Author Picks

Reactive Data Handling

Chapters selected by

Manuel Bernhardt

manning

Reactive Data Handling

Selections by Manuel Bernhardt

Manning Author Picks

To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN 9781617294198
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

contents

Introduction

Web applications play an increasingly important role in all facets of our lives. We depend on our applications to be always available and to provide us with up-to-the-second data. This shift toward real-time data processing is also a key aspect of the Internet of Things, which the Gartner Group predicts by 2020 will include 26 billion actively-connected physical devices sending, receiving, and processing streams. That's a lot of data.

The reactive application architecture is an answer to the requirements of high availability and resource efficiency. To provide these benefits for real-time data processing, reactive applications:

- need to handle high and varying loads so they remain responsive to users;
- can scale in and out, depending on demand, to make use of more or less hardware resources;
- are built with supervision and recovery mechanisms in place to manage and recover from failure;
- rely on asynchronous message-passing as a primary means of communication.

Reactive applications need to ensure that these core principles are applied across the entire stack, from resource management of servers in datacenters up to communication with a user's browser or native application.

These selected chapters introduce technologies and principles you can apply when building reactive applications capable of handling real-time processing with large data loads. This book starts with the high-level architecture of reactive applications and then looks into low-level practical aspects. After you read these chapters, you'll understand the benefits of using the reactive application architecture to manage and process vast quantities of data at a fast pace.

Analyzing streaming data

One core principle of reactive web applications is considering data as a dynamic stream rather than as a static reservoir. The following chapter introduces how to analyze streaming data. It covers the distributed stream-processing architecture as well as a few of the common frameworks available. It should give you an understanding of the high-level architecture and the common concerns of distributed data-streaming systems.

Chapter 4 from *Streaming Data*
by Andrew G. Psaltis

Analyzing streaming data

This chapter covers

- In-flight data analysis
- The common stream-processing architecture
- Key features common to stream-processing frameworks

In the previous chapter we spent time understanding and thinking through the importance of the message queueing tier. Remember that tier is designed to gather data from the collection tier and make it available to be moved through the rest of the streaming architecture. At this point the data is ready and waiting for us to consume and do magic with. In this chapter you're going to learn about the analysis tier. Our goal is to develop an understanding of the underlying principles of this tier, and then in the next chapter we'll dive into the different ways to use this tier to perform magic on the data. With that frame of reference in mind, consult our navigational aid in figure 4.1 to make sure you're oriented with respect to the flow of data.

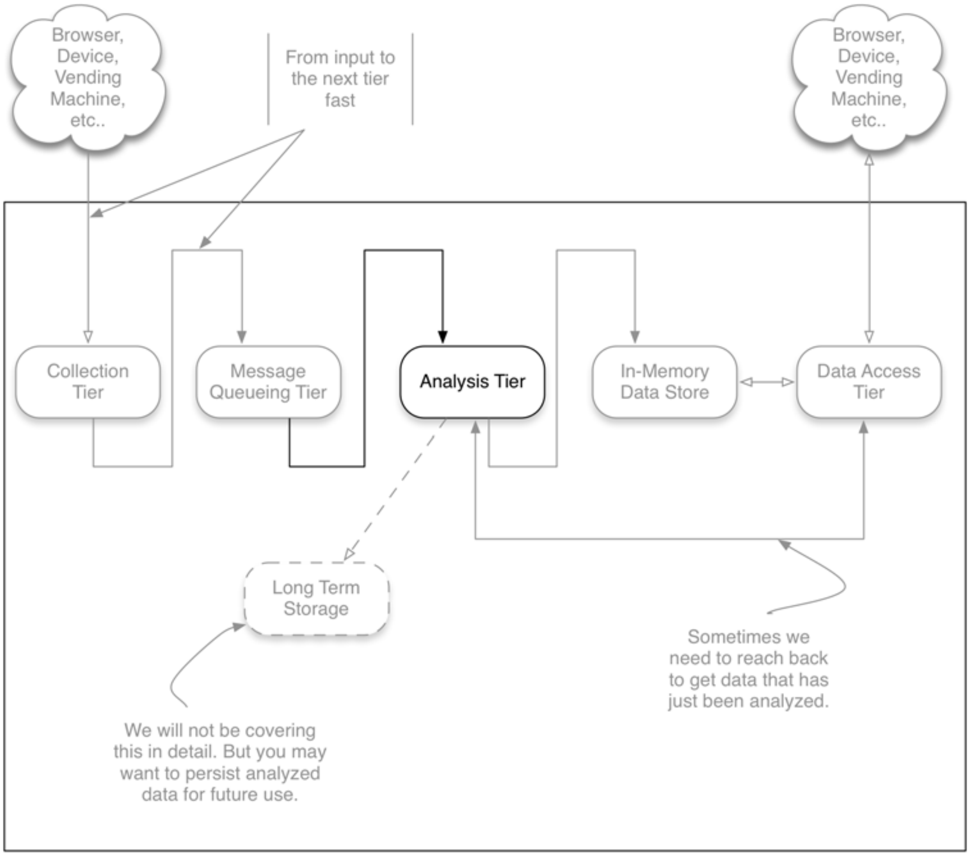

Browser, Device, Vending Machine, etc..
From input to the next tier fast
Browser, Device, Vending Machine, etc..
Collection Tier
Message Queueing Tier
Analysis Tier
In-Memory Data Store
Data Access Tier
Long Term Storage
Sometimes we need to reach back to get data that has just been analyzed.
We will not be covering this in detail. But you may want to persist analyzed data for future use.

Figure 4.1 The streaming data architecture with the analysis tier in focus

One thing that you may notice in figure 4.1 is that unlike in the previous chapter where we discussed the input and output of the data, in this chapter we're only going to concern ourselves with the input. The reason for this is simple: our goal is to understand the core underpinnings of this tier, and in the next chapter we'll discuss the ways we can work with the data in this tier. Therefore, we'll hold off on talking about where the data goes from this tier until next chapter. After finishing this chapter you'll have an understanding of the core concepts found in all the modern tools used for this tier and be ready to learn how to perform various operations on the data. All right, grab a quick coffee refill, and let's get going.

4.1 Understanding in-flight data analysis

A key to understanding the features we'll discuss in this chapter is first coming to grips with what in-flight data means and the concept of *continuous queries*. If the term *in-flight* makes you think of something that's in the air moving and not touching the ground, that's the right idea. When it comes to data, *in-flight* refers to all the tuples in

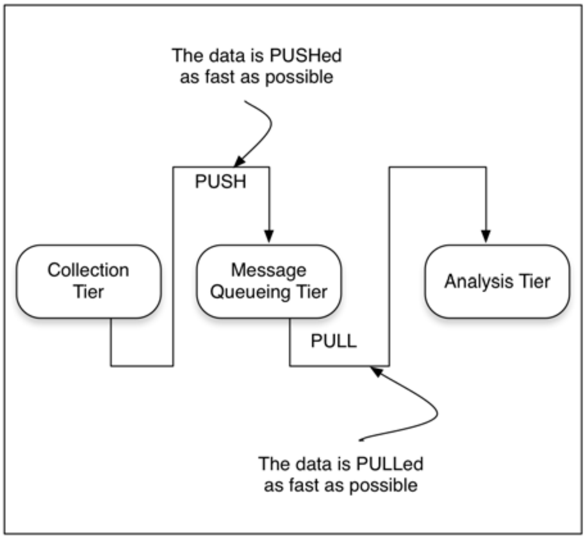


Figure 4.2 Data being pushed from the collection tier and pulled from the analysis tier

the system from the input source (the message queueing tier) to the output to a client (the next tier), the idea being that the data is always in motion and never at rest, meaning that it's never persisted to durable storage. If you haven't heard of the term *data at rest*, don't worry; it's just a fancy way of saying that the data is stored on disk or another storage medium. Take a look at figure 4.2, which shows how this plays out in our streaming architecture.

Looking at figure 4.2, it should be clear that our goal in this tier is to *pull* the data from the message queueing tier as fast as possible; ideally the analysis tier should be able to keep up with the rate at which the collection tier is pushing data into the message queueing tier. So how is this different from a non-streaming system, say one built with a traditional DBMS (RDBMS, Hadoop, HBase, Cassandra, and so on)? In those non-streaming types of systems the data is at rest and we query it for answers. In a streaming system we turn that on its head and the data is moved through the query. This model is referred to as the Continuous Query model, meaning that the query is constantly being evaluated as new data arrives.

