Community Experience Distilled

Apache Kafka

Set up Apache Kafka clusters and develop custom message producers and consumers using practical, hands-on examples.

Nishant Garg

[PACKT] PUBLISHING open source* community experience distilled

Apache Kafka

Set up Apache Kafka clusters and develop custom message producers and consumers using practical, hands-on examples

Nishant Garg

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Apache Kafka

First published: October 2013

Production Reference: 1101013

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK..

ISBN 978-1-78216-793-8

www.packtpub.com

Cover Image by Suresh Mogre (suresh.mogre.99@gmail.com)

Credits

Author
Nishant Garg

Reviewers
Magnus Edenhill
Iuliia Proskurnia

Acquisition Editors
Usha Iyer
Julian Ursell

Commissioning Editor
Shaon Basu

Technical Editor
Veena Pagare

Copy Editors
Tanvi Gaitonde
Sayanee Mukherjee
Aditya Nair
Kirti Pai
Alfida Paiva
Adithi Shetty

Project Coordinator
Esha Thakker

Proofreader
Christopher Smith

Indexers
Monica Ajmera
Hemangini Bari
Tejal Daruwale

Graphics
Abhinash Sahu

Production Coordinator
Kirtee Shingan

Cover Work
Kirtee Shingan

About the Author

Nishant Garg is a Technical Architect with more than 13 years' experience in various technologies such as Java Enterprise Edition, Spring, Hibernate, Hadoop, Hive, Flume, Sqoop, Oozie, Spark, Kafka, Storm, Mahout, and Solr/Lucene; NoSQL databases such as MongoDB, CouchDB, HBase and Cassandra, and MPP Databases such as GreenPlum and Vertica.

He has attained his M.S. in Software Systems from Birla Institute of Technology and Science, Pilani, India, and is currently a part of Big Data R&D team in innovation labs at Impetus Infotech Pvt. Ltd.

Nishant has enjoyed working with recognizable names in IT services and financial industries, employing full software lifecycle methodologies such as Agile and SCRUM. He has also undertaken many speaking engagements on Big Data technologies.

I would like to thank my parents (Sh. Vishnu Murti Garg and Smt. Vimla Garg) for their continuous encouragement and motivation throughout my life. I would also like to thank my wife (Himani) and my kids (Nitigya and Darsh) for their never-ending support, which keeps me going.

Finally, I would like to thank Vineet Tyagi—AVP and Head of Innovation Labs, Impetus—and Dr. Vijay—Director of Technology, Innovation Labs, Impetus—for having faith in me and giving me an opportunity to write.

About the Reviewers

Magnus Edenhill is a freelance systems developer living in Stockholm, Sweden, with his family. He specializes in high-performance distributed systems but is also a veteran in embedded systems.

For ten years, Magnus played an instrumental role in the design and implementation of PacketFront's broadband architecture, serving millions of FTTH end customers worldwide. Since 2010, he has been running his own consultancy business with customers ranging from Headweb—northern Europe's largest movie streaming service—to Wikipedia.

Iuliia Proskurnia is a doctoral student at EDIC school of EPFL, specializing in Distributed Computing. Iuliia was awarded the EPFL fellowship to conduct her doctoral research. She is a winner of the Google Anita Borg scholarship and was the Google Ambassador at KTH (2012-2013). She obtained a Masters Diploma in Distributed Computing (2013) from KTH, Stockholm, Sweden, and UPC, Barcelona, Spain. For her Master's thesis, she designed and implemented a unique real-time, low-latency, reliable, and strongly consistent distributed data store for the stock exchange environment at NASDAQ OMX. Previously, she has obtained Master's and Bachelor's Diplomas with honors in Computer Science from the National Technical University of Ukraine KPI. This Master's thesis was about fuzzy portfolio management in previously uncertain conditions. This period was productive for her in terms of publications and conference presentations. During her studies in Ukraine, she obtained several scholarships. During her stay in Kiev, Ukraine, she worked as Financial Analyst at Alfa Bank Ukraine.

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.

Table of Contents

Preface

This book is here to help you get familiar with Apache Kafka and use it to solve your challenges related to the consumption of millions of messages in publisher-subscriber architecture. It is aimed at getting you started with a feel for programming with Kafka so that you will have a solid foundation to dive deep into its different types of implementations and integrations.

In addition to an explanation of Apache Kafka, we also offer a chapter exploring Kafka integration with other technologies such as Apache Hadoop and Storm. Our goal is to give you an understanding of not just what Apache Kafka is, but also how to use it as part of your broader technical infrastructure.

What this book covers

Chapter 1, *Introducing Kafka*, discusses how organizations are realizing the real value of data and evolving the mechanism of collecting and processing it.

Chapter 2, *Installing Kafka*, describes how to install and build Kafka 0.7.x and 0.8.

Chapter 3, *Setting up the Kafka Cluster*, describes the steps required to set up a single/multibroker Kafka cluster.

Chapter 4, *Kafka Design*, discusses the design concepts used for building a solid foundation for Kafka.

Chapter 5, *Writing Producers*, provides detailed information about how to write basic producers and some advanced-level Java producers that use message partitioning.

Chapter 6, *Writing Consumers*, provides detailed information about how to write basic consumers and some advanced-level Java consumers that consume messages from the partitions.

Chapter 7, *Kafka Integrations*, discusses how Kafka integration works for both Storm and Hadoop to address real-time and batch processing needs.

Chapter 8, *Kafka Tools*, describes information about Kafka tools, such as its administrator tools, and Kafka integration with Camus, Apache Camel, Amazon cloud, and so on.

What you need for this book

In the simplest case, a single Linux-based (CentOS 6.x) machine with JDK 1.6 installed will give you a platform to explore almost all the exercises in this book. We assume you have some familiarity with command-line Linux; any modern distribution will suffice.

Some of the examples in this book need multiple machines to see things working, so you will require access to at least three such hosts. Virtual machines are fine for learning and exploration.

You will generally need the big data technologies, such as Hadoop and Storm, to run your Hadoop and Storm clusters.

Who this book is for

This book is for readers who want to know about Apache Kafka at a hands-on level; the key audience is those with software development experience but no prior exposure to Apache Kafka or similar technologies.

This book is also for enterprise application developers and big data enthusiasts who have worked with other publisher-subscriber-based systems and now want to explore Apache Kafka as a futuristic scalable solution.

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 are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
String messageStr = new String("Hello from Java Producer");
KeyedMessage<Integer, String> data = new KeyedMessage<Integer,
String>(topic, messageStr);
producer.send(data);
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Properties props = new Properties();
props.put("metadata.broker.list","localhost:9092");
props.put("serializer.class","kafka.serializer.StringEncoder");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<Integer, String> producer = new Producer<Integer,
    String>(config);
```

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

```
[root@localhost kafka-0.8]# java SimpleProducer kafkatopic Hello_There
```

Warnings or important notes appear in a box like this.

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 color images of this book

We also provide you with a PDF file that has color images of the screenshots used in this book. You can download this file from `http://www.packtpub.com/sites/default/files/downloads/7938OS_Images.pdf`.

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.

1
Introducing Kafka

Welcome to the world of Apache Kafka.

In today's world, real-time information is continuously getting generated by applications (business, social, or any other type), and this information needs easy ways to be reliably and quickly routed to multiple types of receivers. Most of the time, applications that are producing information and applications that are consuming this information are well apart and inaccessible to each other. This, at times, leads to redevelopment of information producers or consumers to provide an integration point between them. Therefore, a mechanism is required for seamless integration of information of producers and consumers to avoid any kind of rewriting of an application at either end.

In the present big data era, the very first challenge is to collect the data as it is a huge amount of data and the second challenge is to analyze it. This analysis typically includes following type of data and much more:

- User behavior data
- Application performance tracing
- Activity data in the form of logs
- Event messages

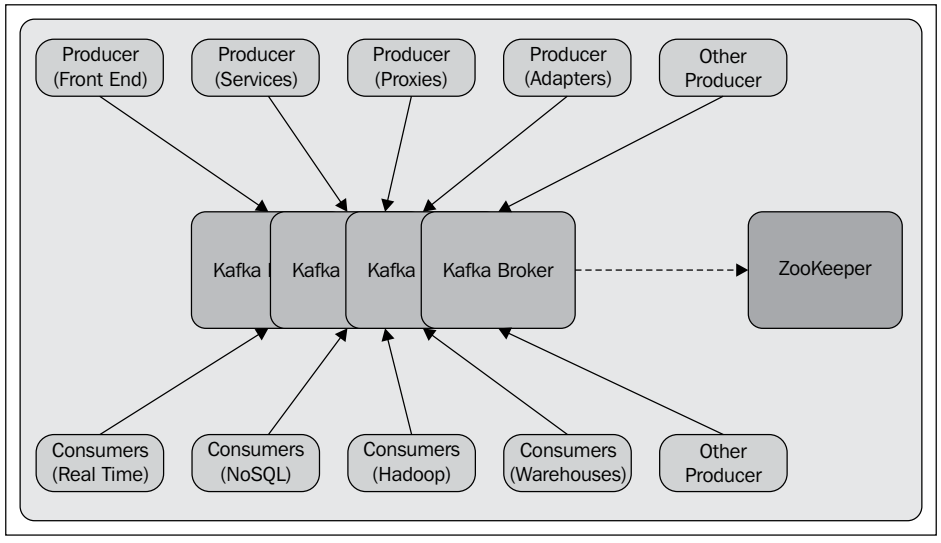
Message publishing is a mechanism for connecting various applications with the help of messages that are routed between them, for example, by a message broker such as Kafka. Kafka is a solution to the real-time problems of any software solution, that is, to deal with real-time volumes of information and route it to multiple consumers quickly. Kafka provides seamless integration between information of producers and consumers without blocking the producers of the information, and without letting producers know who the final consumers are.

Apache Kafka is an open source, distributed publish-subscribe messaging system, mainly designed with the following characteristics:

- **Persistent messaging**: To derive the real value from big data, any kind of information loss cannot be afforded. Apache Kafka is designed with **O(1)** disk structures that provide constant-time performance even with very large volumes of stored messages, which is in order of TB.
- **High throughput**: Keeping big data in mind, Kafka is designed to work on commodity hardware and to support millions of messages per second.
- **Distributed**: Apache Kafka explicitly supports messages partitioning over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.
- **Multiple client support**: Apache Kafka system supports easy integration of clients from different platforms such as Java, .NET, PHP, Ruby, and Python.
- **Real time**: Messages produced by the producer threads should be immediately visible to consumer threads; this feature is critical to event-based systems such as **Complex Event Processing** (**CEP**) systems.

Kafka provides a real-time publish-subscribe solution, which overcomes the challenges of real-time data usage for consumption, for data volumes that may grow in order of magnitude, larger that the real data. Kafka also supports parallel data loading in the Hadoop systems.

The following diagram shows a typical big data aggregation-and-analysis scenario supported by the Apache Kafka messaging system:



At the production side, there are different kinds of producers, such as the following:

- Frontend web applications generating application logs
- Producer proxies generating web analytics logs
- Producer adapters generating transformation logs
- Producer services generating invocation trace logs

At the consumption side, there are different kinds of consumers, such as the following:

- Offline consumers that are consuming messages and storing them in Hadoop or traditional data warehouse for offline analysis
- Near real-time consumers that are consuming messages and storing them in any NoSQL datastore such as HBase or Cassandra for near real-time analytics
- Real-time consumers that filter messages in the in-memory database and trigger alert events for related groups

Need for Kafka

A large amount of data is generated by companies having any form of web-based presence and activity. Data is one of the newer ingredients in these Internet-based systems and typically includes user-activity events corresponding to logins, page visits, clicks, social networking activities such as likes, sharing, and comments, and operational and system metrics. This data is typically handled by logging and traditional log aggregation solutions due to high throughput (millions of messages per second). These traditional solutions are the viable solutions for providing logging data to an offline analysis system such as Hadoop. However, the solutions are very limiting for building real-time processing systems.

According to the new trends in Internet applications, activity data has become a part of production data and is used to run analytics at real time. These analytics can be:

- Search based on relevance
- Recommendations based on popularity, co-occurrence, or sentimental analysis
- Delivering advertisements to the masses
- Internet application security from spam or unauthorized data scraping

Real-time usage of these multiple sets of data collected from production systems has become a challenge because of the volume of data collected and processed.

Apache Kafka aims to unify offline and online processing by providing a mechanism for parallel load in Hadoop systems as well as the ability to partition real-time consumption over a cluster of machines. Kafka can be compared with Scribe or Flume as it is useful for processing activity stream data; but from the architecture perspective, it is closer to traditional messaging systems such as ActiveMQ or RabitMQ.

Few Kafka usages

Some of the companies that are using Apache Kafka in their respective use cases are as follows:

- **LinkedIn** (`www.linkedin.com`): Apache Kafka is used at LinkedIn for the streaming of activity data and operational metrics. This data powers various products such as LinkedIn news feed and LinkedIn Today in addition to offline analytics systems such as Hadoop.
- **DataSift** (`www.datasift.com/`): At DataSift, Kafka is used as a collector for monitoring events and as a tracker of users' consumption of data streams in real time.
- **Twitter** (`www.twitter.com/`): Twitter uses Kafka as a part of its Storm— a stream-processing infrastructure.
- **Foursquare** (`www.foursquare.com/`): Kafka powers online-to-online and online-to-offline messaging at Foursquare. It is used to integrate Foursquare monitoring and production systems with Foursquare, Hadoop-based offline infrastructures.
- **Square** (`www.squareup.com/`): Square uses Kafka as a *bus* to move all system events through Square's various datacenters. This includes metrics, logs, custom events, and so on. On the consumer side, it outputs into Splunk, Graphite, or Esper-like real-time alerting.

The source of the above information is `https://cwiki.apache.org/confluence/display/KAFKA/Powered+By`.

Summary

In this chapter, we have seen how companies are evolving the mechanism of collecting and processing application-generated data, and that of utilizing the real power of this data by running analytics over it.

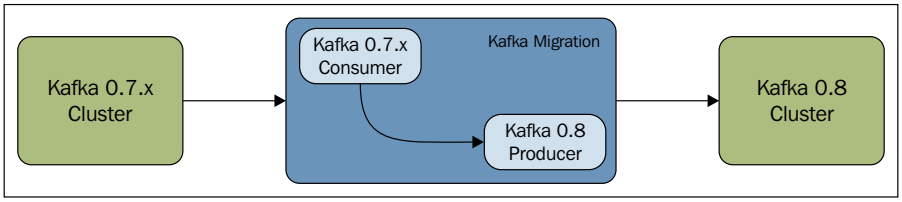
In the next chapter we will look at the steps required to install Kafka.

2
Installing Kafka

Kafka is an Apache project and its current Version 0.7.2 is available as a stable release. Kafka Version 0.8 is available as beta release, which is gaining acceptance in many large-scale enterprises. Kafka 0.8 offers many advanced features compared to 0.7.2. A few of its advancements are as follows:

- Prior to 0.8, any unconsumed partition of data within the topic could be lost if the broker failed. Now the partitions are provided with a replication factor. This ensures that any committed message would not be lost, as at least one replica is available.
- The previous feature also ensures that all the producers and consumers are replication aware. By default, the producer's message send request is blocked until the message is committed to all active replicas; however, producers can also be configured to commit messages to a single broker.
- Like Kafka producers, Kafka consumers' polling model changes to a long pulling model and gets blocked until a committed message is available from the producer, which avoids frequent pulling.
- Additionally, Kafka 0.8 also comes with a set of administrative tools, such as controlled shutdown of cluster and Lead replica election tool, for managing the Kafka cluster.

The major limitation is that Kafka Version 0.7.x can't just be replaced by Version 0.8, as it is not backward compatible. If the existing Kafka cluster is based on 0.7.x, a migration tool is provided for migrating the data from the Kafka 0.7.x-based cluster to the 0.8-based cluster. This migration tool actually works as a consumer for 0.7.x-based Kafka clusters and republishes the messages as a producer to Kafka 0.8-based clusters. The following diagram explains this migration:



More information about Kafka migration from 0.7.x to 0.8 can be found at `https://cwiki.apache.org/confluence/display/KAFKA/Migrating+from+0.7+to+0.8`.

Coming back to installing Kafka, as a first step, we need to download the available stable/beta release (all the commands are tested on CentOS 5.5 OS and may differ on other kernel-based OS).

Installing Kafka

Now let us see what steps need to be followed in order to install Kafka:

Downloading Kafka

Perform the following steps for downloading Kafka release 0.7.x:

1. Download the current stable version of Kafka (0.7.2) into a folder on your file system (for example, `/opt`) using the following command:

   ```
   [root@localhost opt]#wget https://www.apache.org/dyn/closer.cgi/incubator/kafka/kafka-0.7.2-incubating/kafka-0.7.2-incubating-src.tgz
   ```

2. Extract the downloaded `kafka-0.7.2-incubating-src.tgz` using the following command:

   ```
   [root@localhost opt]# tar xzf kafka-0.7.2-incubating-src.tgz
   ```

Perform the following steps for downloading Kafka release 0.8:

1. Download the current beta release of Kafka (0.8) into a folder on your filesystem (for example, `/opt`) using the following command:

   ```
   [root@localhost opt]#wget
   https://dist.apache.org/repos/dist/release/kafka/kafka-0.8.0-beta1-src.tgz
   ```

2. Extract the downloaded `kafka-0.8.0-beta1-src.tgz` using the following command:

   ```
   [root@localhost opt]# tar xzf kafka-0.8.0-beta1-src.tgz
   ```

Going forward, all commands in this chapter are same for both the versions (0.7.x and 0.8) of Kafka.

Installing the prerequisites

Kafka is implemented in Scala and uses the `./sbt` tool for building Kafka binaries. **sbt** is a build tool for Scala and Java projects which requires Java 1.6 or later.

Installing Java 1.6 or later

Perform the following steps for installing Java 1.6 or later:

1. Download the `jdk-6u45-linux-x64.bin` link from Oracle's website: `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
2. Make sure the file is executable:

   ```
   [root@localhost opt]#chmod +x jdk-6u45-linux-x64.bin
   ```

3. Run the installer:

   ```
   [root@localhost opt]#./jdk-6u45-linux-x64.bin
   ```

4. Finally, add the environment variable `JAVA_HOME`. The following command will write the `JAVA_HOME` environment variable to the file `/etc/profile`, which contains system-wide environment configuration:

   ```
   [root@localhost opt]# echo "export JAVA_HOME=/usr/java/
   jdk1.6.0_45" >> /etc/profile
   ```

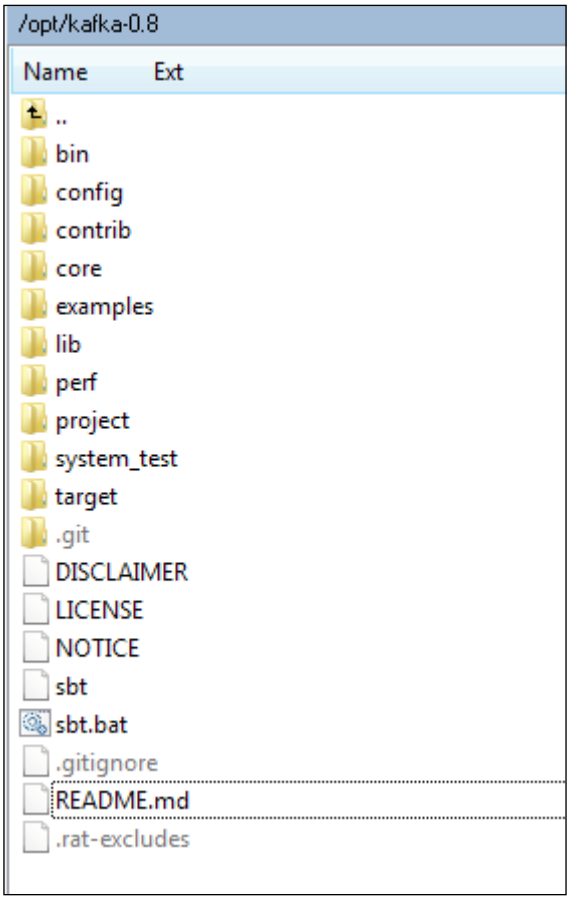
Building Kafka

The following steps need to be followed for building and packaging Kafka:

1. Change the current directory to the downloaded Kafka directory by using the following command:

   ```
   [root@localhost opt]# cd kafka-<VERSION>
   ```

2. The directory structure for Kafka 0.8 looks as follows:



3. The following command downloads all the dependencies such as Scala compiler, Scala libraries, Zookeeper, Core-Kafka update, and Hadoop consumer/producer update, for building Kafka:

   ```
   [root@localhost opt]#./sbt update
   ```

On execution of the previous command, you should see the following output on the command prompt:

