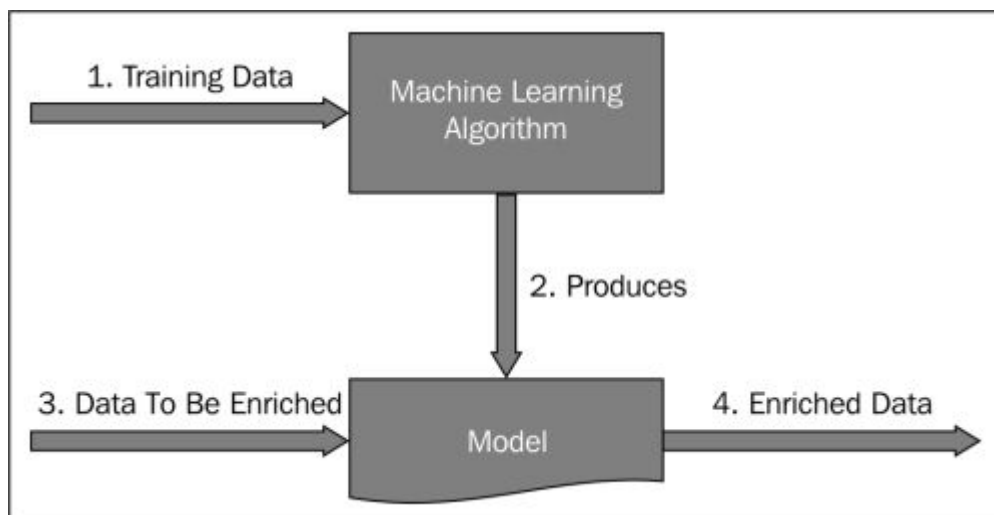
Machine learning

The process of building the model from data is called **training** in the machine learning terminology. Training can happen in real time on a stream of data or can also be done on historical data. When the training is done in real time, the model evolves over time with the changed data. This kind of learning is referred to as **online** learning, and when the model is updated every once in a while by running the training algorithm on a new dataset, it is called **offline** learning.

When we discuss machine learning in the context of Storm, more often than not we are discussing online learning algorithms.

The following are some of the real-world applications on machine learning:

- Online ad optimization
- New article clustering
- Spam detection
- Computer vision
- Sentiment analysis

Using Trident-ML

We introduced Trident in [Chapter 5](#), *Exploring High-level Abstraction in Storm with Trident*, of this book. Trident-ML (GitHub repository: https://github.com/pmerienne/trident-ml) is an online machine-learning library written over Trident that can be used to implement machine-learning algorithms in Storm applications.

It supports the following algorithms out of the box:

- Linear classification
- Linear regression
- K-means clustering
- Feature normalization
- Text feature extraction
- Stream statistics (count, mean, variance, and standard deviation)

If the algorithm you are looking for is not implemented in Trident-ML, you can easily implement it. Trident-ML also comes with a very useful pretrained Twitter sentiment analyzer.

In Trident-ML, various parameters associated with the learned model is stored in a TridentState object. As more training data comes in, these model parameters can be updated. This TridentState object is then used in a DRPC server to retrieve the model parameters to compute or predict new features of the incoming data and enrich the stream to process further.