Let's imagine for a moment that you run a very large news agency and you want to know if an article is trending or if a link to it is broken so that you can adjust your marketing campaign or fix your site. If you were using traditional DBMS technologies, you would have to do the following:

1. Gather the data from your site.
2. Load the data into the DBMS.
3. Execute a query to determine if the link is broken or the article is trending.
4. Take action.
5. Rinse and repeat every X minutes or most likely hours.

Now, let's compare that to the steps you might take if you were using a streaming system:

1. Collect the stream of data.
2. Start a query that determines if the link is broken or the article is trending.
3. Take action.

I think you'll agree that it would be very hard for your business to react to changes happening now using a traditional system, whereas with the streaming system the query is always executing against the data and you can react in real time to trends or problems. In a streaming system a user (or application) registers a query that's executed every time data arrives or at a predetermined time interval. The result of the query is then pushed to the client. The key differences to remember here are these:

- In the architecture of traditional database management systems when a user (the active party) wants an answer to a question, she submits a query to the system (the passive party) and an answer is returned. This is always based on data that has been loaded into the system before it is queried—in essence, the data set is static.
- In a streaming system a query is started and is continually (this could be triggered on an interval or another event) executed over the data as it is flowing. The answer to the query is then pushed to the next tier, which may be a user or application.

This inverts the traditional data management model by assuming users to be passive and the data management system to be active. Figure 4.3 shows this inversion graphically.

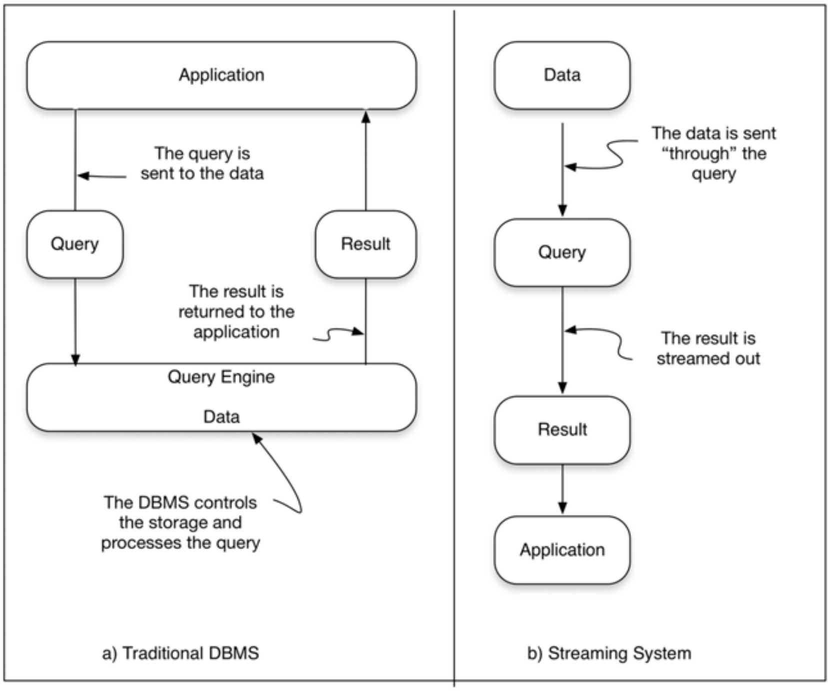

Application
The query is sent to the data
Query
Result
The result is returned to the application
Query Engine
Data
The DBMS controls the storage and processes the query
a) Traditional DBMS
Data
The data is sent "through" the query
Query
The result is streamed out
Result
Application
b) Streaming System

Figure 4.3 Turning things on their head, non-streaming versus streaming

Looking at figure 4.3a, you can see that with the traditional DBMS the query is sent to the data and executed, and the result is returned to the application. In a streaming system, as illustrated in figure 4.3b, this is model is completely changed and the data is sent "through" the query and the result is then sent to an application. In the case of the streaming system the data is being pulled or pushed through our system in a never-ending stream; this undoubtedly has implications on both the design and the way we query these systems. To give you a better feel for these differences, table 4.1 highlights some of the main differences between a traditional DBMS and streaming system.

Table 4.1 Comparison of traditional DBMS to streaming system

	DBMS	Streaming System
Query model	Queries are based on a one-time model and a consistent state of the data. In a one-time model, the user executes a query and gets an answer, and the query is forgotten. This is a pull model.	The query is continuously executed based on the data that is flowing into the system. A user registers a query once, and the results are regularly pushed to the client.
Changing data	During down time the data can't change.	Many stream applications continue to generate data while the streaming analysis tier is down, possibly requiring a catch up following a crash.
Query state	If the system crashes while a query is being executed, it is forgotten and is the responsibility of the application (or user) to re-issue the query when the system comes back up.	Registered continuous queries may or may not need to continue where they left off. Many times it's as if they never stopped in the first place.

If you sit back and think of all the data zipping around you all day long, from the myriad of connected devices and appliances to online activity, the questions you could ask and problems you could solve if it all passed through a streaming analysis tier are amazing. Here are some categories and examples to get you going:

- *Tracking behavior*—Imagine being able to provide personalized advertising based on a customer's location, the weather, and their previous buying habits and preferences. McDonald's did just this using the VMob platform. According to this (http://blogs.microsoft.com/iot/2015/01/12/boosting-retail-sales-with-iot-powered-customer-engagement/) case study, McDonald's in the Netherlands realized a 700% increase in offer redemptions, and customers using the app returned twice as often and spent on average 47% more.
- *Improving traffic safety and efficiency*—According to the European Commission (http://ec.europa.eu/transport/themes/urban/urban_mobility/index_en.htm), congestion in the European Union (EU) in and around urban areas costs nearly €100 billion or 1% of EU GDP annually. According to the Federal Highway Administration (http://ops.fhwa.dot.gov/program_areas/reduce-non-cong.htm), 25% of traffic congestion is nonrecurring; it's caused by traffic incidents. Now

imagine you were able to employ roadway vehicle sensors (see http://www.fhwa.dot.gov/policyinformation/pubs/vdstits2007/03.cfm for an introduction); based on our analysis of the traffic data we can provide drivers with updated traffic conditions and reroute traffic accordingly to maximize driving efficiency. For real-world examples take a look at Blip Systems (http://www.blipsystems.com/traffic/); they have examples of how some cities have solved a myriad of traffic problems.

- *Real-time fraud analytics*—Every time a credit card is swiped, a complex series of algorithms must be executed to determine if the attempted transaction is valid or fraudulent. According to FICO (http://www.fico.com/en/node/8140?file=5582), there has been a 70% reduction in U.S. fraud losses on credit cards as a percentage of credit card sales since real-time fraud analytics have been deployed.

These examples are just the tip of the iceberg and hopefully have whet your appetite for what's possible. They also may help you realize that understanding how to build these systems to harness the myriad of data streams available in the world today is becoming an essential skill. But let's not get ahead of ourselves just yet; we have our work cut out for us learning about the core features of an analysis tier. Let's begin our journey by discussing the general architecture of a stream-processing system and then move onto the key features and see how each of the features plays a role in your decision to use a particular framework.

4.2 Distributed stream processing architecture

It may be possible to run an analysis tier on a single computer, but the velocity and volume of the data at some point make this a non-viable option. For example, if instead of tracking trending or broken links to articles, imagine we were interested in analyzing the performance of a gas turbine in real time to determine if it was functioning correctly. According to General Electric, a single turbine engine can produce approximately 1 TB of data per hour. Clearly, using a single computer will quickly become a non-viable option for us. Therefore, we're going to concentrate on the tools and technologies involved in building a distributed analysis tier. As you survey the technology landscape, you'll find various technologies designed for stream processing. At the time of this writing the three most popular open source products are Apache Spark Streaming, Apache Storm, and Apache Samza. We're not going to go into detail on each of them, but we'll discuss each of them briefly after we go over our generalized streaming architecture so you can see how each fits into it.

A GENERALIZED ARCHITECTURE

If you reflect over the figures for Spark, Storm and Samza, I think you'll start to see a pattern. They all have the following three common parts:

- A component that your streaming application is submitted to; this is similar to how Hadoop Map Reduce works. Your application is sent to a node in the cluster that executes your application.

- Separate nodes in the cluster that execute your streaming algorithms.
- Data sources that are the input to the streaming algorithms.

Taking these central ideas and their respective architectures into consideration, we can generalize this into a single common architecture, as shown in figure 4.4.