```
[info] Resolving org.apache.hadoop#hadoop-core;0.20.2 ...
[info] Done updating.
[info] Updating {file:/opt/kafka-0.8/}java-examples...
[info] Resolving com.yammer.metrics#metrics-annotation;2.2.0 ...
[info] Done updating.
[info] Updating {file:/opt/kafka-0.8/}perf...
[info] Resolving com.yammer.metrics#metrics-annotation;2.2.0 ...
[info] Done updating.
[info] Updating {file:/opt/kafka-0.8/}hadoop-consumer...
[info] Resolving org.apache.hadoop#hadoop-core;0.20.2 ...
[info] Done updating.
[success] Total time: 47 s, completed Aug 28, 2013 12:32:40 AM
[root@localhost kafka-0.8]#
```

4. Finally, compile the complete source code for Core-Kafka, Java examples, and Hadoop producer/consumer, and package them into JAR files using the following command:

   ```
   [root@localhost opt]#./sbt package
   ```

   On execution of the previous command, you should see the following output on the command prompt:

```
[info] Compiling 5 Java sources to /opt/kafka-0.8/examples/target/classes...
[info] Compiling 4 Java sources to /opt/kafka-0.8/contrib/hadoop-producer/target/classes...
[info] Packaging /opt/kafka-0.8/core/target/scala-2.8.0/kafka_2.8.0-0.8.0-beta1.jar ...
[info] Done packaging.
[info] Packaging /opt/kafka-0.8/target/scala-2.8.0/kafka_2.8.0-0.8.0-beta1.jar ...
[info] Done packaging.
[info] Packaging /opt/kafka-0.8/contrib/target/scala-2.8.0/contrib_2.8.0-0.8.0-beta1.jar ...
[info] Done packaging.
[info] Packaging /opt/kafka-0.8/perf/target/scala-2.8.0/kafka-perf_2.8.0-0.8.0-beta1.jar ...
[info] Done packaging.
[info] Packaging /opt/kafka-0.8/contrib/hadoop-consumer/target/hadoop-consumer-0.8.0-beta1.jar ...
[info] Done packaging.
[info] Packaging /opt/kafka-0.8/examples/target/kafka-java-examples-0.8.0-beta1.jar ...
[info] Done packaging.
[info] Packaging /opt/kafka-0.8/contrib/hadoop-producer/target/hadoop-producer-0.8.0-beta1.jar ...
[info] Done packaging.
[success] Total time: 200 s, completed Aug 28, 2013 12:38:13 AM
[root@localhost kafka-0.8]#
```

5. The following additional command is only needed with Kafka 0.8 for producing the dependency artifacts:

   ```
   [root@localhost opt]#./sbt assembly-package-dependency
   ```

On execution of the previous command, you should see the following output on the command prompt:

```
[info] Including jopt-simple-3.2.jar
[info] Including zookeeper-3.3.4.jar
[info] Including slf4j-api-1.7.2.jar
[info] Including slf4j-simple-1.6.4.jar
[info] Including snappy-java-1.0.4.1.jar
[info] Including scala-compiler.jar
[info] Including scala-library.jar
[warn] Merging 'META-INF/NOTICE' with strategy 'rename'
[warn] Merging 'org/xerial/snappy/native/README' with strategy 'rename'
[warn] Merging 'META-INF/maven/org.xerial.snappy/snappy-java/LICENSE' with strategy 'rename'
[warn] Merging 'LICENSE.txt' with strategy 'rename'
[warn] Merging 'META-INF/LICENSE' with strategy 'rename'
[warn] Merging 'META-INF/MANIFEST.MF' with strategy 'discard'
[warn] Strategy 'discard' was applied to a file
[warn] Strategy 'rename' was applied to 5 files
[info] Packaging /opt/kafka-0.8/core/target/scala-2.8.0/kafka-assembly-0.8.0-beta1-deps.jar ...
[info] Done packaging.
[success] Total time: 20 s, completed Aug 28, 2013 12:40:10 AM
[root@localhost kafka-0.8]#
```

If you are planning to play with Kafka 0.8, you may experience lot of warnings with update and package commands, which can be ignored.

Summary

In this chapter we have learned how to install and build Kafka 0.7.x and 0.8. The following chapter discusses the steps required to set up single/multibroker Kafka clusters. From here onwards, the book only focuses on Kafka 0.8.

3
Setting up the Kafka Cluster

Now we are ready to play with the Apache Kafka publisher-based messaging system.

With Kafka, we can create multiple types of clusters, such as the following:

- Single node – single broker cluster
- Single node – multiple broker cluster
- Multiple node – multiple broker cluster

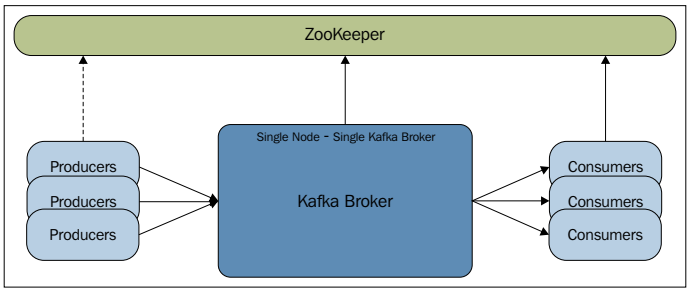
All the commands and cluster setups in this chapter are based on Kafka 0.8.

With Kafka 0.8, replication of clusters can also be established, which will be discussed in brief in the last part of this chapter.

So let's start with the basics.

Single node – single broker cluster

This is the starting point of learning Kafka. In the previous chapter, we built and installed Kafka on a single machine. Now it is time to setup single node – single broker based Kafka cluster as shown in the following diagram



Starting the ZooKeeper server

Kafka provides the default and simple ZooKeeper configuration file used for launching a single local ZooKeeper instance. Here, ZooKeeper serves as the coordination interface between the Kafka broker and consumers. The Hadoop overview given on the Hadoop Wiki site is as follows (`http://wiki.apache.org/hadoop/ZooKeeper/ProjectDescription`):

> *"ZooKeeper (`http://zookeeper.apache.org`) allows distributed processes coordinating with each other through a shared hierarchical name space of data registers (**znodes**), much like a file system.*
>
> *The main differences between ZooKeeper and standard filesystems are that every znode can have data associated with it and znodes are limited to the amount of data that they can have. ZooKeeper was designed to store coordination data: status information, configuration, location information, and so on."*

First start the ZooKeeper using the following command:

```
[root@localhost kafka-0.8]# bin/zookeeper-server-start.sh config/
zookeeper.properties
```

You should get an output as shown in the following screenshot:

```
[2013-06-05 14:14:31,740] INFO Server environment:java.io.tmpdir=/tmp (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,740] INFO Server environment:java.compiler=<NA> (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,740] INFO Server environment:os.name=Linux (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,740] INFO Server environment:os.arch=amd64 (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,740] INFO Server environment:os.version=2.6.18-348.6.1.el5 (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,741] INFO Server environment:user.name=root (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,741] INFO Server environment:user.home=/root (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,742] INFO Server environment:user.dir=/opt/kafka-0.8 (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,767] INFO tickTime set to 3000 (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,767] INFO minSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,767] INFO maxSessionTimeout set to -1 (org.apache.zookeeper.server.ZooKeeperServer)
[2013-06-05 14:14:31,804] INFO binding to port 0.0.0.0/0.0.0.0:2181 (org.apache.zookeeper.server.NIOServerCnxn)
[2013-06-05 14:14:31,852] INFO Snapshotting: 0 (org.apache.zookeeper.server.persistence.FileTxnSnapLog)
```

Kafka comes with the required property files defining minimal properties required for a single broker – single node cluster.

The important properties defined in `zookeeper.properties` are shown in the following code:

```
# Data directory where the zookeeper snapshot is stored.
dataDir=/tmp/zookeeper

# The port listening for client request
clientPort=2181
```

By default, the ZooKeeper server will listen on `*:2181/tcp`. For detailed information on how to set up multiple servers of ZooKeeper, visit `http://zookeeper.apache.org/`.

Starting the Kafka broker

Now start the Kafka broker using the following command:

```
[root@localhost kafka-0.8]# bin/kafka-server-start.sh config/server.
properties
```

You should now see the output as shown in the following screenshot:

```
[2013-08-28 09:20:22,427] INFO Starting ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2013-08-28 09:20:22,456] INFO Opening socket connection to server localhost/127.0.0.1:2181 (org.apache.zooke
eper.ClientCnxn)
[2013-08-28 09:20:22,474] INFO Socket connection established to localhost/127.0.0.1:2181, initiating session
(org.apache.zookeeper.ClientCnxn)
[2013-08-28 09:20:22,562] INFO Session establishment complete on server localhost/127.0.0.1:2181, sessionid =
 0x140c5b9930c0000, negotiated timeout = 6000 (org.apache.zookeeper.ClientCnxn)
[2013-08-28 09:20:22,564] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)
[2013-08-28 09:20:22,704] INFO Registered broker 1 at path /brokers/ids/1 with address localhost.localdomain:
9092. (kafka.utils.ZkUtils$)
[2013-08-28 09:20:22,705] INFO [Kafka Server 1], Connecting to ZK: localhost:2181 (kafka.server.KafkaServer)
[2013-08-28 09:20:22,876] INFO Will not load MX4J, mx4j-tools.jar is not in the classpath (kafka.utils.Mx4jLo
ader$)
[2013-08-28 09:20:22,905] INFO 1 successfully elected as leader (kafka.server.ZookeeperLeaderElector)
[2013-08-28 09:20:23,187] INFO New leader is 1 (kafka.server.ZookeeperLeaderElector$LeaderChangeListener)
[2013-08-28 09:20:23,287] INFO [Kafka Server 1], Started (kafka.server.KafkaServer)
[2013-08-28 09:21:00,662] INFO No state transitions triggered since no partitions are assigned to brokers 2 (
kafka.utils.ZkUtils$)
[2013-08-28 09:21:31,774] INFO No state transitions triggered since no partitions are assigned to brokers 3 (
kafka.utils.ZkUtils$)
```

`server.properties` defines the following important properties required for the Kafka broker:

```
# The id of the broker. This must be set to a unique integer for each
broker.
Broker.id=0

# The directory under which to store log files
log.dir=/tmp/kafka8-logs

# Zookeeper connection string
zookeeper.connect=localhost:2181
```

The last section in this chapter defines few more important properties available for the Kafka broker. For a complete list of Kafka broker properties, visit `http://kafka.apache.org/documentation.html#brokerconfigs`.

Creating a Kafka topic

Kafka provides a command-line utility for creating topics on the Kafka server. Let's create a topic named `kafkatopic` with a single partition and only one replica using this utility:

```
[root@localhost kafka-0.8]# bin/kafka-create-topic.sh --zookeeper
localhost:2181 --replica 1 --partition 1 --topic kafkatopic
```

You should get an output as shown in the following screenshot:

```
[2013-08-28 10:17:16,307] INFO Client environment:java.compiler=<NA> (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,307] INFO Client environment:os.name=Linux (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,307] INFO Client environment:os.arch=amd64 (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,307] INFO Client environment:os.version=2.6.18-348.6.1.el5 (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,307] INFO Client environment:user.name=root (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,308] INFO Client environment:user.home=/root (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,308] INFO Client environment:user.dir=/opt/kafka-0.8_OLD (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,309] INFO Initiating client connection, connectString=localhost:2181 sessionTimeout=30000 w
atcher=org.I0Itec.zkclient.ZkClient@57a7ddcf (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:16,324] INFO Starting ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2013-08-28 10:17:16,360] INFO Opening socket connection to server localhost/127.0.0.1:2181 (org.apache.zookeepe
r.ClientCnxn)
[2013-08-28 10:17:16,408] INFO Socket connection established to localhost/127.0.0.1:2181, initiating session (or
g.apache.zookeeper.ClientCnxn)
[2013-08-28 10:17:16,450] INFO Session establishment complete on server localhost/127.0.0.1:2181, sessionid = 0x
140c5b9930c0003, negotiated timeout = 30000 (org.apache.zookeeper.ClientCnxn)
[2013-08-28 10:17:16,453] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)
creation succeeded!
[2013-08-28 10:17:18,857] INFO Terminate ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2013-08-28 10:17:19,122] INFO Session: 0x140c5b9930c0003 closed (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:17:19,123] INFO EventThread shut down (org.apache.zookeeper.ClientCnxn)
```

The previously mentioned utility will create a topic and show the successful creation message as shown in the previous screenshot.

Starting a producer for sending messages

Kafka provides users with a command-line producer client that accepts inputs from the command line and publishes them as a message to the Kafka cluster. By default, each new line entered is considered as a new message. The following command is used to start the console-based producer for sending the messages

```
[root@localhost kafka-0.8]# bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic kafkatopic
```

You should see an output as shown in the following screenshot:

```
[root@localhost kafka-0.8]# bin/kafka-console-producer.sh --broker-list localhost:9092 --topic kafkatopic
[2013-09-02 01:03:12,620] INFO Verifying properties (kafka.utils.VerifiableProperties)
[2013-09-02 01:03:12,655] INFO Property queue.buffering.max.messages is overridden to 10000 (kafka.utils.Verifia
bleProperties)
[2013-09-02 01:03:12,656] INFO Property key.serializer.class is overridden to kafka.serializer.StringEncoder (ka
fka.utils.VerifiableProperties)
[2013-09-02 01:03:12,656] INFO Property compression.codec is overridden to 0 (kafka.utils.VerifiableProperties)
[2013-09-02 01:03:12,657] INFO Property serializer.class is overridden to kafka.serializer.StringEncoder (kafka.
utils.VerifiableProperties)
[2013-09-02 01:03:12,657] INFO Property request.timeout.ms is overridden to 1500 (kafka.utils.VerifiableProperti
es)
[2013-09-02 01:03:12,657] INFO Property send.buffer.bytes is overridden to 102400 (kafka.utils.VerifiablePropert
ies)
[2013-09-02 01:03:12,657] INFO Property request.required.acks is overridden to 0 (kafka.utils.VerifiableProperti
es)
[2013-09-02 01:03:12,658] INFO Property producer.type is overridden to async (kafka.utils.VerifiableProperties)
[2013-09-02 01:03:12,658] INFO Property metadata.broker.list is overridden to localhost:9092 (kafka.utils.Verifi
ableProperties)
[2013-09-02 01:03:12,658] INFO Property queue.buffering.max.ms is overridden to 1000 (kafka.utils.VerifiableProp
erties)
[2013-09-02 01:03:12,665] INFO Property queue.enqueue.timeout.ms is overridden to 0 (kafka.utils.VerifiablePrope
rties)
```

While starting the producer's command-line client, the following parameters are required:

- `broker-list`
- `topic`

`broker-list` specifies the brokers to be connected as `<node_address:port>`, that is, `localhost:9092`. The topic `Kafkatopic` is a topic that was created in the *Creating a Kafka topic* section. The topic name is required for sending a message to a specific group of consumers.

Now type the following message, `This is single broker`, and press *Enter*. You should see an output as shown in the following screenshot:

```
[2013-09-02 01:03:12,658] INFO Property producer.type is overridden to async (kafka.utils.VerifiableProperties)
[2013-09-02 01:03:12,658] INFO Property metadata.broker.list is overridden to localhost:9092 (kafka.utils.Verifi
ableProperties)
[2013-09-02 01:03:12,658] INFO Property queue.buffering.max.ms is overridden to 1000 (kafka.utils.VerifiableProp
erties)
[2013-09-02 01:03:12,665] INFO Property queue.enqueue.timeout.ms is overridden to 0 (kafka.utils.VerifiablePrope
rties)
This is single broker
[2013-09-02 01:55:48,365] INFO Fetching metadata from broker id:0,host:localhost,port:9092 with correlation id 0
 for 1 topic(s) Set(kafkatopic) (kafka.client.ClientUtils$)
[2013-09-02 01:55:48,453] INFO Connected to localhost:9092 for producing (kafka.producer.SyncProducer)
[2013-09-02 01:55:48,736] INFO Disconnecting from localhost:9092 (kafka.producer.SyncProducer)
[2013-09-02 01:55:49,011] INFO Connected to localhost.localdomain:9093 for producing (kafka.producer.SyncProduce
r)
Welcome to Kafka
This is another Test Message
```

Try some more messages.

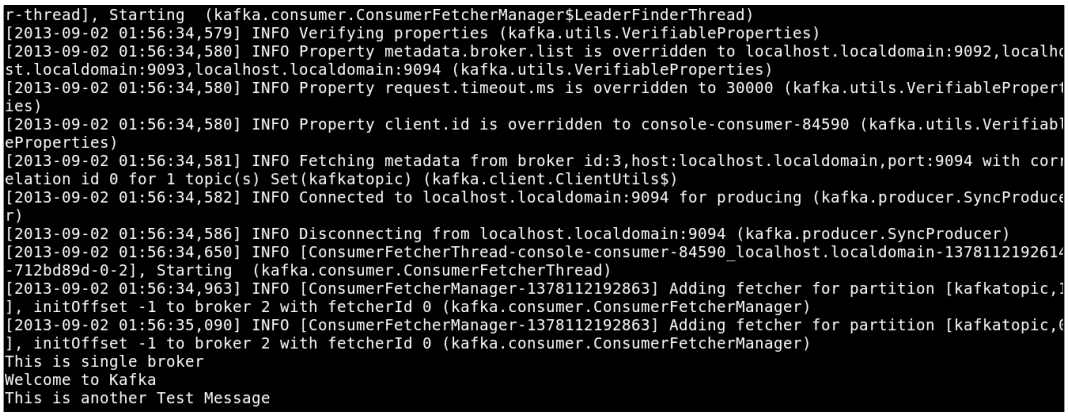
Detailed information about how to write producers for Kafka and producer properties will be discussed in *Chapter 5, Writing Producers*.

Starting a consumer for consuming messages

Kafka also provides a command-line consumer client for message consumption. The following command is used for starting the console-based consumer that shows output at command line as soon as it subscribes to the topic created in Kafka broker:

```
[root@localhost kafka-0.8]# bin/kafka-console-consumer.sh
--zookeeper localhost:2181 --topic kafkatopic  --from-beginning
```

On execution of the previous command, you should get an output as shown in the following screenshot:



The default properties for the consumer are defined in `consumer.properties`. The important properties are:

```
# consumer group id (A string that uniquely identifies a set of
consumers # within the same consumer group)
groupid=test-consumer-group

# zookeeper connection string
zookeeper.connect=localhost:2181
```

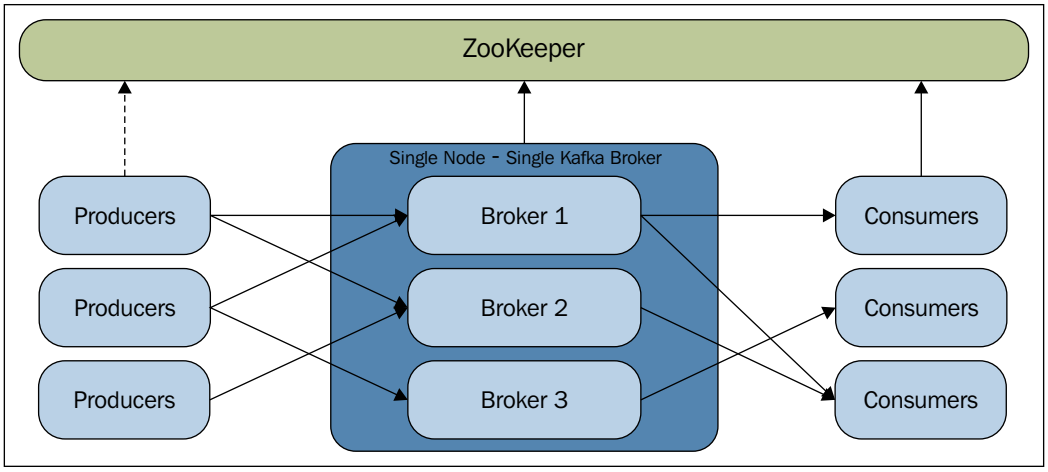
Detailed information about how to write consumers for Kafka and consumer properties is discussed in *Chapter 6*, *Writing Consumers*.

By running all four components (`zookeeper`, `broker`, `producer`, and `consumer`) in different terminals, you will be able to enter messages from the producer's terminal and see them appearing in the subscribed consumer's terminal.

Usage information for both producer and consumer command-line tools can be viewed by running the command with no arguments.

Single node – multiple broker cluster

Now we have come to the next level of Kafka cluster. Let us now set up single node – multiple broker based Kafka cluster as shown in the following diagram:



Starting ZooKeeper

The first step of starting ZooKeeper remains the same for this type of cluster.

Starting the Kafka broker

For setting up multiple brokers on a single node, different server property files are required for each broker. Each property file will define unique, different values for the following properties:

- `brokerid`
- `port`
- `log.dir`

For example, in `server-1.properties` used for `broker1`, we define the following:

- `brokerid=1`
- `port=9092`
- `log.dir=/tmp/kafka8-logs/broker1`

Similarly, for `server-2.properties` used for `broker2`, we define the following:

- `brokerid=2`
- `port=9093`
- `log.dir=/tmp/kafka8-logs/broker2`

A similar procedure is followed for all brokers. Now, we start each broker in a separate console using the following commands:

```
[root@localhost kafka-0.8]# env JMX_PORT=9999 bin/kafka-server-start.sh
config/server-1.properties
[root@localhost kafka-0.8]# env JMX_PORT=10000 bin/kafka-server-start.sh
config/server-2.properties
```

Similar commands are used for all brokers You will also notice that we have defined a separate JMX port for each broker.

The JMX ports are used for optional monitoring and troubleshooting with tools such as JConsole.

Creating a Kafka topic

Using the command-line utility for creating topics on the Kafka server, let's create a topic named `othertopic` with two partitions and two replicas:

```
[root@localhost kafka-0.8]# bin/kafka-create-topic.sh --zookeeper
localhost:2181 --replica 2 --partition 2 --topic othertopic
```

Starting a producer for sending messages

If we use a single producer to get connected to all the brokers, we need to pass the initial list of brokers, and the information of the remaining brokers is identified by querying the broker passed within `broker-list`, as shown in the following command. This metadata information is based on the topic name.