The following diagram illustrates a typical Trident-ML application:
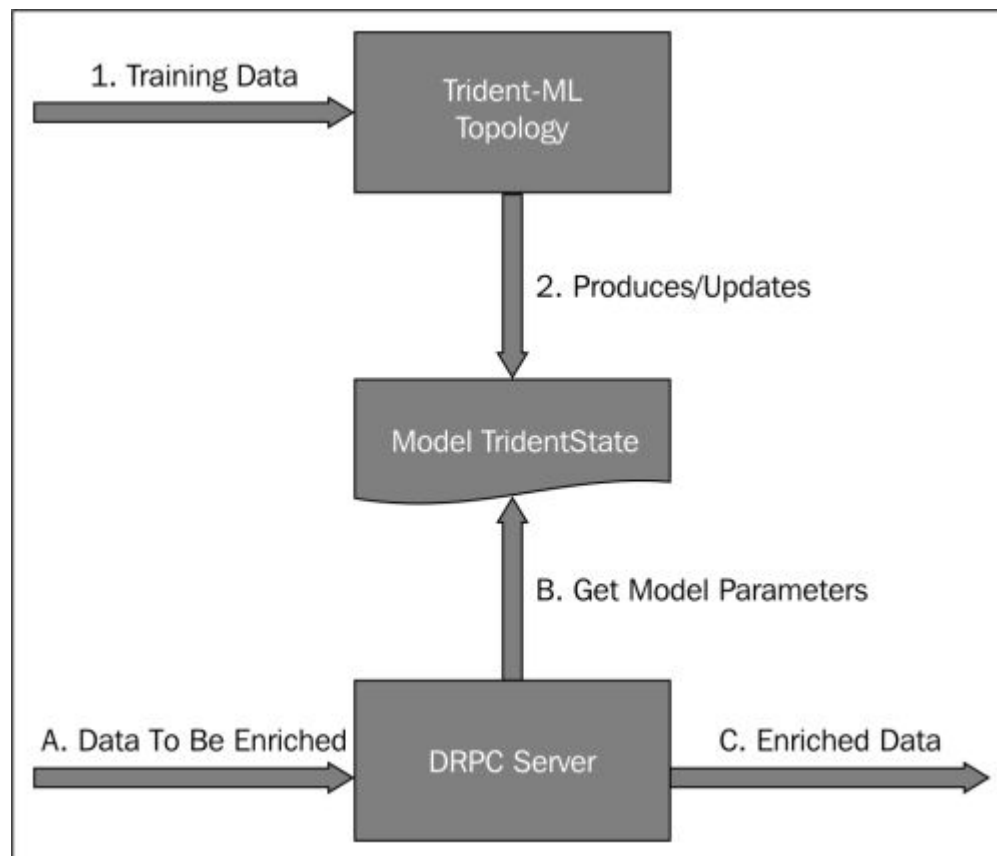
The Trident-ML application

Next, we will look into the use case that we will be developing for in this chapter.

The use case – clustering synthetic control data

A control chart represents how a system behaves over time. It is a graph that plots one or more variables of a system or process over time. This information can be used for quality control in manufacturing and business process. When only one variable is plotted against time, it is called a **univariate** control chart, and when more than one variable is plotted against time, it is called a **multivariate** control chart.

In this chapter, we will be working with a synthetic control chart time series data provided by the UCI Machine Learning Repository. Each of the control chart belongs to one of the following categories:

- Normal
- Cyclic
- Increasing trend
- Decreasing trend
- Upward shift
- Downward shift

Each of the control charts consists of 60 columns, each a decimal value. There are 100 records for each category. Further details about the dataset can be found at http://archive.ics.uci.edu/ml/databases/synthetic_control/synthetic_control.data.html.

We will be using 80 out of 100 records from each category to develop a clustering model, and then we will use the remaining 20 records to predict the category for them. We will be using the K-means clustering algorithm for this, which is provided by Trident-ML.

But before going ahead with the producer, we need to download the dataset from the UCI Machine Learning Repository located at http://archive.ics.uci.edu/ml/databases/synthetic_control/synthetic_control.data. Save this file so that it can be used later for training and testing.

Producing a training dataset into Kafka

The first step while developing a machine-learning pipeline is to get the data in a place from where we can feed it to the training algorithm. In this case study, we will be using Kafka as the source of the training data.

For this, we will be writing a Kafka producer that will stream 80 percent of the data in the data file to the Kafka broker. The remaining 20 percent of the data will be stored in a file, which we will use to test our clustering model created by our topology.

We will be creating a Maven project for publishing data into Kafka. The following are the steps for creating the producer:

1. Create a new Maven project with the com.learningstorm group ID and the ml-kafka-producer artifact ID.
2. Add the following dependencies for Kafka in the pom.xml file:

```
3.    <!-- Apache Kafka Dependency -->
4.    <dependency>
5.      <groupId>org.apache.kafka</groupId>
6.      <artifactId>kafka_2.10</artifactId>
7.      <version>0.8.0</version>
8.      <exclusions>
9.        <exclusion>
10.          <groupId>com.sun.jdmk</groupId>
11.          <artifactId>jmxtools</artifactId>
12.        </exclusion>
13.        <exclusion>
14.          <groupId>com.sun.jmx</groupId>
15.          <artifactId>jmxri</artifactId>
16.        </exclusion>
17.      </exclusions>
      </dependency>
```