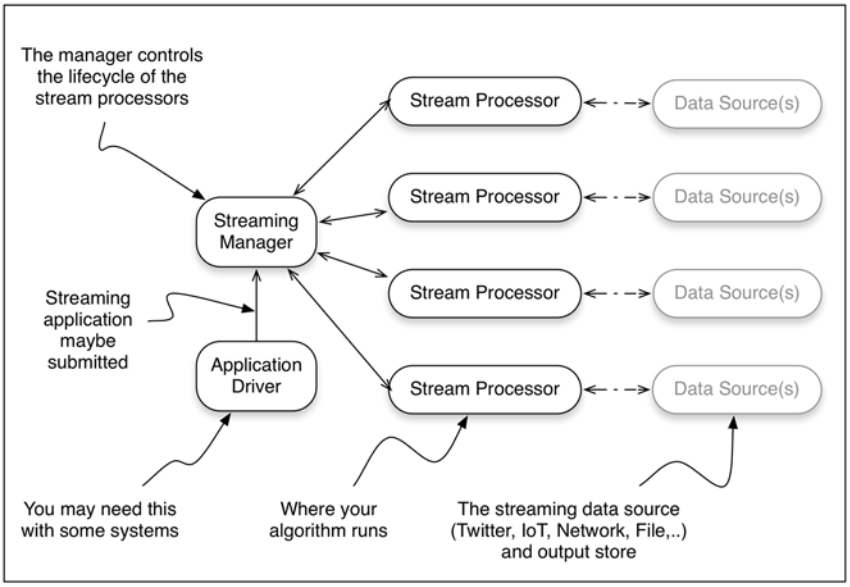

The manager controls the lifecycle of the stream processors
Stream Processor
Data Source(s)
Stream Processor
Data Source(s)
Streaming Manager
Stream Processor
Data Source(s)
Streaming application maybe submitted
Application Driver
Stream Processor
Data Source(s)
You may need this with some systems
Where your algorithm runs
The streaming data source (Twitter, IoT, Network, File,..) and output store

Figure 4.4 Generic streaming analysis architecture you will find with many products on the market

There are other streaming systems on the market today that have not achieved the same level of popularity as those we discussed in the previous section, and undoubtedly there will be more in the future. In many cases other products will map onto this common architecture and help in your understanding of how they work. To make sure we're on the same page, let's briefly discuss the common architectural pieces shown in figure 4.4.

- *Application driver*—With some streaming systems, this will be the client code that defines your streaming programming and communicates with the streaming manager. For example, with Spark Streaming your client code is broken into two logical pieces: the driver and the streaming algorithm(s) or job. The driver submits the job to the streaming manager, may collect results at the end, and controls the lifetime of your job.
- *Streaming manager*—The streaming manager has the general responsibility of getting your streaming job to the stream processor(s); in some cases it will control or request the resources required by the stream processors.
- *Stream processor*—This is really where the rubber meets the road, the place where your job actually runs. Although this may take many shapes based on the

streaming platform in use, the job remains the same: to execute the job that was submitted.

- *Data source(s)*—This represents the input and potentially the output data from your streaming job. With some platforms your job may be able to ingest data from multiple sources in a single job, whereas others may only allow ingestion from a single source. One thing that may not be obvious from the architectures is where the output of the jobs goes. In some cases you may want to collect the data in your driver, whereas in other cases you may wish to write it out to a different data source to be used by another system or as input for another job.

Now that you have an understanding of the various architectures and have boiled them down to our common architecture, let's go back to our example of monitoring the performance of gas turbines to determine if they're functioning correctly and map that to our common architecture. In figure 4.5 you can see our common architecture (simplified so it is less busy) with our business problem mapped to it.

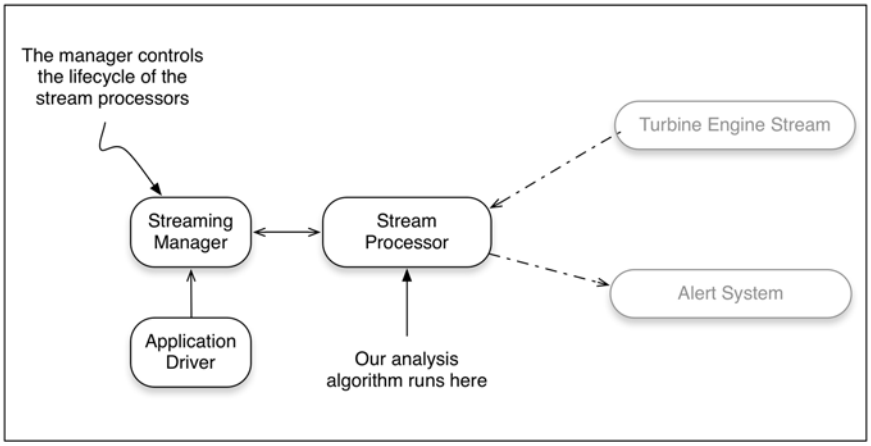

The manager controls the lifecycle of the stream processors
Turbine Engine Stream
Streaming Manager
Stream Processor
Alert System
Application Driver
Our analysis algorithm runs here

Figure 4.5 Turbine engine monitoring on our common architecture

I realize that this may have been a lot to digest, so take a moment to see if you can map your business problem to the common architecture we've derived. When you're ready, we'll discuss the architecture of the three major streaming systems and then go a little deeper and discuss some of the key features you'll want to think about when choosing a stream-processing framework.

Apache Spark Streaming

Apache Spark Streaming, often just called Spark Streaming, is built on Apache Spark, as depicted in figure 4.6.

As you'll notice, there are various other features built on top of Apache Spark. Apache Spark is becoming the de facto platform for general-purpose distributed computation. It provides support for multiple languages (Java, Scala, Python, and R) and

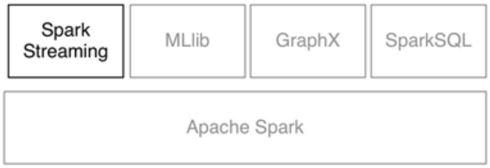


Figure 4.6 Apache Spark Streaming with the basic Spark stack

at the time of this writing has the following high-level tools built on top of it: Spark Streaming, MLlib (Machine Learning), SparkR (integration with R), and GraphX (for graph processing). Outside the normal project documentation, a great resource to start learning more about Spark is Marko Bonaći and Petar Zečević's book *Spark in Action* (https://www.manning.com/books/spark-in-action). Keeping our focus on Spark Streaming, let's take a look at its overall architecture, shown in figure 4.7.

Let's discuss the general flow of how things work so that you have an understanding of its basic architecture. Starting from the left in the figure we have our program, which contains what is called a Spark StreamingContext; collectively this is commonly referred to as *the driver*. Without diving into the details, the Spark StreamingContext contains all of the logic to be able to keep track of incoming data, set up the streaming jobs, schedule them on the Spark workers, and execute the jobs. You may notice here that we're talking about jobs and not a stream. The reason for this is that Spark and subsequently Spark Streaming operate on batches of work. In the case of Spark Streaming these batches represent data over a period of time and can be scheduled to

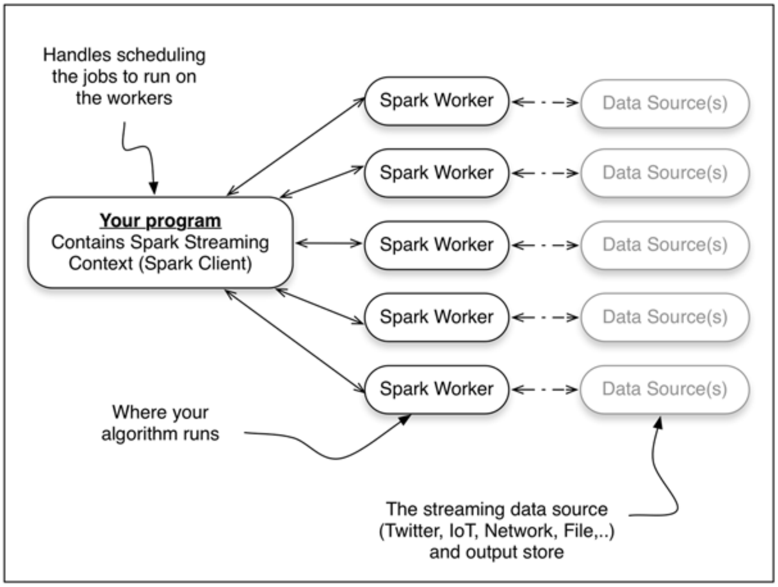

Handles scheduling the jobs to run on the workers
Your program
Contains Spark Streaming Context (Spark Client)
Spark Worker
Spark Worker
Spark Worker
Spark Worker
Spark Worker
Data Source(s)
Data Source(s)
Data Source(s)
Data Source(s)
Data Source(s)
Where your algorithm runs
The streaming data source (Twitter, IoT, Network, File,..) and output store

Figure 4.7 Spark Streaming high-level architecture

run at frequencies less than half a second. A *job* in Spark Streaming is just the logic of your program that's bundled up and passed to the Spark workers. If you've read about or worked with Hadoop Map Reduce, this is the same concept. Moving to the middle of figure 4.7 you see the Spark workers; these run on any number of computers (from one to thousands) and are where your job (your streaming algorithm) is executed. As you'll notice, they receive data from an external data source and communicate with the Spark StreamingContext that's running as part of the driver.