```
--broker-list localhost:9092,localhost:9093
```

Use the following command to start the producer:

```
[root@localhost kafka-0.8]# bin/kafka-console-producer.sh --broker-list
localhost:9092,localhost:9093 --topic othertopic
```

If we have a requirement to run multiple producers connecting to different combinations of brokers, we need to specify the broker list for each producer like we did in the case of multiple brokers.

Starting a consumer for consuming messages

The same consumer client, as in the previous example, will be used in this process. Just as before, it shows the output on the command line as soon as it subscribes to the topic created in the Kafka broker:

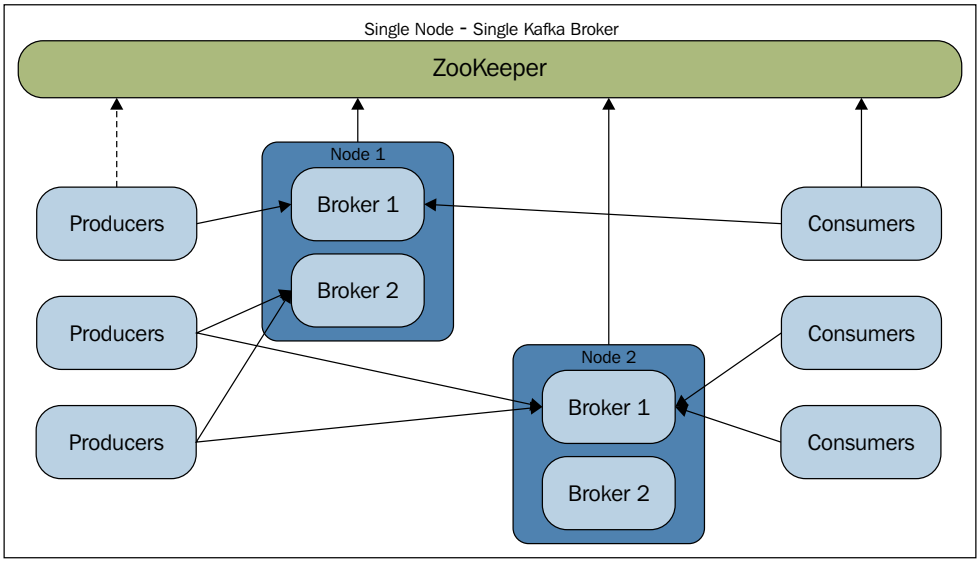
```
[root@localhost kafka-0.8]# bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic othertopic  --from-beginning
```

Multiple node – multiple broker cluster

This cluster scenario is not discussed in detail in this book, but as in the case of multiple-node Kafka cluster, where we set up multiple brokers on each node, we should install Kafka on each node of the cluster, and all the brokers from the different nodes need to connect to the same ZooKeeper.

For testing purposes, all the commands will remain identical to the ones we used in the single node – multiple brokers cluster.

The following diagram shows the cluster scenario where multiple brokers are configured on multiple nodes (**Node 1** and **Node 2** in this case), and the producers and consumers are getting connected in different combinations:



Kafka broker property list

The following is the list of few important properties that can be configured for the Kafka broker. For the complete list, visit `http://kafka.apache.org/documentation.html#brokerconfig`.

Property name	Description	Default value
`broker.id`	Each broker is uniquely identified by an ID. This ID serves as the broker's name, and allows the broker to be moved to a different host/port without confusing consumers.	`0`
`log.dirs`	These are the directories in which the log data is kept.	`/tmp/kafka-logs`
`zookeeper.connect`	This specifies the ZooKeeper's connection string in the form `hostname:port/chroot`. Here, `chroot` is a base directory that is prepended to all path operations (this effectively namespaces all Kafka znodes to allow sharing with other applications on the same ZooKeeper cluster).	`localhost:2181`

Summary

In this chapter, we have learned how to set up a Kafka cluster with single/multiple brokers on a single node, run command-line producers and consumers, and exchange some messages. We have also discussed some details about setting up a multinode – multibroker cluster.

In the next chapter, we will look at the internal design of Kafka.

4
Kafka Design

Before we start getting our hands dirty by coding Kafka producers and consumers, let's quickly discuss the internal design of Kafka.

In this chapter we shall be focusing on the following topics:

- Kafka design fundamentals
- Message compression in Kafka
- Cluster mirroring in Kafka
- Replication in Kafka

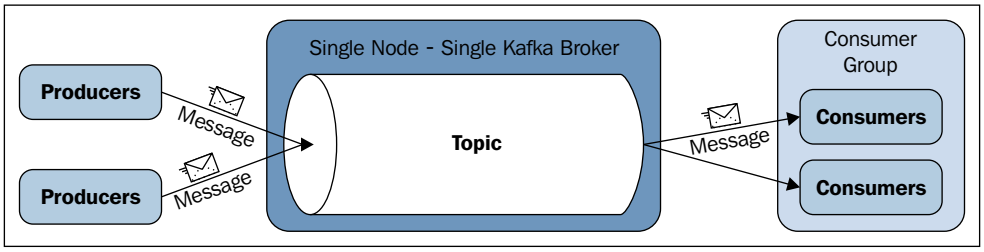
Due to the overheads associated with JMS and its various implementations and limitations with the scaling architecture, LinkedIn (`www.linkedin.com`) decided to build Kafka to address their need for monitoring activity stream data and operational metrics such as CPU, I/O usage, and request timings.

While developing Kafka, the main focus was to provide the following:

- An API for producers and consumers to support custom implementation
- Low overhead for network and storage with message persistence
- High throughput supporting millions of messages
- Distributed and highly scalable architecture

Kafka design fundamentals

In a very basic structure, a producer publishes messages to a Kafka topic, which is created on a Kafka broker acting as a Kafka server. Consumers then subscribe to the Kafka topic to get the messages. This is described in the following diagram:



In the preceding diagram a single node – single broker architecture is shown. This architecture considers that all three parties—producers, Kafka broker, and consumers—are running on different machines.

Here, each consumer is represented as a process and these processes are organized within groups called **consumer groups**.

A message is consumed by a single process (consumer) within the consumer group, and if the requirement is such that a single message is to be consumed by multiple consumers, all these consumers need to be kept in different consumer groups.

By Kafka design, the message state of any consumed message is maintained within the message consumer, and the Kafka broker does not maintain a record of what is consumed by whom, which also means that poor designing of a custom consumer ends up in reading the same message multiple times.

Important Kafka design facts are as follows:

- The fundamental backbone of Kafka is message caching and storing it on the filesystem. In Kafka, data is immediately written to the OS kernel page. Caching and flushing of data to the disk is configurable.
- Kafka provides longer retention of messages ever after consumption, allowing consumers to reconsume, if required.
- Kafka uses a message set to group messages to allow lesser network overhead.

- Unlike most of the messaging systems, where metadata of the consumed messages are kept at server level, in Kafka, the state of the consumed messages is maintained at consumer level. This also addresses issues such as:
    - Loosing messages due to failure
    - Multiple deliveries of the same message

  By default, consumers store the state in ZooKeeper, but Kafka also allows storing it within other storage systems used for **Online Transaction Processing** (**OLTP**) applications as well.
- In Kafka, producers and consumers work on the traditional push-and-pull model, where producers push the message to a Kafka broker and consumers pull the message from the broker.
- Kafka does not have any concept of a master and treats all the brokers as peers. This approach facilitates addition and removal of a Kafka broker at any point, as the metadata of brokers are maintained in ZooKeeper and shared with producers and consumers.
- In Kafka 0.7.x, ZooKeeper-based load balancing allows producers to discover the broker dynamically. A producer maintains a pool of broker connections, and constantly updates it using ZooKeeper watcher callbacks. But in Kafka 0.8.x, load balancing is achieved through Kafka metadata API and ZooKeeper can only be used to identify the list of available brokers.
- Producers also have an option to choose between asynchronous or synchronous mode for sending messages to a broker.

Message compression in Kafka

As we have discussed, Kafka uses message set feature for grouping the messages. It also provides a message group compression feature. Here, data is compressed by the message producer using either **GZIP** or **Snappy** compression protocols and decompressed by the message consumer. There is lesser network overhead for the compressed message set where it also puts very little overhead of decompression at the consumer end.

This compressed set of messages can be presented as a single message to the consumer who later decompresses it. Hence, the compressed message may have infinite depth of messages within itself.

To differentiate between compressed and uncompressed messages, a compression-attributes byte is introduced in the message header. Within this compression byte, the lowest two bits are used to represent the compression codec used for compression and the value `0` of these last two bits represents an uncompressed message.

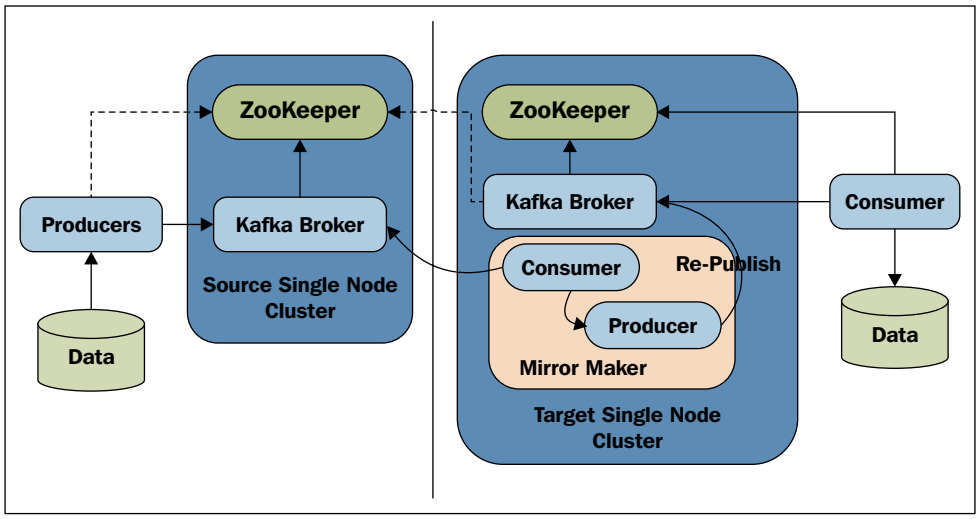
Message compression techniques are very useful for mirroring the data across datacenters using Kafka, where large amounts of data get transferred from active to passive datacenters in compressed format.

For more detailed usage on Kafka message compression, visit `https://cwiki.apache.org/confluence/display/KAFKA/Compression`.

Cluster mirroring in Kafka

The Kafka mirroring feature is used for creating the replica of an existing cluster, for example, for the replication of an active datacenter into a passive datacenter. Kafka provides a mirror maker tool for mirroring the source cluster into target cluster.

The following diagram depicts the mirroring tool placement in architectural form:



In this architecture, the job of the mirror tool is to consume the messages from the source cluster and republish them on the target cluster.

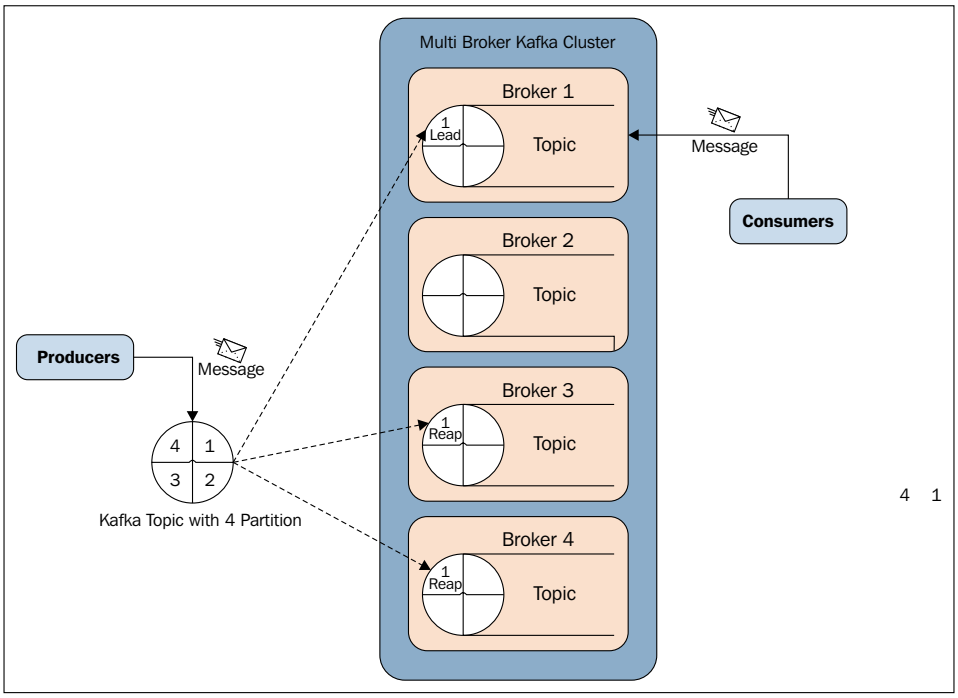
A similar approach is used by the Kafka migration tool to migrate from 0.7.x Kafka cluster to 0.8.x Kafka cluster.

For detailed explanation on the mirror maker tool setup and configuration, visit `https://cwiki.apache.org/confluence/display/KAFKA/Kafka+mirroring+%28MirrorMaker%29`.

Replication in Kafka

Before we talk about replication in Kafka, let's talk about message partitioning. In Kafka, message partitioning strategy is used at the Kafka broker end. The decision about how the message is partitioned is taken by the producer, and the broker stores the messages in the same order as they arrive. The number of partitions can be configured for each topic within the Kafka broker.

Kafka replication is one of the very important features introduced in Kafka 0.8. Though Kafka is highly scalable, for better durability of messages and high availability of Kafka clusters, replication guarantees that the message will be published and consumed even in case of broker failure, which may be caused by any reason. Here, both producers and consumers are replication aware in Kafka. The following diagram explains replication in Kafka:



Let's discuss the preceding diagram in detail.

In replication, each partition of a message has *n* replicas and can afford *n-1* failures to guarantee message delivery. Out of the *n* replicas, one replica acts as the lead replica for the rest of the replicas. ZooKeeper keeps the information about the lead replica and the current in-sync follower replica (lead replica maintains the list of all in-sync follower replicas).

Each replica stores its part of the message in local logs and offsets, and is periodically synced to the disk. This process also ensures that either a message is written to all the replicas or to none of them.

If the lead replica fails, either while writing the message partition to its local log or before sending the acknowledgement to the message producer, a message partition is resent by the producer to the new lead broker.

The process of choosing the new lead replica is that all followers' **In-sync Replicas** (**ISRs**) register themselves with ZooKeeper. The very first registered replica becomes the new lead replica, and the rest of the registered replicas become the followers.

Kafka supports the following replication modes:

- **Synchronous replication**: In synchronous replication, a producer first identifies the lead replica from ZooKeeper and publishes the message. As soon as the message is published, it is written to the log of the lead replica and all the followers of the lead start pulling the message, and by using a single channel, the order of messages is ensured. Each follower replica sends an acknowledgement to the lead replica once the message is written to its respective logs. Once replications are complete and all expected acknowledgements are received, the lead replica sends an acknowledgement to the producer.

  On the consumer side, all the pulling of messages is done from the lead replica.

- **Asynchronous replication**: The only difference in this mode is that as soon as a lead replica writes the message to its local log, it sends the acknowledgement to the message client and does not wait for the acknowledgements from follower replicas. But as a down side, this mode does not ensure the message delivery in case of broker failure.

For detailed explanation on Kafka replication and its usage, visit `https://cwiki.apache.org/confluence/display/KAFKA/Kafka+Replication`.

Summary

In this chapter, we have learned the design concepts used for building a solid foundation for Kafka.

In the next chapter, we shall be focusing on how to write Kafka producers using the API provided.

5
Writing Producers

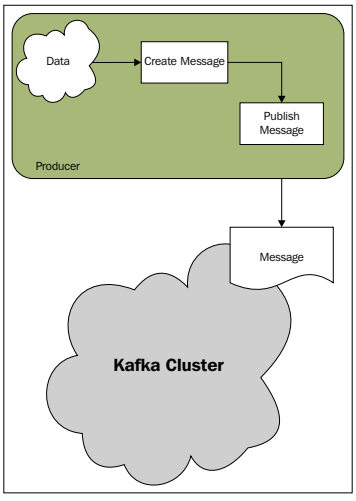
Producers are applications that create messages and publish them to the Kafka broker for further consumption. These producers can be different in nature; for example, frontend applications, backend services, proxy applications, adapters to legacy systems, and producers for Hadoop. These producers can also be implemented in different languages such as Java, C, and Python.

In this chapter we shall be focusing on the following topics:

- The Kafka API for message producers
- Simple Java-based Kafka producers
- Java-based Kafka producers using message partitioning

At the end of the chapter, we will explore some of the important properties required for the Kafka producer.

Let's begin. The following diagram explains the Kafka API for message producers:

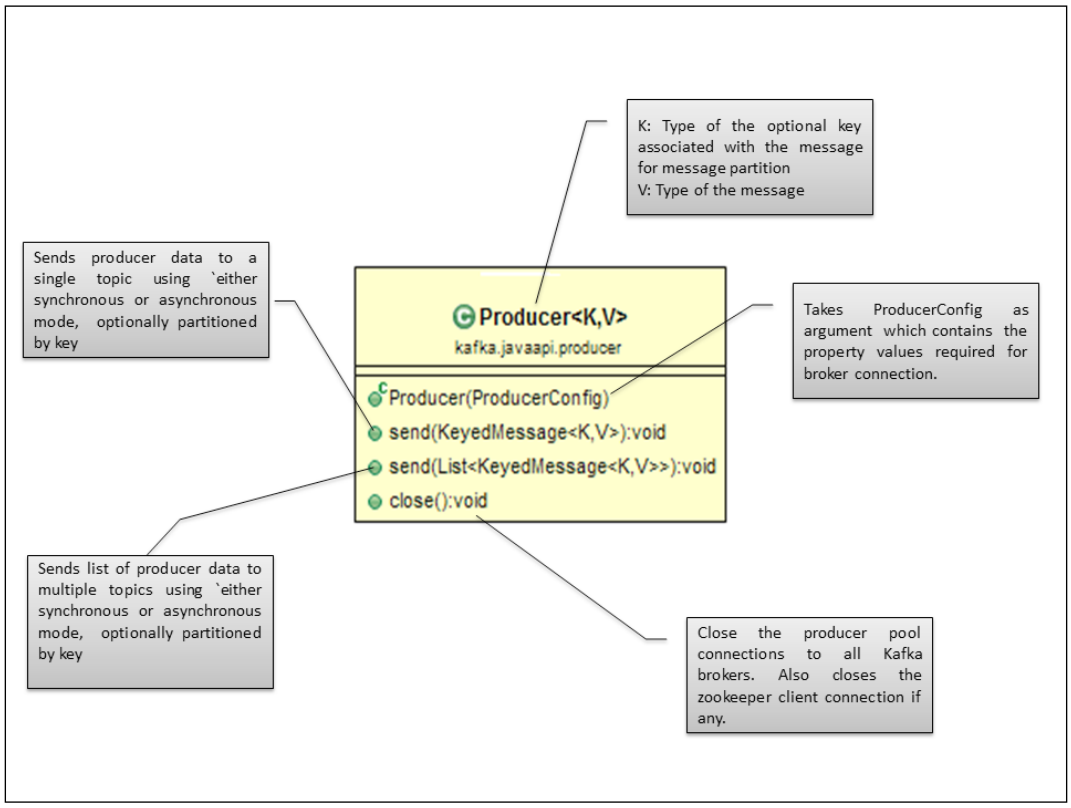


In the next few sections, we will discuss the API provided by Kafka for writing Java-based custom producers.

The Java producer API

The following are the classes that are imported to write the Java-based basic producers for a Kafka cluster:

- **Producer**: Kafka provides the `Producer` class (`class Producer<K,V>`) for creating messages for single or multiple topics with message partition as an optional feature. The following is the class diagram and its explanation:



Here, `Producer` is a type of Java generic (`http://en.wikipedia.org/wiki/Generics_in_Java`) written in Scala where we need to specify the type of parameters; `K` and `V` specify the types for the partition key and message value, respectively.

- **KeyedMessage**: The `KeyedMessage` class takes the topic name, partition key, and the message value that needs to be passed from the producer as follows:

  ```
  class KeyedMessage[K, V](val topic: String, val key: K, val
  message: V)
  ```

  Here, `KeyedMessage` is a type of Java generic written in Scala where we need to specify the type of the parameters; `K` and `V` specify the type for the partition key and message value, respectively, and the topic is always of type `String`.

- **ProducerConfig**: The `ProducerConfig` class encapsulates the values required for establishing the connection with brokers such as the broker list, message partition class, serializer class for the message, and partition key.

Simple Java producer

Now we will start writing a simple Java-based producer to transmit the message to the broker. This `SimpleProducer` class is used to create a message for a specific topic and transmit it.

Importing classes

As the first step, we need to import the following classes:

```
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;
```

Defining properties

As the next step in writing the producer, we need to define properties for making a connection with Kafka broker and pass these properties to the Kafka producer:

```
Properties props = new Properties();
props.put("metadata.broker.list","localhost:9092");
props.put("serializer.class","kafka.serializer.StringEncoder");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<Integer, String> producer = new Producer<Integer,
  String>(config);
```

Now let us see the major properties mentioned in the code:

- `metadata.broker.list`: This property specifies the broker `<node:port>` that the producer needs to connect to (more information is provided in the next example).
- `serializer.class list`: This property specifies the serializer class that needs to be used while preparing the message for transmission from the producer to the broker. In this example, we will be using the string encoder provided by Kafka. By default, the serializer class for the key and message is the same, but we can change the serializer class for the key by using the `key.serializer.class` property.
- `request.required.acks`: This property instructs the Kafka broker to send an acknowledgment to the producer when a message is received. By default, the producer works in the "fire and forget" mode and is not informed in case of message loss.

Building the message and sending it

As the final step, we need to build the message and send it to the broker as shown in the following code:

```
String messageStr = new String("Hello from Java Producer");
KeyedMessage<Integer, String> data = new KeyedMessage<Integer,
String>(topic, messageStr);
producer.send(data);
```

The complete program will look as follows:

```
package test.kafka;

import java.util.Properties;
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class SimpleProducer {
    private static Producer<Integer, String> producer;
    private final Properties props = new Properties();
    public    SimpleProducer()
    {
      props.put("broker.list", "localhost:9092");
      props.put("serializer.class", "kafka.serializer.StringEncoder");
      props.put("request.required.acks", "1");
      producer = new Producer<Integer, String>(new
      ProducerConfig(props));
    }
```

```
public static void main(String[] args) {
    SimpleProducer sp = new SimpleProducer();
    String topic = (String) args[0];
    String messageStr = (String) args[1];
    KeyedMessage<Integer, String> data = new KeyedMessage<Integer,
    String>(topic, messageStr);
    producer.send(data);
    producer.close();
  }
}
```

Compile the preceding program and use the following command to run it:

```
[root@localhost kafka-0.8]# java SimpleProducer kafkatopic Hello_There
```

Here, `kafkatopic` is the topic that will be created automatically when the message `Hello_There` is sent to the broker.

Creating a simple Java producer with message partitioning

The previous example is a very basic example of a `Producer` class and only uses a single broker with no explicit partitioning of messages. Let's jump to the next level and write another program that connects to multiple brokers and uses message partitioning.

Importing classes

This step remains the same for this program.

Defining properties

As the next step, we need to define properties for making a connection with the Kafka broker, as shown in the following code, and pass these properties to the Kafka producer:

```
Properties props = new Properties();
props.put("metadata.broker.list","localhost:9092, localhost:9093");
props.put("serializer.class","kafka.serializer.StringEncoder");
props.put("partitioner.class", "test.kafka.SimplePartitioner");
props.put("request.required.acks", "1");
ProducerConfig config = new ProducerConfig(props);
Producer<Integer, String> producer = new Producer<Integer,
String>(config);
```

The only change in the previous property list is in `metadata.broker.list` and `partitioner.class`.

- `metadata.broker.list`: This property specifies the list of brokers (in the `[<node:port>, <node:port>]` format) that the producer needs to connect to. Kafka producers automatically find out the lead broker for the topic as well as partition it by raising a request for the metadata before it sends any message to the the broker.
- `partitioner.class`: This property defines the class to be used for determining the partitioning in the topic where the message needs to be sent. If the key is null, Kafka uses random partitioning for message assignment.

Implementing the Partitioner class

Next, we need to implement the `Partitioner` class as shown in the following code:

```
package test.kafka;
import kafka.producer.Partitioner;
public class SimplePartitioner implements Partitioner<Integer> {
    public int partition(Integer key, int numPartitions) {
        int partition = 0;
        int iKey = key;
        if (iKey > 0) {
           partition = iKey % numPartitions;
        }
       return partition;
  }
}
```

Building the message and sending it

As the final step, we need to build the message and send it to the broker. The following is the complete listing of the program:

```
package test.kafka;

import java.util.Properties;
import java.util.Random;
import kafka.javaapi.producer.Producer;
import kafka.producer.KeyedMessage;
import kafka.producer.ProducerConfig;

public class MultiBrokerProducer {
    private static Producer<Integer, String> producer;
```

```
    private final Properties props = new Properties();

      public MultiBrokerProducer()
    {
    props.put("metadata.broker.list","localhost:9092,
    localhost:9093");
      props.put("serializer.class","kafka.serializer.StringEncoder");

      props.put("partitioner.class", "test.kafka.SimplePartitioner");

      props.put("request.required.acks", "1");

      ProducerConfig config = new ProducerConfig(props);
      producer = new Producer<Integer, String>(config);
    }
  public static void main(String[] args) {
    MultiBrokerProducer sp = new MultiBrokerProducer();
        Random rnd = new Random();

        String topic = (String) args[0];
        for (long messCount = 0; messCount < 10; messCount++) {
               Integer key = rnd.nextInt(255);
               String msg = "This message is for key - " + key;
               KeyedMessage<Integer, String> data1 = new
               KeyedMessage<Integer, String>(topic, key, msg);
               producer.send(data1);
        }
    producer.close();
  }
}
```

Compile the previous program. Before running it, read the following information box.

Before we run this program, we need to make sure our cluster is running as a multibroker cluster (either single or multiple nodes). For more information on how to set up a single node – multibroker cluster, refer to *Chapter 3, Setting up the Kafka Cluster*.

Once your multibroker cluster is up, create a topic with five partitions and set the replication factor as `2` before running this program using the following command:

```
[root@localhost kafka-0.8]# bin/kafka-topics.sh --zookeeper
localhost:2181 --create --topic kafkatopic --partitions 5 --replication-
factor 2
```

Now run the preceding program using the following command:

```
[root@localhost kafka-0.8]# java MultiBrokerProducer kafkatopic
```

For verifying the data that is getting published to the Kafka broker, the Kafka console consumer can be used as follows:

```
[root@localhost kafka-0.8]# bin/kafka-console-consumer.sh --zookeeper
localhost:2181 --topic kafkatopic --from-beginning
```

The Kafka producer property list

The following table shows the list of a few important properties that can be configured for Kafka producer. For the complete list, visit `http://kafka.apache.org/08/configuration.html`.

Property name	Description	Default value
`metadata.broker.list`	The producer uses this property for getting metadata (topics, partitions, and replicas). The socket connections for sending the actual data will be established based on the broker information returned in the metadata. The format is `host1:port1,host2:port2`.	
`serializer.class`	This specifies the serializer class for messages. The default encoder accepts a byte and returns the same byte.	`DefaultEncoder`
`producer.type`	This property specifies how the messages will be sent: `async` for asynchronous sending and `sync` for synchronous sending.	`sync`

Property name	Description	Default value
`request.required.acks`	This value controls *when* the producer receives an acknowledgment from the broker. The value `0` means the producer will not receive any acknowledgment from the broker. The value `1` means the producer receives an acknowledgment once the lead broker has received the data. The value `-1` means the producer will receive the acknowledgment once all the in-sync replicas have received the data.	`0`
`key.serializer.class`	This specifies the serializer class for keys (defaults to the same as for the message serializer class).	`${serializer.class}`
`partitioner.class`	This is the partitioner class for partitioning messages among subtopics. The default partitioner is based on the hash value of the key.	`DefaultPartitioner`

Summary

In this chapter we have learned how to write basic producers and some advanced Java producers that use message partitioning.

In the next chapter, we will learn how to write Java-based consumers for message consumption.

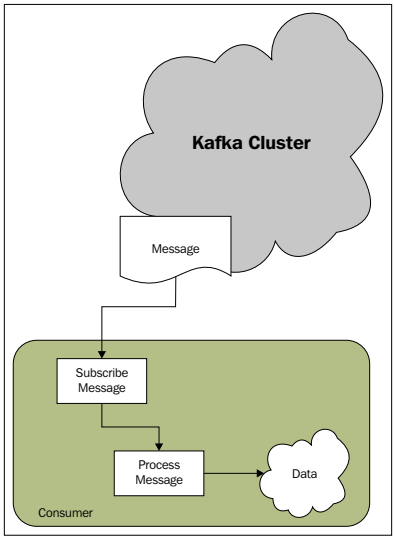
6
Writing Consumers

Consumers are the applications that consume the messages published by Kafka producers and process the data extracted from them. Like producers, consumers can also be different in nature, such as applications doing real-time or near-real-time analysis, applications with NoSQL or data warehousing solutions, backend services, consumers for Hadoop, or other subscriber-based solutions. These consumers can also be implemented in different languages such as Java, C, and Python.

In this chapter, we will focus on the following topics:

- Kafka API for message consumers
- Simple Java-based Kafka consumers
- Java-based Kafka consumers consuming partitioned messages

At the end of the chapter, we will explore some of the important properties required for a Kafka consumer. So, let's start.



In the next few sections, we will discuss the API provided by Kafka for writing Java-based custom consumers.

All the Kafka classes referred to in this book are actually written in Scala.

Java consumer API

Kafka provides two types of API for Java consumers:

- The high-level consumer API
- The simple consumer API

The high-level consumer API provides an abstraction over the low-level implementation of the consumer API, whereas the simple consumer API provides more control to the consumer by allowing it to override its default low-level implementation.

High-level consumer API

The high-level consumer API is used when only data is needed and the handling of message offsets is not required. Hence, most of the low-level details are abstracted during message consumption. The high-level consumer stores the last offset read from a specific partition in ZooKeeper. This offset is stored based on the consumer group name provided to Kafka at the beginning of the process.

Message offset is the position within the message partition to know where the consumer left off consuming the message.

The consumer group name is unique and global across the Kafka cluster and any new consumers with an in-use consumer group name may cause ambiguous behavior in the system. When a new process is started with the existing consumer group name, Kafka triggers rebalance between the new and existing process threads for the consumer group. Post rebalance, some of the messages that are intended for a new process may go to an old process, causing unexpected results. To avoid this ambiguous behavior, any existing consumers should be shut down before starting new consumers for an existing consumer group name.

The following are the classes that are imported to write Java-based basic consumers using the high-level consumer API for a Kafka cluster:

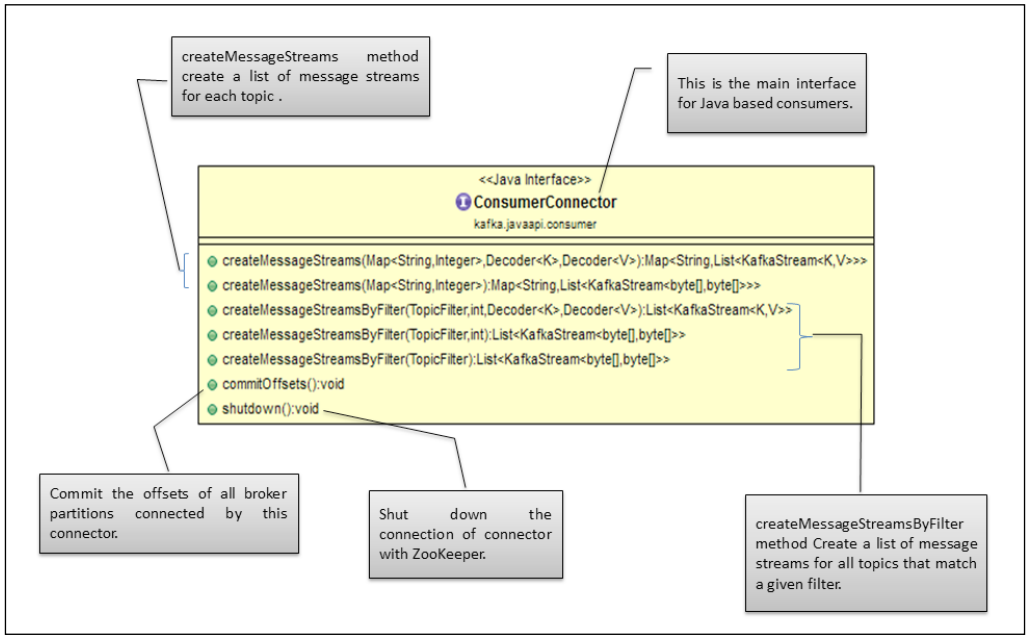
- **KafkaStream**: Objects of the `kafka.consumer.KafkaStream` class are returned by the `ConsumerConnector` implementation. This list of the `KafkaStream` objects is returned for each topic, which can further create an iterator shown as follows over messages in the stream:

  ```
  class KafkaStream[K,V]
  ```

  Here, the parameters `K` and `V` specify the type for the partition key and message value, respectively.

- **ConsumerConfig**: The `kafka.consumer.ConsumerConfig` class encapsulates the property values required for establishing the connection with ZooKeeper, such as ZooKeeper URL, group ID, ZooKeeper session timeout, and ZooKeeper sink time.
- **ConsumerConnector**: Kafka provides the `ConsumerConnector` interface (`interface ConsumerConnector`) which is further implemented by ZookeeperConsumerConnector class (`kafka.javaapi.consumer.ZookeeperConsumerConnector`). This class is responsible for all the interaction of a consumer with ZooKeeper.

  The following is the class diagram for the `ConsumerConnector` class:

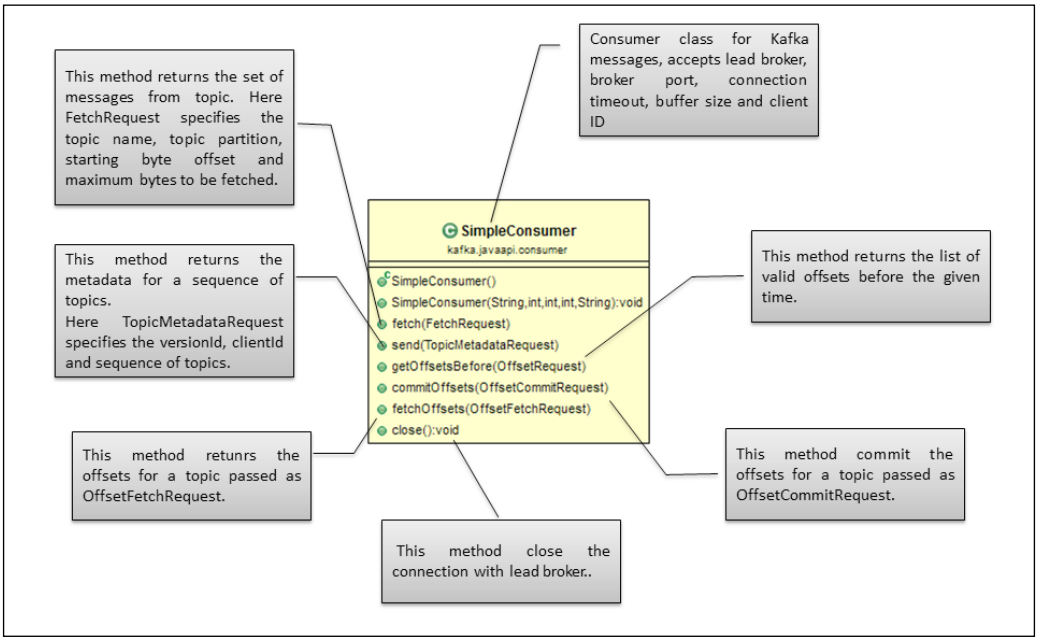


Simple consumer API

Features such as setting the initial offset when restarting the consumer are not provided by the high-level consumer API. The simple consumer API provides low-level control to Kafka consumers for partition consumptions, for example, multiple reads for the same message or managing transactions, and so on.

Compared to high-level consumer API, developer needs to put in extra efforts to gain this low-level control within consumers, that is, consumers need to keep track of offsets and also need to figure out the lead broker for the topic and partition, and so on.

The main class used within the simple consumer API is `SimpleConsumer` (`kafka.javaapi.consumer.SimpleConsumer`). The following is the class diagram for the `SimpleConsumer` class:



A simple consumer class provides a connection to the lead broker for fetching the messages from the topic and methods to get the topic metadata and the list of offsets.

A few more important classes for building different request objects are `FetchRequest` (`kafka.api.FetchRequest`), `OffsetRequest` (`kafka.javaapi.OffsetRequest`), `OffsetFetchRequest` (`kafka.javaapi.OffsetFetchRequest`), `OffsetCommitRequest` (`kafka.javaapi.OffsetCommitRequest`), and `TopicMetadataRequest` (`kafka.javaapi.TopicMetadataRequest`).

The following examples in this chapter are based on the high-level consumer API. For examples based on the simple consumer API, refer to `https://cwiki.apache.org/confluence/display/KAFKA/0.8.0+SimpleConsumer+Example`.

Simple high-level Java consumer

Now, we will start writing a single-threaded simple Java consumer developed using high-level consumer API for consuming the messages from a topic. This `SimpleHLConsumer` class is used to fetch a message from a specific topic and consume it, assuming that there is a single partition within the topic.

Importing classes

As a first step, we need to import the following classes:

```
import kafka.consumer.ConsumerConfig;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;
```

Defining properties

As a next step, we need to define properties for making a connection with ZooKeeper and pass these properties to the Kafka consumer using the following code:

```
Properties props = new Properties();
props.put("zookeeper.connect", "localhost:2181");
props.put("group.id", "testgroup");
props.put("zookeeper.session.timeout.ms", "500");
props.put("zookeeper.sync.time.ms", "250");
props.put("auto.commit.interval.ms", "1000");
new ConsumerConfig(props);
```

Now, let us see the major properties mentioned in the code:

- `zookeeper.connect`: This property specifies the ZooKeeper `<node:port>` connection details
- `group.id`: This property specifies the name for the consumer group shared by all the consumers within the group
- `zookeeper.session.timeout.ms`: This property specifies ZooKeeper session timeout in milliseconds

- `zookeeper.sync.time.ms`: This property specifies ZooKeeper sync time in milliseconds with ZooKeeper leader
- `auto.commit.interval.ms`: This property defines the frequency in milliseconds for the consumer offsets to get committed to ZooKeeper

Reading messages from a topic and printing them

As a final step, we need to read the message using the following code:

```
Map<String, Integer> topicCount = new HashMap<String, Integer>();
topicCount.put(topic, new Integer(1));

Map<String, List<KafkaStream<byte[], byte[]>>> consumerStreams =
consumer.createMessageStreams(topicCount);

List<KafkaStream<byte[], byte[]>> streams = consumerStreams.
get(topic);

for (final KafkaStream stream : streams) {
ConsumerIterator<byte[], byte[]> consumerIte = stream.iterator();
  while (consumerIte.hasNext())
    System.out.println("Message from Single Topic :: "
    + new String(consumerIte.next().message()));
}
```

So, the complete program will look like the following code:

```
package test.kafka.consumer;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;

import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

public class SimpleHLConsumer {

  private final ConsumerConnector consumer;
```

```
  private final String topic;

public SimpleHLConsumer(String zookeeper, String groupId, String
topic) {

    Properties props = new Properties();
    props.put("zookeeper.connect", zookeeper);
    props.put("group.id", groupId);
    props.put("zookeeper.session.timeout.ms", "500");
    props.put("zookeeper.sync.time.ms", "250");
    props.put("auto.commit.interval.ms", "1000");

    consumer = Consumer.createJavaConsumerConnector(
    new ConsumerConfig(props));
    this.topic = topic;
  }

  public void testConsumer() {

    Map<String, Integer> topicCount = new HashMap<String, Integer>();
        // Define single thread for topic
    topicCount.put(topic, new Integer(1));

Map<String, List<KafkaStream<byte[], byte[]>>> consumerStreams =
consumer.createMessageStreams(topicCount);

List<KafkaStream<byte[], byte[]>> streams = consumerStreams.
get(topic);

    for (final KafkaStream stream : streams) {
      ConsumerIterator<byte[], byte[]> consumerIte = stream.
      iterator();
      while (consumerIte.hasNext())
      System.out.println("Message from Single Topic :: " +
      new String(consumerIte.next().message()));
    }
    if (consumer != null)
      consumer.shutdown();
  }

  public static void main(String[] args) {

    String topic = args[0];
SimpleHLConsumer simpleHLConsumer = new SimpleHLConsumer("localho
st:2181", "testgroup", topic);
    simpleHLConsumer.testConsumer();
  }
}
```

Compile the previous program and use the following command to run it:

```
[root@localhost kafka-0.8]# java SimpleHLConsumer kafkatopic
```

Here, `kafkatopic` is the topic where the Kafka producer places the messages for consumption.

Multithreaded consumer for multipartition topics

The previous example is a very basic example of a consumer who consumes messages from a single broker with no explicit partitioning of messages within the topic. Let's jump to the next level and write another program, which consumes messages from multiple partitions connecting to single/multiple topics.

A multithreaded high-level consumer-API-based design is usually based on the number of partitions in the topic and follows a one-to-one mapping approach between the thread and the partitions within the topic. For example, if four partitions are defined for any topic, as a best practice, only four threads should be initiated with the consumer application to read the data; otherwise some conflicting behavior, such as threads never receiving a message or thread receiving messages from multiple partitions, may occur. Also, receiving multiple messages will not guarantee that the messages will be placed in order. For example, a thread may receive two messages from the first partition and three from the second partition, then three more from the first partition, followed by some more from the first partition, even if the second partition has data available.

Let's move further.

Importing classes

This step remains the same as the previous program.

Defining properties

This step remains the same for this program as well.

Reading the message from threads and printing it

The only difference in this section from the previous section is that we first create a thread pool and get the Kafka streams associated with each thread within the thread pool as shown in the following code:

```
Map<String, Integer> topicCount = new HashMap<String, Integer>();
    topicCount.put(topic, new Integer(threadCount));

Map<String, List<KafkaStream<byte[], byte[]>>> consumerStreams =
consumer.createMessageStreams(topicCount);

List<KafkaStream<byte[], byte[]>> streams = consumerStreams.
get(topic);

// Launching the thread pool
executor = Executors.newFixedThreadPool(threadCount);
```

The complete program listing for the multithread Kafka consumer based on the Kafka high-level consumer API is as follows:

```
package test.kafka.consumer;

import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import kafka.consumer.Consumer;
import kafka.consumer.ConsumerConfig;
import kafka.consumer.ConsumerIterator;
import kafka.consumer.KafkaStream;
import kafka.javaapi.consumer.ConsumerConnector;

public class MultiThreadHLConsumer {

  private ExecutorService executor;
  private final ConsumerConnector consumer;
  private final String topic;

  public MultiThreadHLConsumer(String zookeeper, String groupId,
  String topic) {

    Properties props = new Properties();
```

```
    props.put("zookeeper.connect", zookeeper);
    props.put("group.id", groupId);
    props.put("zookeeper.session.timeout.ms", "500");
    props.put("zookeeper.sync.time.ms", "250");
    props.put("auto.commit.interval.ms", "1000");

    consumer = Consumer.createJavaConsumerConnector(new
    ConsumerConfig(props));
    this.topic = topic;
  }

  public void testConsumer(int threadCount) {

    Map<String, Integer> topicCount = new HashMap<String, Integer>();

    // Define thread count for each topic
    topicCount.put(topic, new Integer(threadCount));

    // Here we have used a single topic but we can also add
    // multiple topics to topicCount MAP
    Map<String, List<KafkaStream<byte[], byte[]>>> consumerStreams =
    consumer.createMessageStreams(topicCount);

    List<KafkaStream<byte[], byte[]>> streams = consumerStreams.
    get(topic);

    // Launching the thread pool
    executor = Executors.newFixedThreadPool(threadCount);

    //Creating an object messages consumption
    int threadNumber = 0;
    for (final KafkaStream stream : streams) {
      ConsumerIterator<byte[], byte[]> consumerIte = stream.
      iterator();
      threadNumber++;

      while (consumerIte.hasNext())
    System.out.println("Message from thread :: " + threadNumber + " --
    " + new String(consumerIte.next().message()));
    }
```

```
        if (consumer != null)
          consumer.shutdown();
        if (executor != null)
          executor.shutdown();
      }

      public static void main(String[] args) {

        String topic = args[0];
        int threadCount = Integer.parseInt(args[1]);
        MultiThreadHLConsumer simpleHLConsumer = new MultiThreadHLConsumer
        ("localhost:2181", "testgroup", topic);
        simpleHLConsumer.testConsumer(threadCount);
      }
    }
```

Compile the previous program and, before running it, read the following information box.

Before we run this program, we need to make sure our cluster is running as a multibroker cluster (comprising either single or multiple nodes). For more information on how to set up single node – multiple broker cluster, refer to *Chapter 3*, *Setting up the Kafka Cluster*.

Once your multibroker cluster is up, create a topic with four partitions and set the replication factor to `2` before running this program using the following command:

```
[root@localhost kafka-0.8]# bin/kafka-topics.sh --zookeeper
localhost:2181 --create --topic kafkatopic --partitions 4 --replication-
factor 2
```

Now run the previous program using the following command:

```
[root@localhost kafka-0.8]# java MultiThreadHLConsumer kafkatopic 4
```

This program will print all the partitions of messages associated with each thread.

Kafka consumer property list

The following is the list of a few important properties that can be configured for high-level consumer-API-based Kafka consumers. For the complete list, visit `http://kafka.apache.org/08/configuration.html`.

Property name	Description	Default value
`group.id`	This property defines a unique identity for the set of consumers within the same consumer group.	
`zookeeper.connect`	This property specifies the ZooKeeper connection string, `< hostname:port/chroot>`. Kafka uses ZooKeeper to store offsets of messages consumed for a specific topic and partition by the consumer group.	
`client.id`	The `Client.id` value is specified by the Kafka consumer client and is used to distinguish between different clients.	`${group.id}`
`zookeeper.session.timeout.ms`	This property defines the time (in milliseconds) for Kafka to wait for ZooKeeper to respond to any read/write request before closing a session.	`6000`
`zookeeper.connection.timeout.ms`	This value defines the maximum waiting time (in milliseconds) for the client to establish a connection with ZooKeeper.	`6000`
`zookeeper.sync.time.ms`	This property defines the time it takes to sync ZooKeeper with ZooKeeper leader (in milliseconds).	`2000`
`auto.commit.interval.ms`	This property defines the frequency (in milliseconds) for the consumed offsets to get committed to ZooKeeper.	`60 * 1000`

Summary

In this chapter, we have learned how to write basic consumers and learned about some advanced levels of Java consumers that consume messages from partitions.

In the next chapter, we will learn how to integrate Kafka with Storm and Hadoop.

7
Kafka Integrations

Consider a use case for a website where continuous security events, such as user authentication and authorization to access secure resources need to be tracked, and decisions need to be taken in real time for any security breach. Using any typical batch-oriented data processing systems, such as Hadoop, where all the data needs to be collected first and then processed to find out patterns, will make it too late to decide whether there is any security threat to the web application or not. Hence, this is the classical use case for real-time data processing.

Let's consider another use case, where raw clickstreams generated by customers through website usage are captured and preprocessed. Processing these clickstreams provides valuable insight into customer preferences and these insights can be coupled later with marketing campaigns and recommendation engines to offer an analysis of consumers. Hence, we can simply say that this large amount of clickstream data stored on Hadoop will get processed by Hadoop MapReduce jobs in batch mode, not in real time.

In this chapter, we shall be exploring how Kafka can be integrated with the following technologies to address different use cases, such as real-time processing using Storm and batch processing using Hadoop:

- Kafka integration with Storm
- Kafka integration with Hadoop

So let's start.

Kafka integration with Storm

Processing small amount of data at real time was never a challenge using technologies, such as the **Java Messaging Service** (**JMS**); however, these processing systems show performance limitations while dealing with large volumes of streaming data. Also, these systems are not good scalable solutions.

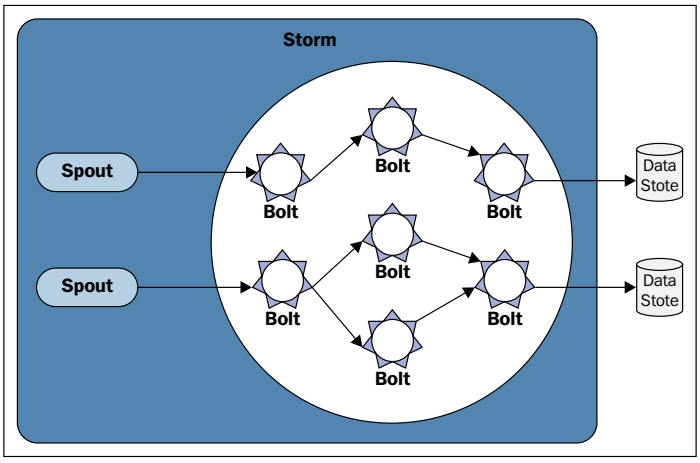
Introduction to Storm

Storm is an open source, distributed, reliable, and fault-tolerant system for processing streams of large volumes of data in real time. It supports many use cases, such as real-time analytics, online machine learning, continuous computation, and **Extract Transformation Load** (**ETL**) paradigm.

There are various components that work together for streaming data processing, which are as follows:

- **Spout**: This is a source of stream, which is a continuous stream of log data.
- **Bolt**: The spout passes the data to a component called **bolt**. A bolt consumes any number of input streams, does some processing, and possibly emits new streams. For example, emitting a stream of trend analysis by processing a stream of tweets.

The following diagram shows spout and bolt in the Storm architecture:



We can assume a Storm cluster to be a chain of bolt components, where each bolt performs some kind of transformation on the data streamed by the spout.

Next in the Storm cluster, jobs are typically referred to as **topologies**; the only difference is that these topologies run forever. For real-time computation on Storm, topologies, which are nothing but graphs of computation, are created. Typically, topologies define how data will flow from spouts through bolts. These topologies can be transactional or non-transactional in nature.

Complete information about Storm can be found at `http://storm-project.net/`.

The following section is useful if you have worked with Storm or have working knowledge of Storm.

Integrating Storm

We have already learned in the previous chapters that Kafka is a high-performance publisher-subscriber-based messaging system with highly scalable properties. Kafka Spout is available for integrating Storm with Kafka clusters.

The source code for Kafka Storm Spout is available at `https://github.com/nathanmarz/storm-contrib/tree/master/storm-kafka`.

The Kafka Spout is a regular spout implementation that reads the data from a Kafka cluster. It requires the following parameters to get connected to the Kafka cluster:

- List of Kafka brokers
- Number of partitions per host
- Topic name used to pull the message
- Root path in ZooKeeper, where Spout stores the consumer offset
- ID for the consumer required for storing the consumer offset in ZooKeeper

The following code sample shows the `KafkaSpout` class instance initialization with the previous parameters:

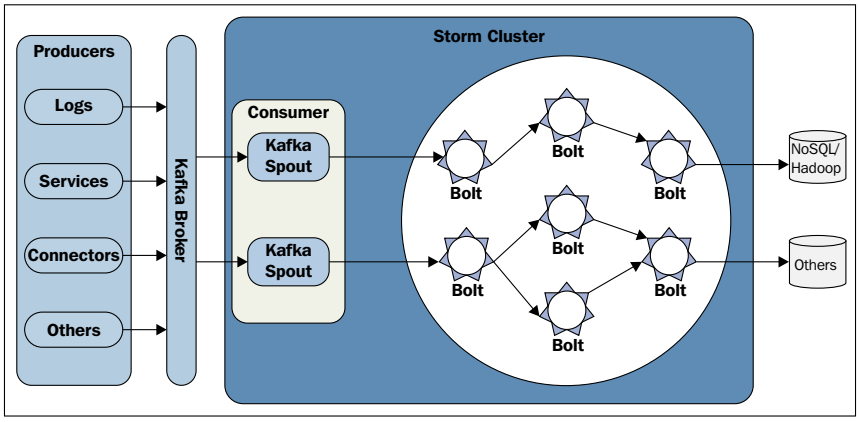
```
SpoutConfig spoutConfig = new SpoutConfig(
  ImmutableList.of("localhost:9092", "localhost:9093"),
  2,
  " othertopic",
  "/kafkastorm",
  "consumID");
KafkaSpout kafkaSpout = new KafkaSpout(spoutConfig);
```

The Kafka Spout uses ZooKeeper to store the states of the message offset and segment consumption tracking if it is consumed. These offsets are stored at the root path specified for the ZooKeeper. By default, Storm uses its own ZooKeeper cluster for storing the message offset, but other ZooKeeper clusters can also be used by setting it in Spout configuration. As per the design, Spout works in a single-threaded model, as all the parallelism is handled by the Storm cluster. It also has a provision to rewind to a previous offset rather than starting from the last saved offset.

Kafka Spout also provides an option to specify how Spout fetches messages from a Kafka cluster by setting properties, such as buffer sizes and timeouts.

To run Kafka with Storm, clusters for both Storm and Kafka need to be set up and should be running. A Storm cluster setup is out of the scope of this book.

The following diagram shows the high-level integration view of what a Kafka Storm working model will look like:



Kafka integration with Hadoop

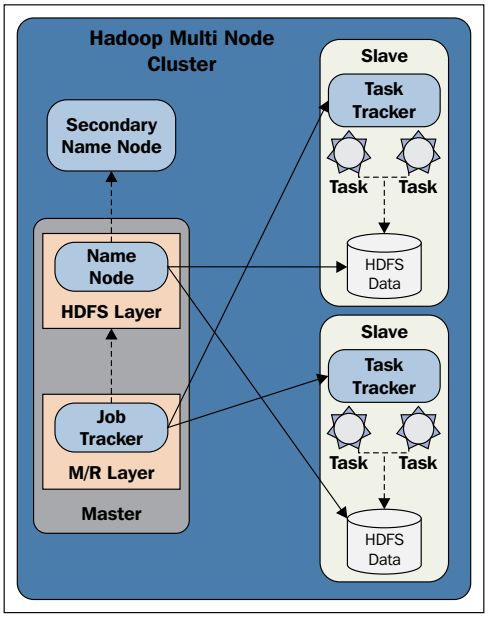
Resource sharing, stability, availability, and scalability are a few of the many challenges of distributed computing. Nowadays, an additional challenge is to process extremely large volumes of data in TBs or PBs.

Introduction to Hadoop

Hadoop is a large-scale distributed batch processing framework which parallelizes data processing across many nodes and addresses the challenges for distributed computing, including big data.

Hadoop works on the principle of the MapReduce framework (introduced by Google), which provides a simple interface for the parallelization and distribution of large-scale computations. Hadoop has its own distributed data filesystem called **HDFS** (**Hadoop Distributed File System**). In any typical Hadoop cluster, HDFS splits the data into small pieces (called **blocks**) and distributes it to all the nodes. HDFS also creates the replication of these small pieces of data and stores it to make sure that if any node is down, the data is available from another node.

The following diagram shows the high-level view of a multinode Hadoop cluster:



Hadoop has the following main components:

- **Name Node**: This is a single point of interaction for HDFS. A name node stores the information about the small pieces (blocks) of data that are distributed across the node.
- **Secondary Name Node**: This node stores the edit logs, which are helpful to restore the latest updated state of HDFS in case of a name node failure.
- **Data Node**: These nodes store the actual data distributed by the name node in blocks and also store the replicated copy of data from other nodes.
- **Job Tracker**: This is responsible for splitting the MapReduce jobs into smaller tasks.
- **Task Tracker**: The task tracker is responsible for the execution of tasks split by the job tracker.

The data nodes and the task tracker share the same machines, MapReduce jobs split, and execution of tasks is done based on the data store location information provided by the name node.

Complete information about Hadoop can be found at `http://hadoop.apache.org/`.

Integrating Hadoop

This section is useful if you have worked with Hadoop or have working knowledge of Hadoop.

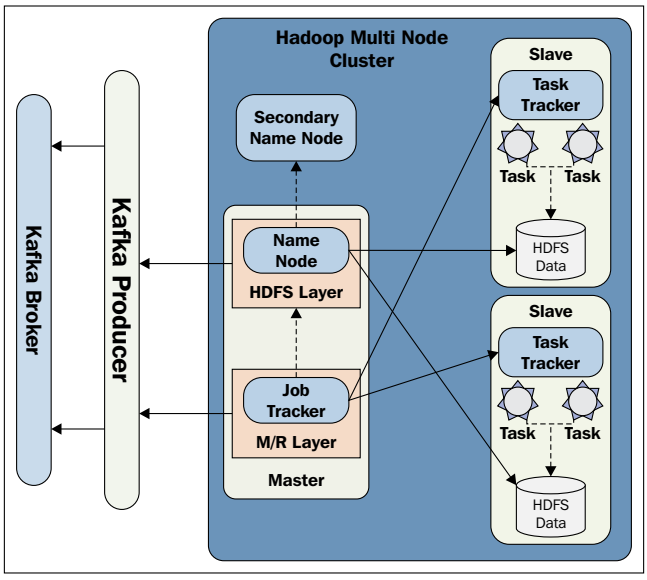
For real-time publish-subscribe use cases, Kafka is used to build a pipeline that is available for real-time processing or monitoring and to load the data into Hadoop, NoSQL, or data warehousing systems for offline processing and reporting.

Kafka provides the source code for both the Hadoop producer and consumer, under its `contrib` directory.

This section only discusses the source code provided with Kafka codebases for Hadoop; no other third-party solution for Kafka and Hadoop integrations is discussed.

Hadoop producer

A Hadoop producer provides a bridge for publishing the data from a Hadoop cluster to Kafka, as shown in the following diagram:



For a Kafka producer, Kafka topics are considered as URIs, and to connect to a specific Kafka broker, URIs are specified as follows:

```
kafka://<kafka-broker>/<kafka-topic>
```

The Hadoop producer code suggests two possible approaches for getting the data from Hadoop:

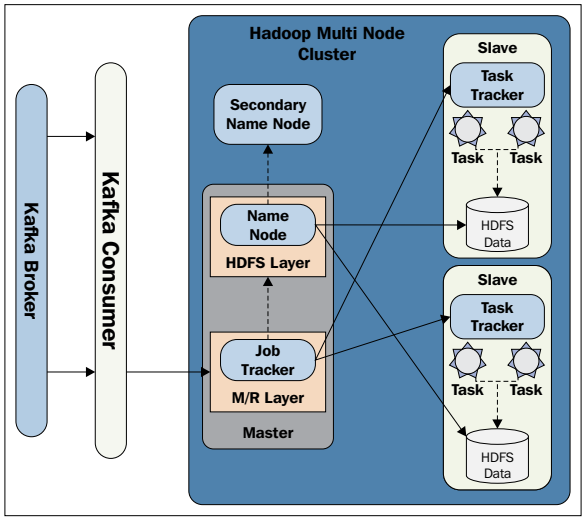
- **Using the Pig script and writing messages in Avro format**: In this approach, Kafka producers use **Pig** scripts for writing data in a binary **Avro** format, where each row signifies a single message. For pushing the data into the Kafka cluster, the `AvroKafkaStorage` class (extends Pig's `StoreFunc` class) takes the Avro schema as its first argument and connects to the Kafka URI. Using the `AvroKafkaStorage` producer, we can also easily write to multiple topics and brokers in the same Pig script-based job.
- **Using the Kafka OutputFormat class for jobs**: In this approach, the Kafka `OutputFormat` class (extends Hadoop's `OutputFormat` class) is used for publishing data to the Kafka cluster. This approach publishes messages as bytes and provides control over output by using low-level methods of publishing. The Kafka `OutputFormat` class uses the `KafkaRecordWriter` class (extends Hadoop's `RecordWriter` class) for writing a record (message) to a Hadoop cluster.

For Kafka producers, we can also configure Kafka producer parameters and Kafka broker information under a job's configuration.

For more detailed usage of the Kafka producer, refer to `README` under the `Kafka-0.8/contrib/hadoop-producer` directory.

Hadoop consumer

A Hadoop consumer is a Hadoop job that pulls data from the Kafka broker and pushes it into HDFS. The following diagram shows the position of a Kafka consumer in the architecture pattern:



A Hadoop job performs parallel loading from Kafka to HDFS, and the number of mappers for loading the data depends on the number of files in the input directory. The output directory contains data coming from Kafka and the updated topic offsets. Individual mappers write the offset of the last consumed message to HDFS at the end of the map task. If a job fails and jobs get restarted, each mapper simply restarts from the offsets stored in HDFS.

The ETL example provided in the `Kafka-0.8/contrib/hadoop-consumer` directory demonstrates the extraction of Kafka data and loading it to HDFS.

For more information on the detailed usage of a Kafka consumer, refer to `README` under the `Kafka-0.8/contrib/hadoop-consumer` directory.

Summary

In this chapter, we have learned how Kafka integration works for both Storm and Hadoop to address real-time and batch processing needs.

In the next chapter, which is also the last chapter of this book, we will look at some of the other important facts about Kafka.

8
Kafka Tools

In this last chapter, we will be exploring tools available in Kafka and its integration with third-party tools. We will also be discussing in brief the work taking place in the area of performance testing of Kafka.

The main focus areas for this chapter are:

- Kafka administration tools
- Integration with other tools
- Kafka performance testing

Kafka administration tools

There are a number of tools or utilities provided by Kafka 0.8 to administrate features such as replication and topic creation. Let's have a quick look at these tools.

Kafka topic tools

By default, Kafka creates the topic with a default number of partitions and replication factor (the default value is `1` for both). But in real-life scenarios, we may need to define the number of partitions and replication factors more than once.

The following is the command for creating the topic with specific parameters:

```
[root@localhost kafka-0.8]# bin/kafka-create-topic.sh --zookeeper
localhost:2181 --replica 3 --partition 2 --topic kafkatopic
```

Kafka also provides the utility for finding out the list of topics within the Kafka server. The List Topic tool provides the listing of topics and information about their partitions, replicas, or leaders by querying ZooKeeper.

The following is the command for obtaining the list of topics:

```
[root@localhost kafka-0.8]#bin/kafka-list-topic.sh --zookeeper
localhost:2181
```

On execution of the above command, you should get an output as shown in the following screenshot:

```
[2013-08-28 10:26:53,437] INFO Client environment:os.version=2.6.18-348.6.1.el5 (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:26:53,437] INFO Client environment:user.name=root (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:26:53,437] INFO Client environment:user.home=/root (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:26:53,437] INFO Client environment:user.dir=/opt/kafka-0.8_OLD (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:26:53,438] INFO Initiating client connection, connectString=localhost:2181 sessionTimeout=30000 w
atcher=org.I0Itec.zkclient.ZkClient@788ab708 (org.apache.zookeeper.ZooKeeper)
[2013-08-28 10:26:53,464] INFO Starting ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2013-08-28 10:26:53,474] INFO Opening socket connection to server localhost/127.0.0.1:2181 (org.apache.zookeepe
r.ClientCnxn)
[2013-08-28 10:26:53,494] INFO Socket connection established to localhost/127.0.0.1:2181, initiating session (or
g.apache.zookeeper.ClientCnxn)
[2013-08-28 10:26:53,520] INFO Session establishment complete on server localhost/127.0.0.1:2181, sessionid = 0x
140c5ef3917000b, negotiated timeout = 30000 (org.apache.zookeeper.ClientCnxn)
[2013-08-28 10:26:53,525] INFO zookeeper state changed (SyncConnected) (org.I0Itec.zkclient.ZkClient)
topic: kafkatopic       partition: 0    leader: 1       replicas: 2,3,1 isr: 1,2
topic: kafkatopic       partition: 1    leader: 1       replicas: 3,1,2 isr: 1,2
topic: othertopic       partition: 0    leader: 2       replicas: 2,1   isr: 2,1
[2013-08-28 10:26:54,325] INFO Terminate ZkClient event thread. (org.I0Itec.zkclient.ZkEventThread)
[2013-08-28 10:26:54,341] INFO EventThread shut down (org.apache.zookeeper.ClientCnxn)
[2013-08-28 10:26:54,341] INFO Session: 0x140c5ef3917000b closed (org.apache.zookeeper.ZooKeeper)
[root@localhost kafka-0.8]#
```

The above console output shows that we can get the information about the topic and partitions that have replicated data. The output from the previous screenshot can be explained as follows:

- `leader` is a randomly selected node for a specific portion of the partitions and is responsible for all reads and writes for this partition
- `replicas` represents the list of nodes that holds the log for a specified partition
- `isr` represents the subset of in-sync replicas' list that is currently alive and in sync with the leader

Note that `kafkatopic` has two partitions (partitions `0` and `1`) with three replications, whereas `othertopic` has just one partition with two replications.

Kafka replication tools

For better management of replication features, Kafka provides tools for selecting a replica lead and controlling shut down of brokers.

As we have learned from Kafka design, in replication, multiple partitions can have replicated data, and out of these multiple replicas, one replica acts as a lead, and the rest of the replicas act as in-sync followers of the lead replica. In case of non-availability of a lead replica, maybe due to broker shutdown, a new lead replica needs to be selected.

For scenarios such as shutting down of the Kafka broker for maintenance activity, election of the new leader is done sequentially, and this causes significant read/write operations at ZooKeeper. In any big cluster with many topics/partitions, sequential election of lead replicas causes delay in availability.

To ensure high availability, Kafka provides tools for a controlled shutdown of Kafka brokers. If the broker has the lead partition shut down, this tool transfers the leadership proactively to other in-sync replicas on another broker. If there is no in-sync replica available, the tool will fail to shut down the broker in order to ensure no data is lost.

The following is the format for using this tool:

```
[root@localhost kafka-0.8]# bin/kafka-run-class.sh kafka.admin.
ShutdownBroker --zookeeper <zookeeper_host:port/namespace> --broker
<brokerID>
```

The ZooKeeper host and the broker ID that need to be shut down are mandatory parameters. We can also specify the number of retries (`--num.retries`, default value `0`) and retry interval in milliseconds (`--retry.interval.ms`, default value `1000`) with a controlled shutdown tool.

Next, in any big Kafka cluster with many brokers and topics, Kafka ensures that the lead replicas for partitions are equally distributed among the brokers. However, in case of shutdown (controlled as well) or broker failure, this equal distribution of lead replicas may get imbalanced within the cluster.

Kafka provides a tool that is used to maintain the balanced distribution of lead replicas within the Kafka cluster across available brokers.

The following is the format for using this tool:

```
[root@localhost kafka-0.8]# bin/kafka-preferred-replica-election.sh
--zookeeper <zookeeper_host:port/namespace>
```

This tool retrieves all the topic partitions for the cluster from ZooKeeper. We can also provide the list of topic partitions in a JSON file format. It works asynchronously to update the ZooKeeper path for moving the leader of partitions and to create a balanced distribution.

For detailed explanation on Kafka tools and their usage, please refer to `https://cwiki.apache.org/confluence/display/KAFKA/Replication+tools`.

Integration with other tools

This section discusses the contributions by many contributors, providing integration with Apache Kafka for various needs such as logging, packaging, cloud integration, and Hadoop integration.

Camus (`https://github.com/linkedin/camus`) is another art of work done by LinkedIn, which provides a pipeline from Kafka to HDFS. Under this project, a single MapReduce job performs the following steps for loading data to HDFS in a distributed manner:

1. As a first step, it discovers the latest topics and partition offsets from ZooKeeper.
2. Each task in the MapReduce job fetches events from the Kafka broker and commits the pulled data along with the audit count to the output folders.
3. After the completion of the job, final offsets are written to HDFS, which can be further consumed by subsequent MapReduce jobs.
4. Information about the consumed messages is also updated in the Kafka cluster.

Some other useful contributions are:

- Automated deployment and configuration of Kafka and ZooKeeper on Amazon (`https://github.com/nathanmarz/kafka-deploy`)
- Logging utility (`https://github.com/leandrosilva/klogd2`)
- REST service for Mozilla Matrics (`https://github.com/mozilla-metrics/bagheera`)
- Apache Camel-Kafka integration (`https://github.com/BreizhBeans/camel-kafka/wiki`)

For a detailed list of Kafka ecosystem tools, please refer to `https://cwiki.apache.org/confluence/display/KAFKA/Ecosystem`.

Kafka performance testing

Kafka contributors are still working on performance testing, and their goal is to produce a number of script files that help in running the performance tests. Some of them are provided in the Kafka `bin` folder:

- `Kafka-producer-perf-test.sh`: This script will run the `kafka.perf.ProducerPerformance` class to produce the incremented statistics into a CSV file for the producers
- `Kafka-consumer-perf-test.sh`: This script will run the `kafka.perf.ConsumerPerformance` class to produce the incremented statistics into a CSV file for the consumers

Some more scripts for pulling the Kafka server and ZooKeeper statistics are provided in the CSV format. Once CSV files are produced, the R script can be created to produce the graph images.

For detailed information on how to go for Kafka performance testing, please refer to `https://cwiki.apache.org/confluence/display/KAFKA/Performance+testing`.

Summary

In this chapter, we have added some more information about Kafka, such as its administrator tools, its integration, and Kafka non-Java clients.

During this complete journey through Apache Kafka, we have touched upon many important facts about Kafka. We have learned the reason why Kafka was developed, its installation, and its support for different types of clusters. We also explored the design approach of Kafka, and wrote few basic producers and consumers.

In the end, we discussed its integration with technologies such as Hadoop and Storm.

The journey of evolution never ends.

Index

G

H

I

J

K

L

M

N

O

P

R

S

T

Z

Thank you for buying

Apache Kafka

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: `www.packtpub.com`.

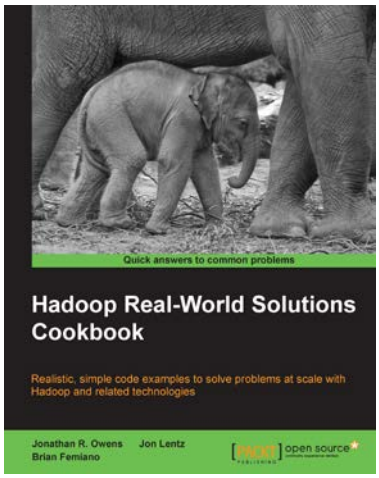
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



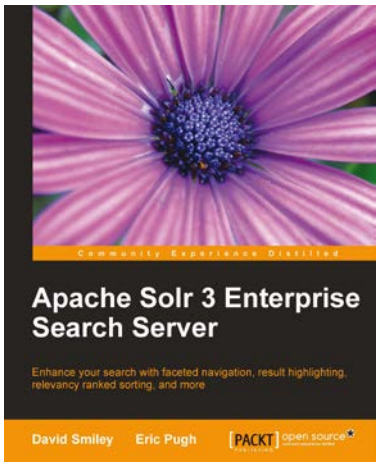
Hadoop Real-World Solutions Cookbook

ISBN: 978-1-84951-912-0 Paperback: 316 pages

Realistic, simple code examples to solve problems at scale with Hadoop and related technologies

1. Solutions to common problems when working in the Hadoop environment
2. Recipes for (un)loading data, analytics, and troubleshooting
3. In depth code examples demonstrating various analytic models, analytic solutions, and common best practices

---



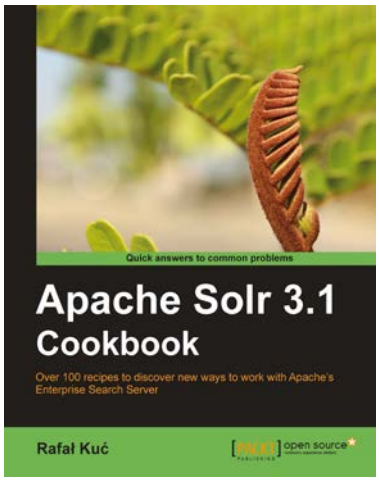
Apache Solr 3 Enterprise Search Server

ISBN: 978-1-84951-606-8 Paperback: 418 pages

Enhance your search with faceted navigation, result highlighting, relevancy ranked sorting, and more

1. Comprehensive information on Apache Solr 3 with examples and tips so you can focus on the important parts
2. Integration examples with databases, web-crawlers, XSLT, Java & embedded-Solr, PHP & Drupal, JavaScript, Ruby frameworks
3. Advice on data modeling, deployment considerations to include security, logging, and monitoring, and advice on scaling Solr and measuring performance

[PACKT] open source
community experience distilled
PUBLISHING



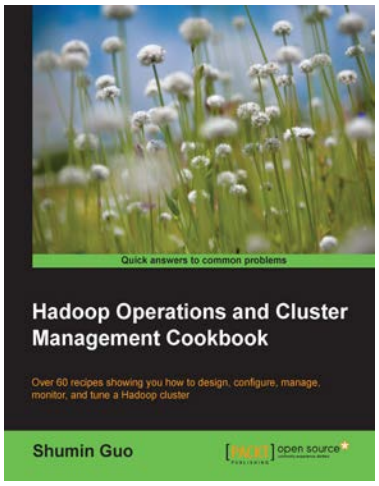
Apache Solr 3.1 Cookbook

ISBN: 978-1-84951-218-3 Paperback: 300 pages

Over 100 recipes to discover new ways to work with Apache's Enterprise Search Server

1. Improve the way in which you work with Apache Solr to make your search engine quicker and more effective
2. Deal with performance, setup, and configuration problems in no time
3. Discover little-known Solr functionalities and create your own modules to customize Solr to your company's needs

---



Hadoop Operations and Cluster Management Cookbook

ISBN: 978-1-78216-516-3 Paperback: 368 pages

Over 60 recipes showing you how to design, configure, manage, monitor, and tune a Hadoop cluster

1. Hands-on recipes to configure a Hadoop cluster from bare metal hardware nodes
2. Practical and in depth explanation of cluster management commands
3. Easy-to-understand recipes for securing and monitoring a Hadoop cluster, and design considerations