18. Add the following build plugins to the pom.xml file. It will allow us to execute the producer using Maven:

```
19.    <plugin>
20.      <groupId>org.codehaus.mojo</groupId>
21.      <artifactId>exec-maven-plugin</artifactId>
22.      <version>1.2.1</version>
23.      <executions>
24.        <execution>
25.          <goals>
26.            <goal>exec</goal>
27.          </goals>
28.        </execution>
29.      </executions>
30.      <configuration>
31.        <executable>java</executable>
32.        <includeProjectDependencies>true</includeProjectDependencies>
33.        <includePluginDependencies>false</includePluginDependencies>
34.        <classpathScope>compile</classpathScope>
35.        <mainClass>com.learningstorm.ml.kafka.KafkaProducer</mainClass>
36.      </configuration>
      </plugin>
```

37. Now, we will create the com.learningstorm.ml.kafka.KafkaProducer class that reads the input dataset and produces 80 percent of the data into Kafka to train the model and the remaining data in a file that will be used for predictions later. The following is the code of the KafkaProducer class:

38.   public class KafkaProducer {

39.

40.   public static void main(String[] args) throws IOException {

41.

42.   // Build the configuration required for connecting to Kafka

43.   Properties props = new Properties();

44.

45.   // List of kafka brokers.

46.   props.put("metadata.broker.list", "localhost:9092");

47.

48.   // Serializer used for sending data to kafka. //Since we are sending

49.   // strings, we are using StringEncoder.

50.   props.put("serializer.class", "kafka.serializer.StringEncoder");

51.

52.   // We want acks from Kafka that messages are properly received.

53.   props.put("request.required.acks", "1");

54.

55.   // Create the producer instance

56.   ProducerConfig config = new ProducerConfig(props);

57.   Producer<String, String> producer = new Producer<String, String>(config);

58.

59.   // This is the input file. This should be the path to the file downloaded

60.   // from UIC Machine Learning Repository at

61.   //
    http://archive.ics.uci.edu/ml/databases/synthetic_control/synthetic_control.data

62.   File file = new File("/home/anand/Desktop/synthetic_control.data");

63.   Scanner scanner = new Scanner(file);

64.

65.   // This is the output file for prediction data. Change it to something

66.   // appropiate for your setup

67.   File predictioFile = new File("/home/anand/Desktop/prediction.data");

68.   BufferedWriter writer = new BufferedWriter(new FileWriter(predictioFile));

69.

70.   int i = 0;

```
71.        while(scanner.hasNextLine()){
72.          String instance = scanner.nextLine();
73.          if(i++ % 5 == 0){
74.            // write to file
75.            writer.write(instance+"\n");
76.          } else {
77.            // produce to kafka
78.            KeyedMessage<String,  String>  data  =  new  KeyedMessage<String,
    String>("training", instance);
79.            producer.send(data);
80.          }
81.        }
82.
83.        // close the files
84.        scanner.close();
85.        writer.close();
86.
87.        // close the producer
88.        producer.close();
89.
90.        System.out.println("Produced data");
91.      }
    }
```

92. Now that the producer is ready, make sure Kafka is running on your system.

93. Now, run the producer with the following command:

94. **mvn exec:java**

The following output is displayed:

[INFO] --
[INFO] Building ml-kafka-producer 0.0.1-SNAPSHOT
[INFO] --
[INFO]
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ ml-kafka-producer ---
Produced data

95. Now, let's verify that the data has been produced into Kafka by executing the following command and verifying that the topic has been created:

96. **./bin/kafka-list-topic.sh   --zookeeper localhost:2181**

The following output is displayed:

topic: training   partition: 0   leader: 0   replicas: 0   isr: 0

97. The file that will be used for prediction should also be generated at the path given in the class. Please verify that it exists.

Building a Trident topology to build the clustering model

Now that we have the data to be used to train and predict in place, we will develop the Trident topology using the Trident-ML library.

Again, we will create a Maven project to implement our topology. The following are the steps to create this project:

1. Create a new Maven project with the com.learningstorm group ID and the ml artifact ID.
2. Add the following dependencies for Kafka in the pom.xml file:
3.   <!-- Dependency for Storm -->
4.   <dependency>
5.   <groupId>storm</groupId>
6.   <artifactId>storm-core</artifactId>
7.   <version>0.9.0.1</version>
8.   <scope>provided</scope>
9.   </dependency>
10.
11.   <!-- Dependency for Storm-Kafka spout -->
12.   <dependency>
13.   <groupId>net.wurstmeister.storm</groupId>
14.   <artifactId>storm-kafka-0.8-plus</artifactId>
15.   <version>0.4.0</version>
16.   </dependency>
17.
18.   <!-- Dependency for Trident-ML -->
19.   <dependency>
20.   <groupId>com.github.pmerienne</groupId>
21.   <artifactId>trident-ml</artifactId>