Apache Storm

Apache Storm is tuple-at-a-time stream-processing framework designed for real-time processing of data streams. There are many features to Storm, which we won't cover in detail. For great references to learn more about Storm see Sean Allen, Peter Pathirana, and Matthew Jankowski's *Storm Applied* (Manning 2013). The overall architecture for Storm is shown in figure 4.8.

Looking at figure 4.8, you can see that from a high-level it's very similar to Spark Streaming or perhaps a Hadoop cluster if you change Nimbus to a job tracker and the supervisors to data nodes. Unlike Hadoop and Spark that use the term *job* to describe the unit of work, with Storm the term *topology* is used instead. The reasoning behind this is that a *job* will eventually finish whereas a *topology* will run forever. Let's not get bogged down by this semantic sugar; at the end of the day they both represent a way to deploy your program to the worker nodes. With this definition in mind, let's walk through figure 4.8 and discuss the different pieces.

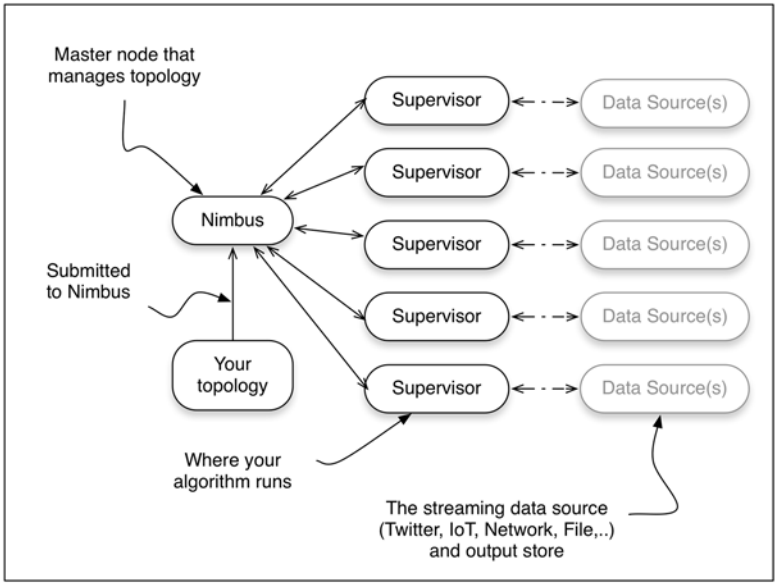


Figure 4.8 High-level overview of Storm architecture showing Nimbus, supervisors, and a data source

Starting on the bottom left you can see the topology; this is submitted to a component called Nimbus. Nimbus is in charge of deciding how the topology is deployed across the supervisors, assigns different tasks to the supervisors, and monitors the entire system for failures. Moving to the middle of the figure you will see the supervisor nodes; these are where your topology actually runs. On the right is the *data source*; this just represents the data that will be ingested by the running topology.

APACHE SAMZA

The streaming model with Apache Samza is slightly different in that it provides a stage-wise stream processing framework. To do this it leverages two prominent technologies found in the Big Data space, those being Apache Yarn and Apache Kafka. We won't spend much time talking about those technologies, but we will discuss them briefly as they relate to the high-level Samza architecture shown in figure 4.9.

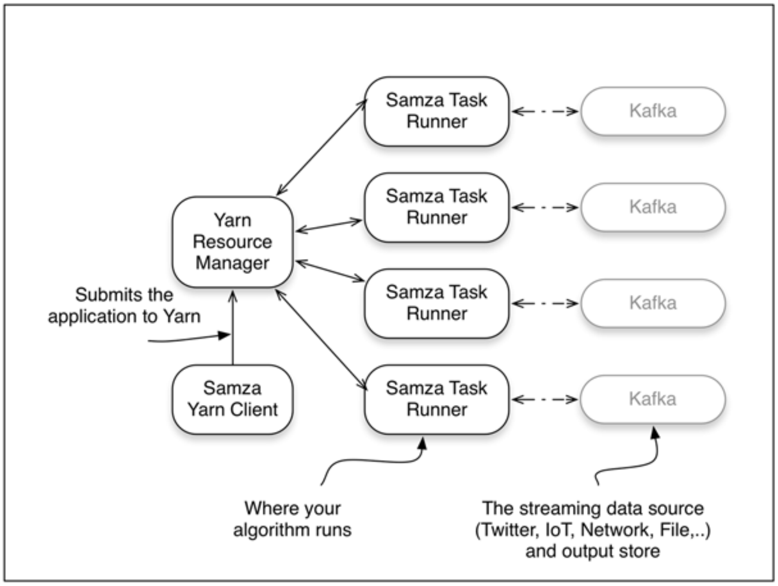

Samza Task Runner
Kafka
Samza Task Runner
Kafka
Yarn Resource Manager
Samza Task Runner
Kafka
Submits the application to Yarn
Samza Yarn Client
Samza Task Runner
Kafka
Where your algorithm runs
The streaming data source (Twitter, IoT, Network, File,..) and output store

Figure 4.9 High-level Apache Samza architecture

As we've done with the others, let's take a minute to walk through this architecture. One of the things that I think will jump right out at you is the new technologies that appear, in particular Yarn and Kafka. Yarn is a cluster manager designed to handle resource management and job scheduling/monitoring. I know that's a mouthful; just think of it this way: the resource management part is responsible for allocating resources (CPUs, memory, disk, network, and so on) for the various applications that are running on a cluster of computers. The job scheduling/monitoring aspect is responsible for actually running the job on the cluster. When looking at figure 4.9 you can see that the Samza Yarn client makes a request to the Yarn resource manager asking that the requested resources be allocated for the Samza application to run. Subsequently after some resource negotiation, the Samza task runners are executed in various nodes in the cluster. This is intentionally simplified, because focusing on the Yarn specifics at this time doesn't add value to our discussion and is subject to change as the project matures. Moving to the center of the figure you can see our Samza tasks running. In this case all input and output to our Samza tasks will be done using Apache Kafka. Apache Kafka is a technology that squarely fits into our discussion in chapter 3 on the message queueing tier and a technology we'll revisit in future chapters. For now you can think of it as a high-speed data store that our streaming tasks will read from and write to. Some great resources to learn more about Yarn are Alex Holme's *Hadoop in Practice*, Second Edition (Manning 2014) and Chuck Lam, Mark Davis, and Ajit Gaddam's book *Hadoop in Action*, Second Edition (Manning 2014). To find the latest information on Apache Samza, please visit http://samza.apache.org.

4.3 Key features of stream-processing frameworks

Many different stream-processing frameworks can be used in the analysis tier of our streaming data architecture. When we boil them down there are a handful of key features that we want to pay special attention to when comparing them and deciding if they're suitable for solving our business problem. In this section we'll discuss the key features you need to pay special attention to; make sure you understand each of them and can apply this knowledge when selecting the stream-processing framework you'll use in your streaming data architecture.

4.3.1 Message delivery semantics

In chapter 3 you learned about message delivery semantics in respect to the message queueing tier and the producers, brokers, and consumers. This time we focus our discussion of message delivery semantics on the analysis tier. The definitions don't change, but you'll notice that the implications are a little different. First, let's refresh your memory on the definitions of the different guarantees:

- *At-most-once*—A message may get lost, but it will never be processed a second time.
- *At-least-once*—A message will never be lost, but it may be processed more than once.
- *Exactly-once*—A message is never lost and will be processed only once.

Those definitions are a slightly more generic version of what you saw before with the message queueing tier. When we discussed these semantics in chapter 3, our focus was on understanding what each of them meant in respect to producing and consuming messages. In this chapter with the stream-processing tools, we're really talking about the continuation of the consumer side of the message queueing tier. So how do these manifest themselves in the stream-processing tools you may use in this tier? Let's overlay them on a data flow diagram and then walk through them to understand what they mean.

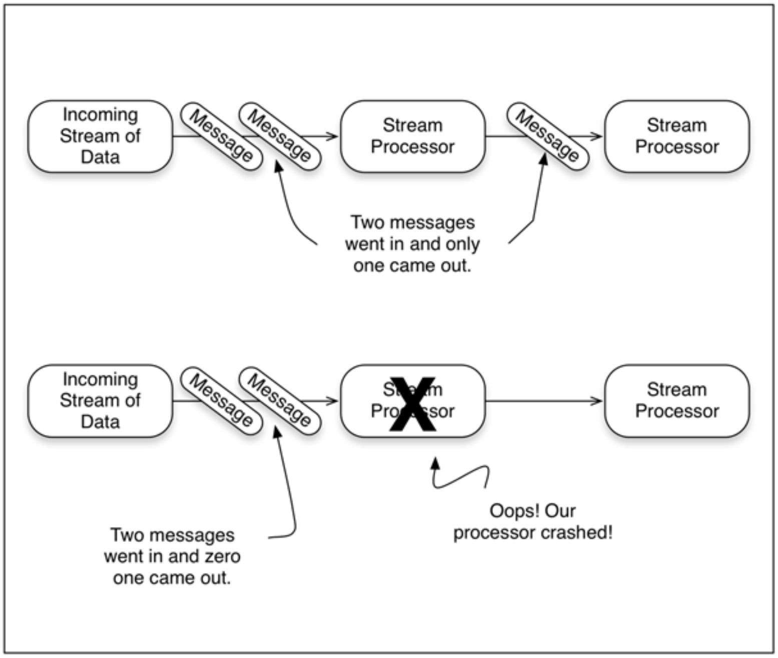


Figure 4.10 At-least-once message delivery shown with the streaming data flow

Figure 4.10 shows at-most-once semantics with the two failure scenarios: a message dropping and a streaming task processor failing. The second scenario, a streaming task processor failing, will also result in message loss until a replacement processor comes online.

At-most-once is the simplest delivery guarantee a system can offer; no special logic is required anywhere. In essence, if a message gets dropped, a stream processor crashes, or the machine that a stream processor is running on fails, the message is lost.

At-least-once increases the complexity because the streaming system must keep track of every message that was sent to the stream processor and an acknowledgement that it was received. If the streaming manager determines that the message was not processed (perhaps it was lost or the stream processor didn't respond within a given time boundary), then it will be resent. It's important to keep in mind that at this level of messaging guarantee your streaming job may be sent the same message multiple times. Therefore, your streaming job must be idempotent, meaning that every time your streaming job receives the same message, it produces the same result. If you keep this in mind when designing your streaming jobs, then you will be able to handle the duplicate-messages situation.

Exactly-once semantics ratchets up the complexity a little more for the stream-processing framework. Besides the bookkeeping that it must keep for all messages that have been sent, it now must also detect and ignore duplicates. With this level of guarantee your streaming job no longer has to worry about dealing with duplicate messages;

it only has to make sure it responds with a success or failure after a message is processed. Even though it's not required of your streaming job to be idempotent with this level of messaging guarantee, I highly recommend that you approach all of your streaming jobs with the expectation that they should be idempotent. It will make troubleshooting and reasoning about them much easier.

You may be wondering which of these guarantees you need; in reality it will depend on the business problem you're trying to solve. Let's take our example from earlier: the turbine engine monitoring system. Remember that for this system we want to constantly analyze how our turbine engine is performing so we can predict when a failure may occur and pre-emptively perform maintenance. Earlier we said our turbines produce approximately 1 TB of data every hour, which may not seem like a lot of data, but keep in mind that is one turbine and we're monitoring thousands to be able to predict when a failure may occur. What do you think; do we need to ensure we don't lose a single message? We may, but it would be worth investigating if our prediction algorithm needs all of the data or not. If it can perform adequately if data is missing, then I'd choose the least complex guarantee first and work from there.

What if instead your business problem involved making a financial transaction based on a streaming query? Perhaps you operate an ad network and you provide real-time billing to your clients. In this case you'd want to ensure that the streaming system you choose provides exactly-once semantics.

I think you get the hang of it and can apply this to your business problem. Now let's move on to talk about state management.

State management

Once your streaming analysis algorithm becomes more complicated than just using the current message without dependencies on any previous messages and/or external data, you'll need to maintain state and will likely need the state management services provided by your framework of choice. Let's take a simple example that we can work with to help you understand where and perhaps how state needs to be managed.

Pretend you're the marketing manager for a large e-commerce site and you want to know the number of page views per hour for each visitor.

I know you're thinking "an hour" that can be done in a batch process. We're not going to worry about that right now; instead, let's focus on the implied state you must keep to satisfy this business question. Figure 4.11 shows how your streaming task processors would be organized to answer this question.

It becomes obvious when looking at figure 4.11 where you need to keep state—right there in the stream processor where your job performs the counting by user ID. If your streaming analysis tool of choice doesn't provide state management capabilities, then one viable option is for you is to keep the data in memory and flush it every hour. This would work as long as you're okay with the potential of losing all the data if the streaming processor or job fails at any time. Of course, as luck would have it, your job would be running smoothly and then one day start to fail at 59 minutes into the hour. Depending on your business case, the risk and possible loss of data by keeping all

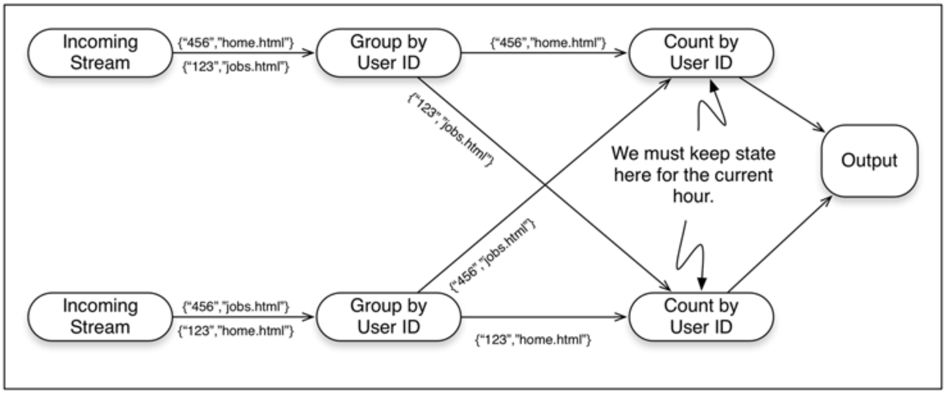


Figure 4.11 Simple example of counting page views per user over an hour

state in memory may be acceptable. But in many business cases life is not so simple and you do need to worry about managing state. To help in these scenarios, many stream-processing frameworks provide state management features that you can leverage.

The state management facilities provided by various systems naturally fall along a complexity continuum, as shown in figure 4.12.

The continuum starts on the left with a naïve in-memory-only choice similar to what we used earlier and progresses to the other end of the spectrum with systems that provide a queryable persistent state that's replicated. If you find yourself saying "these seem like two totally different slants on state management," you're not alone. The solutions on the low-complexity side only solve the problem of maintaining the state of a computation in the face of failures. For the simple operations of keeping a running count current and not losing track of the current value in the face of failure, these systems are a great fit. On the other end of the spectrum, the frameworks that offer state management by way of a replicated queryable persistent store help you answer much different and more complicated questions. With these frameworks you

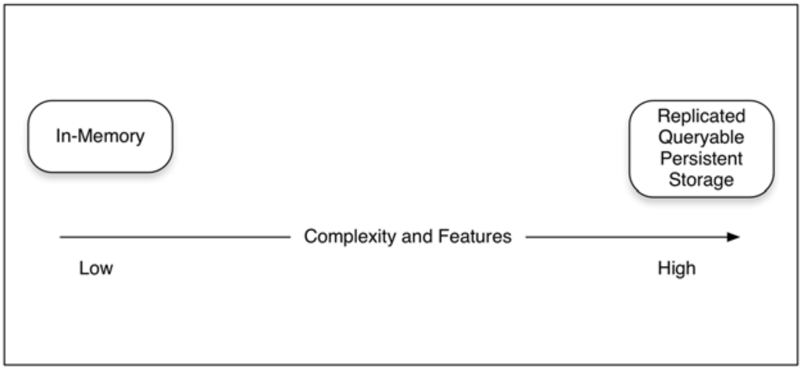


Figure 4.12 State management complexity continuum for stream processing tools

can join different streams of data together. For example, imagine you were running an ad-serving business and you wanted to track two things: the ad impression and the ad click. It's reasonable that the collection of this data would result in two streams of data, one for ad impressions and one for ad clicks. Figure 4.13 shows how these streams and your streaming job would be set up for handling this.