22.   `<version>0.0.4</version>`

    `</dependency>`

23. Add the following repository in the pom.xml file:

24. `<repository>`

25.   `<id>clojars.org</id>`

26.   `<url>http://clojars.org/repo</url>`

    `</repository>`

27. Add the following build plugins to the pom.xml file. It will allow us to execute the Trident topology in the local mode using Maven:

28. `<plugin>`

29.   `<groupId>org.codehaus.mojo</groupId>`

30.   `<artifactId>exec-maven-plugin</artifactId>`

31.   `<version>1.2.1</version>`

32.   `<executions>`

33.   `<execution>`

34.   `<goals>`

35.   `<goal>exec</goal>`

36.   `</goals>`

37.   `</execution>`

38.   `</executions>`

39.   `<configuration>`

40.   `<executable>java</executable>`

41.   `<includeProjectDependencies>true</includeProjectDependencies>`

42.   `<includePluginDependencies>false</includePluginDependencies>`

43.   `<classpathScope>compile</classpathScope>`

44.   `<mainClass>com.learningstorm.ml.TridentMLTopology</mainClass>`

45.   `</configuration>`

    `</plugin>`

46. The Trident-ML library takes the input—for both model building and later prediction—as objects of the com.github.pmerienne.trident.ml.core.Instance class. Let's create the com.learningstorm.ml.FeaturesToValues class that will convert the first string from the tuple into an Instance object. It will split the string on space character and convert each number into a double value to create an Instance object. The following is the code for the FeaturesToValues class:

47. `public class FeaturesToValues extends BaseFunction {`

48.

```java
49.    @SuppressWarnings("rawtypes")
50.    public void execute(TridentTuple tuple, TridentCollector collector) {
51.        // get the input string
52.        String line = tuple.getString(0);
53.
54.        double[] features = new double[60];
55.
56.        // split the input string and iterate over them and covert to double
57.        String[] featureList = line.split("\\s+");
58.        for(int i = 0; i < features.length; i++){
59.            features[i] = Double.parseDouble(featureList[i]);
60.        }
61.
62.        // emit the Instance object with the features from given input string
63.        collector.emit(new Values(new Instance(features)));
64.    }
    }
```

65. Now, we will create the Trident topology that will create the K-means clustering model and will also expose this model as a DRPC call so that the model can be used to predict the class for the test data. Create the com.learningstorm.ml.TridentMLTopology class with the following code:

```java
66.    public class TridentMLTopology {
67.
68.    public static void main(String[] args) throws InterruptedException, IOException {
69.        // Kafka Spout definition
70.        // Specify the zk hosts for Kafka, change as needed
71.        BrokerHosts brokerHosts = new ZkHosts("localhost:2181");
72.
73.        // Specify the topic name for training and the client id
74.        // here topic name is 'training' and client id is 'storm'
75.        TridentKafkaConfig    kafkaConfig    =    new    TridentKafkaConfig(brokerHosts,
    "training", "storm");
76.
77.        // We will always consume from start so that we can run the topology multiple
    times while debugging. In production, this should be false.
78.        kafkaConfig.forceFromStart = true;
79.
80.        // We have string data in the kafka, so specify string scheme here
81.        kafkaConfig.scheme = new SchemeAsMultiScheme(new StringScheme());
```

```
82.
83.        // Define the spout for reading from kafka
84.        TransactionalTridentKafkaSpout          kafkaSpout          =          new
   TransactionalTridentKafkaSpout(kafkaConfig);
85.
86.        // Topology definition
87.        // now we will define the topology that will build the clustering model
88.        TridentTopology topology = new TridentTopology();
89.
90.        // Training stream:
91.        // 1. Read a from string from kafka
92.        // 2. Convert trident tuple to instance
93.        // 3. Update the state of clusterer
94.        TridentState          kmeansState          =          topology.newStream("samples",
   kafkaSpout).each(new          Fields("str"),          new          FeaturesToValues(),          new
   Fields("instance")).partitionPersist(new          MemoryMapState.Factory(),          new
   Fields("instance"), new ClusterUpdater("kmeans", new KMeans(6)));
95.
96.        // Now we will build LocalDRPC that will be used to predict the cluster of a tuple
97.        LocalDRPC localDRPC = new LocalDRPC();
98.
99.        // Clustering stream
100.       // 1. Define a new clustering stream with name = 'predict'
101.       // 2. Convert DRPC args to instance
102.       // 3. Query cluster to classify the instance
103.
104.       // We are using KMeans(6) as we want to cluster into 6 categories
105.       topology.newDRPCStream("predict",   localDRPC).each(new   Fields("args"),   new
   FeaturesToValues(),          new          Fields("instance")).stateQuery(kmeansState,          new
   Fields("instance"), new ClusterQuery("kmeans"), new Fields("prediction"));
106.
107.       // Create a new local cluster for testing
108.       LocalCluster cluster = new LocalCluster();
109.
110.       // submit the topology for execution
111.       cluster.submitTopology("kmeans", new Config(), topology.build());
112.
113.       // give the topology enough time to create the clustering model
```

```
114.        Thread.sleep(10000);
115.
116.        // Create the prediction consumer, please change the path for input and output
117.        // file as needed
118.        PredictionConsumer predictionConsumer = new PredictionConsumer(localDRPC,
    "/home/anand/Desktop/prediction.data", "/home/anand/Desktop/predicted.data");
119.
120.        // Predict and write the output
121.        predictionConsumer.predict();
122.
123.        // shutdown cluster and drpc
124.        cluster.shutdown();
125.        localDRPC.shutdown();
126.    }
    }
```