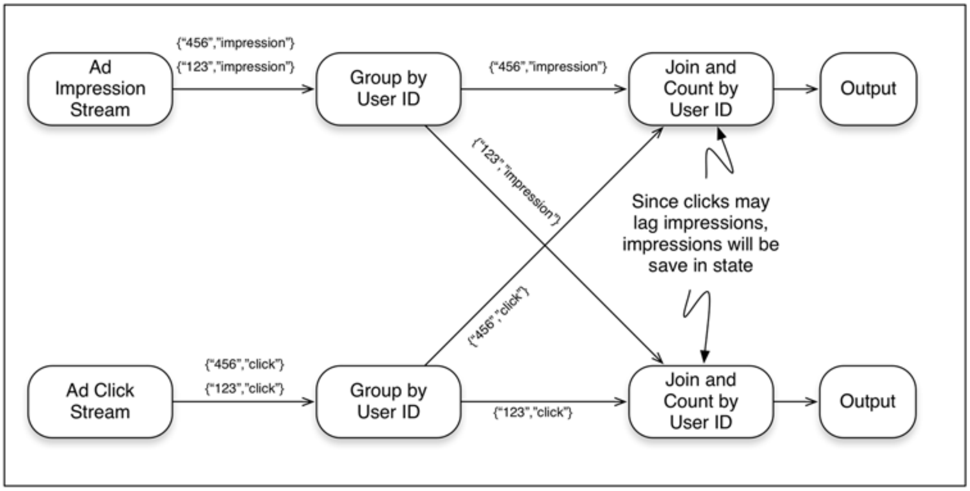


Figure 4.13 Handling ad impression and ad click streams that use stream state

In this example the ad impressions and ad clicks arrive in two separate streams; because the ad clicks will lag the ad impressions, we'll join the two streams and then count by the user ID. Because of the lag in the ad click stream, using a stream-processing framework that persists the state in a replicated queryable data store enables us to join the two streams and produce a single result. I think you'll agree that being able to join streams by leveraging the state management facilities of a stream-processing framework is quite a bit different than just making sure the current value of an aggregation is persisted. If you give some more thought to this example, I'm sure you'll come up with other ideas of how you can join more than one stream of data. It's a fascinating topic and something we'll look at in more depth in our next chapter. For now, let's continue on to the next feature you need to understand when choosing a stream-processing framework.

Fault tolerance

It's nice to think of a world where things don't fail, but in reality it's not a matter of *if* things will fail but only a matter of *when*. A stream-processing framework's ability to keep going in the face of failures is a direct result of its fault tolerance capabilities. When we consider all of the pieces involved in stream-processing, there are quite a few places where it can fail. Let's take another look at the pieces involved and use figure 4.14 to identify all of the failure points.

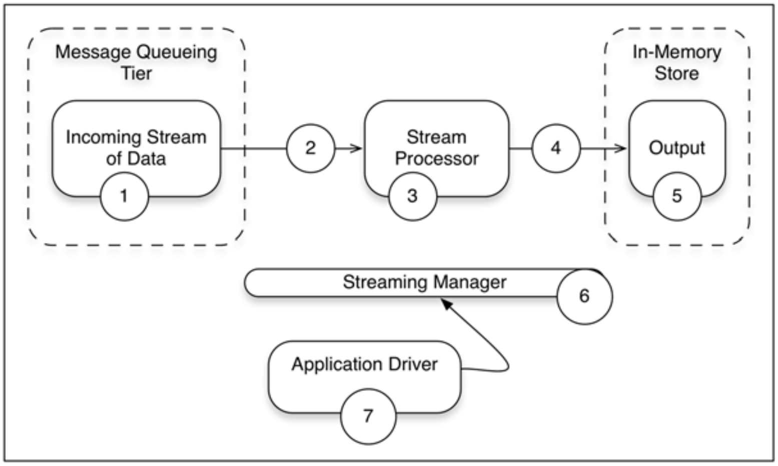


Figure 4.14 The points of failure with stream processing in the context of the streaming architecture

In figure 4.14 we've identified seven points of failure in a very simple stream-processing data flow. Go through them and make sure you understand what you'll need from a stream-processing framework in respect to fault tolerance:

1. *Incoming stream of data*—In all fairness the message queueing tier won't be under the control of the stream-processing framework, but there is the potential for the message queueing system to fail, in which case the stream-processing framework must respond gracefully and not fail if data is not available or the resource is not available.
2. *Network carrying input stream*—This is something that the stream-processing framework can't control, but it needs to handle the disruption gracefully.
3. *Stream processor*—This is where your code is running and it should be under supervision of the stream-processing framework. If something goes wrong here, perhaps your software fails or the machine it's running on fails, then the streaming manager should take steps to restart the processor or move the processing to a different machine.
4. *Connection to output destination*—The stream task manager may not be able to control the network path to the output, but it should be able to control the flow of data from the last stream processor so that it doesn't become overwhelmed by network back pressure or fail if the network or destination is unavailable.
5. *Output destination*—This would not be under the direct supervision of the stream task manager, but its failing could impact the processing of the stream and therefore it needs to be considered.
6. *Streaming manager*—If this fails, then you end up with a situation that's often referred to as "running headless." This refers to the situation where the stream processors would continue to run without being supervised by the streaming

manager. Thus if this is component fails, there's no supervisor for the data flow and the stream processors—no new ones can be started or failed ones recovered.

7. *Application driver*—This comes in two flavors. With the first, the application driver does nothing more than submit the streaming job to the streaming manager—we're not worried about this type. The second flavor is where the application driver logically contains the streaming manager and in turn is subject to the same risk as the streaming manager.

Now that you understand what you need from a stream-processing framework, let's go through how these problems are solved or could be solved. First, let's boil our problem down a bit. If we look the previous list, we can eliminate the incoming stream and output destination availability from the concerns of the streaming framework. It should go without saying that the streaming framework must not fail if there are failures with the input or output destinations. But for this discussion we won't consider those aspects to be fault tolerance related. Now if we take the list and consolidate it down to the common elements, we end up with the following:

- *Data loss*—This covers data lost on the network and also the stream processor or your job crashing and losing data that was in memory during the crash.
- *Loss of resource management*—This covers the streaming manager and your application driver in the event you have one.

If you recall our discussion of fault tolerance in chapter 3, you may remember that we discussed ways to prevent data loss. When it comes to stream-processing frameworks, all of the common techniques for dealing with failures involve some variant of replication and coordination. A common approach would be for the stream manager to replicate the state of a computation (the state of your streaming job) onto different stream processors. If there's a failure, then the streaming manager must coordinate the replicas in order to recover properly from failures. It's common for fault-tolerance techniques to be designed with a tolerance up to a predefined number of simultaneous failures, in which case you'll hear of a system being called k-fault tolerant, where k represents the number of simultaneous failures.

In general there are two common approaches a streaming system may take toward replication and coordination. Both cases assume that we have designed and thought about our streaming algorithm in an idempotent way. If you recall from before, for our streaming job to be idempotent it means that two non-faulty stream processors that receive the same input in the same order will produce the same output in the same order. If we can ensure that, then we can refer to those two stream processors and hence our streaming job as idempotent and consistent if they generate the same output in the same order. The first approach that's sometimes used by stream processing systems is known as the state-machine approach. With this approach the stream manager replicates the streaming job on independent nodes and coordinates the replicas by sending the same input in the same order to all. This approach will require k + 1 times the resources of a single replica, where k is the number of stream processors our streaming job is running on. But this allows for quick failover, resulting in very little disruption. For some applications, such as an intrusion-detection system that has low-latency requirements at all times, the extra resource cost may be justifiable.

The second approach is known as rollback recovery. In this approach, the stream processor periodically packages the state of our computation into what is called a checkpoint, and it copies the checkpoint to a different stream processor node or a nonvolatile location such as a disk. Between checkpoints, the stream processor has to keep track of the computation. Given the relative high latency of disks, once they're introduced the latency of our streaming computation will go up. It therefore may not be unreasonable for a stream-processing framework to instead decide to take the approach of copying the checkpointed state to other stream processor nodes and also maintain logs in memory. In this case if a stream processor fails, the stream manager would need to reconstruct the state from the most recent checkpoint and replay the log to recover the exact pre-failure state of the streaming job. Compared to the first approach, this approach has a lower overhead but it's more expensive in terms of time to recover when a failure does happen. This approach is useful in situations where fault tolerance is important and rare moderate latencies are acceptable.

As you investigate which stream-processing framework to use to solve your business problem, you'll find that if they offer fault-tolerance they'll all be some variant on these two common approaches. If you're interested in taking a deeper dive into either of these approaches, you may find the following articles of interest: Elnozahy, Alvisi, Wang, and Johnson's "A Survey of Rollback-Recovery Protocols in Message-Passing Systems"[1] and Schneider's "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial."[2]

4.4 Summary

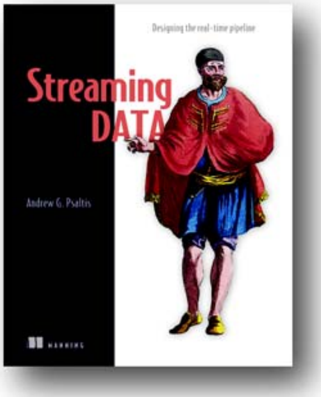
In this chapter we took a dive into the common architecture of stream-processing frameworks you'll find when surveying the landscape, and we went over the core features that you need to consider.

You learned

- About the common architecture of stream-processing frameworks
- What message delivery semantics mean for this tier
- What state is and how it can be managed
- What fault tolerance is and why you need it

[1] *ACM Computing Surveys*, 34(3):375–408 (2002).

[2] *ACM Computing Surveys*, 22(4):299–319 (1990).



Streaming Data introduces the concepts and requirements of streaming and real-time data systems. Through this book you'll develop a foundation to understand the challenges and solutions of building in-the-moment data systems before committing to specific technologies. Using copious diagrams, this book systematically builds up the blueprint for an in-the-moment system concept by concept. Although code may occasionally appear in examples, this book focuses on the big ideas of streaming and real-time data systems rather than the implementation details.

Many of the technologies discussed in the book—Spark, Storm, Kafka, Impala, RabbitMQ, etc.—are covered individually in other books. As you read, you'll get a clear picture of how these technologies work individually and together, gain insight on how to choose the correct technologies, and discover how to fuse them together to architect a robust system.

What's inside

- Understand and architect a complete system for collecting and analyzing data in real time
- Harness the "Internet of Things" by handling live data from billions of devices
- Use the specific functions of each tier of an in-the-moment system to solve real business problems
- Combine emerging technologies like Spark, Storm, Kafka, RabbitMQ, and WebSockets
- Integrating and extending the Lambda architecture into a complete system

No experience with streaming or real-time data systems required. Perfect for developers or architects, this book is also written to be accessible to technical managers and business decision makers.

Fault tolerance and recovery patterns

As applications grow in size and complexity, the impact of unanticipated error or application failure looms large. Reactive design introduces new ways of thinking about managing failure. The following chapter provides an overview of fault tolerance and recovery patterns, which, arguably, are neglected during the development of many applications. In the worst cases, this can lead to cascading failures that render an entire system unusable.

Chapter 12 from *Reactive Design Patterns*
by Roland Kuhn
with Brian Hanafee and Jamie Allen

Fault tolerance and recovery patterns

In this chapter you will learn how to incorporate the possibility of failure into the design of your application. We will demonstrate the patterns in the concrete use-case of building a resilient computation engine that allows batch job submissions and their execution on elastically provisioned hardware resources. We build upon what we learned in chapters 6 and 7, so you might want to refresh your understanding of those.

We start out with considering a single component and its failure and recovery strategies, then we build up more complex systems by hierarchical composition as well as client–server relationships. In particular we discuss the following patterns:

- The Simple Component Pattern (a.k.a. the Single Responsibility Principle)
- The Error Kernel Pattern
- The Let-It-Crash Pattern
- The Circuit Breaker Pattern

12.1 The Simple Component Pattern

"A component shall do only one thing, but do it in full."

This pattern applies wherever a system performs multiple functions or the functions it performs are so complex that they need to be broken up into different components. An example is a text editor that includes spell checking: these are two separate functions (editing can be done without spell checking, and spelling can also be checked on the finished text and does not require editing capabilities), but neither of these functions is trivial.

This pattern derives from the *Single Responsibility Principle* that was formulated by De Marco in his 1979 book *Structured analysis and system specification* (Yourdon, New York). In its abstract form it demands to “maximize cohesion and minimize coupling,” applied to object-oriented software design it is usually stated as “a class should have only one reason to change.”[1]

From the discussion of *divide et regna* in chapter 6 we know that in order to break a large problem up into a set of smaller ones, we can find help and orientation by looking at the responsibilities that the resulting components will have. Applying the process of responsibility division recursively allows us to reach any desired granularity and results in a component hierarchy that we can then implement.

12.1.1 The Problem Setting

As an example consider a service that offers computing capacity in a batch-like fashion: users submit jobs to be processed, stating a job’s resource requirements and including an executable description of the data sources and the computation that is to be performed. The service will have to watch over the resources that it manages: it will have to implement quotas for the resource consumption of its clients and schedule jobs in a fair fashion. It will also have to persistently queue the jobs that it accepts such that clients can rely upon their eventual execution.

THE TASK

Your mission is to sketch the components that make up the full batch service, noting for each one what exactly its responsibility is. Start from the top-level and work your way downwards until you reach components that are concrete and small enough so that you could task teams with implementing them.

12.1.2 Applying the Pattern

We can immediately conclude that the service implementation will be made up of two parts: one that does the coordination and that the clients communicate with, and another that will be responsible for the actual execution of the jobs; this is shown in figure 12.1. In order to make the whole service elastic, the coordinating part would tap into an external pool of resources and dynamically spin up or down executor instances. We can see that the coordination will be a rather complex task and therefore we want to break it up further.

To follow the flow of a single job request through this system, we start with the job submission interface that is offered to clients. This part of the system needs to present a network endpoint that clients can contact; it needs to implement a network protocol for

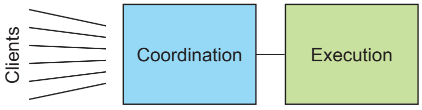


Figure 12.1 Initial component separation

[1] Uncle Bob, “Principles of OOD” (May 11, 2005; http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod)

this purpose and it will interact with the rest of the system on behalf of the clients. We could break up responsibility even finer along these lines, but for now let us consider this aspect of representing clients within the client as one responsibility; the client interface will thus be our second dedicated component.

Once a job has been accepted and the client has been informed by way of an acknowledgement message our system must ensure that eventually the job will be executed. This can only be achieved by storing the incoming jobs on some persistent medium and we might be tempted to place this storage within the client interface component. But we can already anticipate that other parts of the system will have to access these jobs, for example in order to start their execution. This means that in addition to representing the clients this component would assume responsibility for making job descriptions accessible to the rest of the system, which is in violation of the single responsibility principle.

Another temptation might be to share the responsibility for the handling of job descriptions between the interested parties—at least the client interface and the job executor as we may surmise—but that will also greatly complicate each of these components, as they will now have to coordinate their actions, running counter to the Simple Component Pattern's goal. It is much simpler to keep one responsibility within one component and avoid the communication and coordination overhead that comes with distributing it across multiple components. Besides these runtime concerns we also need to consider the implementation: sharing the responsibility means that one component needs to know about the inner workings of the other, so their development needs to be tightly coordinated as well. These are the reasons behind the second part of "do only one thing, *but do it in full.*"

This leads us to identify the storage of job descriptions as another segregated responsibility of the system and thereby as the third dedicated component. A valid interjection at this point is that the client interface component might well benefit from persisting the incoming jobs within its own responsibility, this would allow shorter response times for the job submission acknowledgement and it also makes the client interface independent from the job storage component in case of temporary unavailability. However, such a persistent queue would only have the purpose of eventually delivering accepted jobs to the storage component, who then will take responsibility of them. Therefore these notions are not in conflict with each other, we might implement both if system requirements demand it.

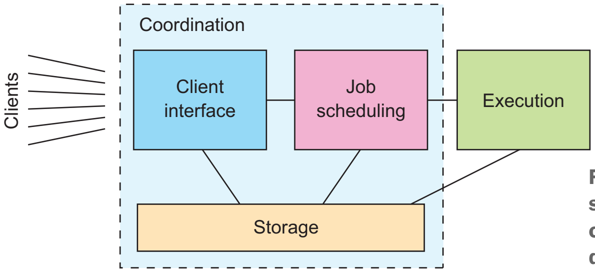


Figure 12.2 Intermediate component separation with the Coordination component being broken up in three distinct responsibilities.