127. Now that the topology is ready, let's create a consumer that will predict the category for the test data generated in the last section. For this, create the com.learningstorm.ml.PredictionConsumer class with the following code:

```
128. public class PredictionConsumer {
129.    // drpc instance used for prediction
130.    private final LocalDRPC drpc;
131.
132.    // input file, generated by kafka producer for prediction
133.    private final String input;
134.
135.    // output file, where the predicted data will be stored
136.    private final String output;
137.
138.    public PredictionConsumer(LocalDRPC drpc, String input, String output) {
139.        this.drpc = drpc;
140.        this.input = input;
141.        this.output = output;
142.    }
143.
144.    /**
145.     * This method predicts the categories for the records in the input file and writes them
    to the output file.
146.     */
```

```
147.    public void predict() throws IOException{
148.        // Scanner on the input file
149.        Scanner scanner = new Scanner(new File(input));
150.
151.        // Writer for the output
152.        BufferedWriter writer = new BufferedWriter(new FileWriter(new File(output)));
153.
154.        while(scanner.hasNextLine()){
155.            String line = scanner .nextLine();
156.            if(line.trim().length()==1){
157.                // empty line, skip
158.                continue;
159.            }
160.
161.            // predict the category for this line
162.            String prediction = drpc.execute("predict", line);
163.
164.            // write the predicted category for this line
165.            writer.write(prediction+"\n");
166.        }
167.
168.        // close the scanner and writer
169.        scanner.close();
170.        writer.close();
171.    }
    }
```

172. Now we have all the components in place and we can run the topology. Now, when running, it will first create the clustering model and then classify the test data generated earlier using that mode. To run it using Maven, execute the following command:

173. **mvn exec:java**

Note

If we are not running in the local mode DRPC, we will need to launch the DRPC server before running the topology. The following are the steps to run the DRPC server in the clustered mode:

1. Start the DRPC server with the following command:
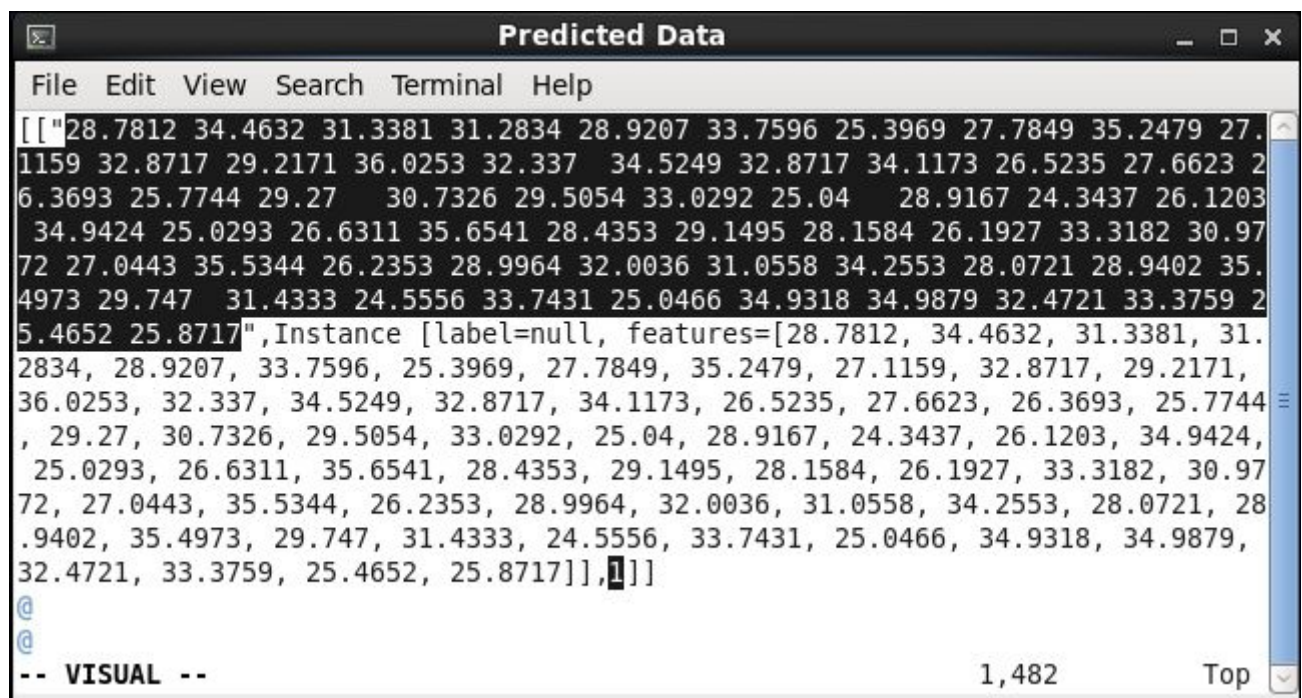
bin/storm drpc

2. Add DRPC servers in the storm.yaml file with the following entry:

drpc.servers:
  - "server1"
  - "server2"

174. After running the preceding command, you should be able to see the output with the classified example. Let's look at the first line in that file, which is shown in the following screenshot:

The predicted data

The first highlighted string is the input tuple for which the prediction is to be made. After that, we can see that this input instance was converted into an Instance object with label = null and features extracted from the input string in the form of a double array. The final highlighted number—1, in this case—represents the predicted category for this input.

Here, we have run the topology and classification in the local mode using LocalCluster and LocalDRPC, but this can run equally well on a Storm cluster. The only change that we will need to make is to write predictions to some central storage, such as NFS, instead of the local filesystem.

Summary

In this chapter, we introduced the topic of machine learning. You also learned how to run K-means clustering algorithms over Storm using Trident-ML and then use the generated model to predict the category of data using DRPC.

Although we used Trident-ML in this chapter, there are other machine learning packages also available for Storm. Storm.pattern (GitHub repository: https://github.com/quintona/storm-pattern) is one such library that can import models from other non-Storm packages, such as R, Weka, and so on.

With this, we come to the end of this book. Through the course of this book, we have come a long way from taking our first steps with Apache Storm to developing real-world applications with it. Here, we would like to summarize everything that we learned.

We introduced you to the basic concepts and components of Storm and covered how we can write and deploy/run the topology in the local and clustered modes. We also walk through the basic commands of Storm and cover how we can modify the parallelism of the Storm topology in runtime. We also dedicated an entire chapter to monitoring Storm, which is an area often neglected during development, but is a critical part of any production setting. You also learned about Trident, which is an abstraction over the low-level Storm API to develop more complex topologies and maintain the application state.

No enterprise application can be developed in a single technology, and so our next step was to see how we could integrate Storm with other Big Data tools and technologies. We saw specific implementation of Storm with Kafka, Hadoop, HBase, and Redis. Most of the Big Data applications use Ganglia as a centralized monitoring tool. Hence, we also covered how we could monitor the Storm cluster through JMX and Ganglia.

You also learned about various patterns to integrate diverse data sources with Storm. Finally, in Chapter 8, *Log Processing with Storm*, and this chapter, we implemented two case studies in Apache Storm, which can serve as a starting point for developing more complex applications.

We hope that reading this book has been a fruitful journey for you, and that you developed a basic understanding of Storm and, in general, various aspects of developing a real-time stream processing application. Apache Storm is turning into a de facto standard for stream processing, and we hope that this book will act as a catalyst for you to jumpstart the exciting journey of building a real-time stream processing applications.

Index

A

F

G

H

I

J