Taking stock, by now we have identified the client interface, the job storage, and the job executor as three dedicated components with non-overlapping responsibilities. What remains to be done is to figure out which jobs to run in what order, we call this part "job scheduling." The current state of our system's decomposition is shown in figure 12.2; now we apply this pattern recursively until the problem is broken up into simple components.

Probably the most complex task in the whole service is to figure out the execution schedule for the accepted jobs, in particular when prioritization or fairness is to be implemented between different clients that share a common pool of resources—the corresponding allocation of computing shares are usually a matter of intense discussion between competing groups of people.[2] The scheduling algorithm will need to have access to job descriptions in order to extract scheduling requirements (maximum running time, possible expiry deadline, which kind of resources are needed, etc.), so this is another client of the job storage component.

It takes a lot of effort—both for the implementation and at runtime—to plan the execution order of those jobs that are accepted for execution, and this task is independent from deciding which jobs to accept. Therefore it will be beneficial to separate the responsibility of job validation into its own component. This also has the advantage of removing the rejected tasks before they become a burden for the scheduling algorithm. The overall responsibility of job scheduling now consists of two components, but its overall function should still be represented consistently to the rest of the system. For example, the executors need to be able to retrieve the next job to run at any given time independently of whether there is a scheduling run in progress or not. For this reason we place the external interactions in an overall scheduling component of which the validation and planning responsibilities are delegated to subcomponents. The resulting split of responsibilities for the whole system is shown in figure 12.3.

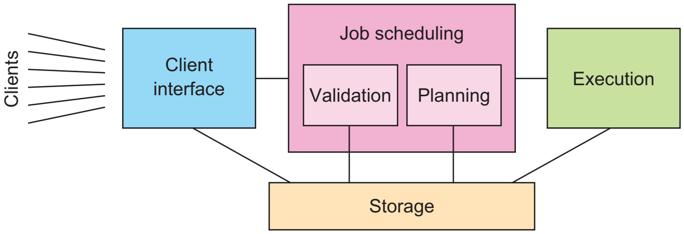


Figure 12.3 The resulting component separation

12.1.3 The Pattern Revisited

The goal of this pattern is to implement the Single Responsibility Principle and we did that by considering the responsibilities of the overall system at the highest level—client interface, storage, scheduling, execution—and separating these into dedicated

[2] The authors have some experience with such allocation between different groups of scientists competing for data analysis resources in order to extract the insights they need for academic publications.

components, keeping an eye on their anticipated communication needs. We then dived into the scheduling component and repeated the process, finding that there are sizable and non-overlapping sub-responsibilities which we split out into their own sub-components. This left the overall scheduling responsibility in a parent component because we anticipate coordination tasks that will be needed independently of the sub-components' functions.

By this process we arrived at segregated components that can be treated independently during the further development of the system. Each of these has one clearly defined purpose and each core responsibility of the system lies with exactly one component. Though the overall system and the internals of any component may be complex, the Single Responsibility Principle yields the simplest division of components to further work on—it frees us from always having to consider the whole picture when working on smaller pieces. This is its quintessential feature: it addresses the concern of system complexity.

Additionally, following the Simple Component Pattern simplifies the treatment of failures as we will exploit in the following two patterns.

12.1.4 Applicability

This is the most basic pattern to follow and it is universally applicable. Its application may lead you to a fine-grained split of your problem or to the realization that you are dealing with only one single component—the important part is that afterward you know *why* you chose your system structure as you did. It helps all later phases of the design and implementation to document and remember this, because when questions come up later of where to place certain functionality in detail you can let yourself be guided by the simple question of "what is its purpose?" The answer will directly point you toward one of the responsibilities you identified, or it will send you back to the drawing board in case you forgot to consider it.

It is important to remember that this pattern is meant to be applied in a recursive fashion, making sure that none of the identified responsibilities remain too complex or high-level. One word of warning, though: once you start dividing up components hierarchically it is easy to get carried away and go too far—the goal is simple components that have a real responsibility, not trivial components without an individual reason to exist.

12.2 The Error Kernel Pattern

"In a supervision hierarchy keep important application state or functionality near the root while delegating risky operations towards the leaves."

This pattern builds upon the Simple Component Pattern and is applicable wherever components of different failure probability and reliability requirements are combined into a larger system or application—some functions of the system must "never" go down while others are necessarily exposed to failure. Applying the Simple Component

Pattern will frequently leave you in this position, hence it pays to familiarize yourself well with the Error Kernel.

This pattern has been established in Erlang programs for decades[3] and was one of the main reasons that inspired Jonas Bonér to implement an Actor framework—Akka—on the JVM. The name "AKKA" was originally conceived as the palindrome of "Actor Kernel," referring to this core design pattern.

12.2.1 The Problem Setting

From the discussion of hierarchical failure handling in chapter 7 we know that each component of a reactive system is supervised by another component that is responsible for its lifecycle management. This implies that if the supervisor component fails then all its subordinates will be affected by the subsequent restart, resetting everything to a known good state and potentially losing intermediate updates. If the recovery of important pieces of state data is expensive then such a failure will lead to extensive service downtimes, a condition that reactive systems aim to minimize.

THE TASK

Consider each of the six components identified in the previous example as a failure domain and ask yourself which component should be responsible for reacting to its failures as well as which components will be directly affected by them. Summarize your findings by drawing the supervision hierarchy for the resulting system architecture.

12.2.2 Applying the Pattern

Since recovering from a component's failure implies the loss and subsequent recreation of its state, we shall look for opportunities to separate likely points of failure from the places where important and expensive data are kept. The same applies to pieces that provide services that shall be highly available: these should not be obstructed by frequent failure nor long recovery times. In the example we identified the following disparate responsibilities:

- Communication with clients (accepting jobs and delivering their results)
- Persistent storage of job descriptions and their status
- Overall job scheduling responsibility
- Validation of jobs against quotas or authorization requirements
- Job schedule planning
- Job execution

Each of these responsibilities benefits from being decoupled from the rest. For example the communication with clients should not be obstructed by a failure of the job scheduling logic, just as client-induced failures should not affect the currently running jobs. The same reasoning applies to the other pieces analogously. This is another reason in addition to the single responsibility principle for considering them as dedicated components as shown again in figure 12.4.

[3] The Ericsson AXD301's legendary reliability is attributed in part to this design pattern and its success popularized both the pattern and the Erlang language and runtime that were used in its implementation.

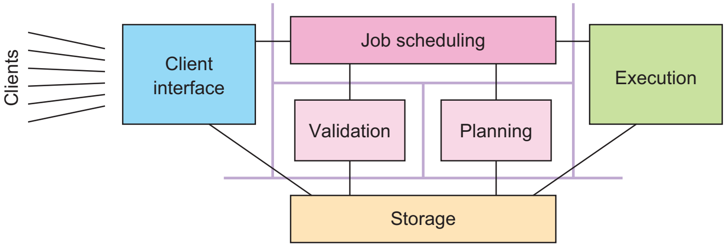


Figure 12.4 The six components drawn as separate failure domains

The next step is to consider the failure domains in the system and ask ourselves how each of them should recover and how costly that process will be. To this end we follow the path by which a job travels through the system.

Jobs enter the service through the communication component, which speaks an appropriate protocol with the clients, maintaining protocol state and validating inputs. The state that is kept is short-lived, tied to the communication sessions that are currently open with clients. When this component fails, affected clients will have to re-establish a session and possibly send commands or queries again, but our component does not need to take responsibility for these activities. In this sense it is effectively stateless—the state that it does keep is ephemeral and local. Recovery of such components is trivially done by just terminating the old and starting the new runtime instance.

Once a job has been received from a client, it will need to be persisted, a responsibility that we placed with the storage component. This component will have to allow all other components to query the list of jobs, selecting them by current status or client account and holding all necessary meta-information. Apart from caches for more efficient operation, this component does not hold any runtime state; its function is only to operate a persistent storage medium, therefore it can easily be restarted in case of failure. This assumes that the responsibility of providing persistence will be split out into a sub-component—which today is a likely approach—that we would have to consider as well. If the contents of the persistent storage become corrupted then it is a business decision whether to implement (partial) automatic resolution of these cases or leave it to the operations personnel. Automatic recovery would presumably interfere with normal operation of the storage medium and would therefore fall into the storage component's responsibility.

The next stop of a job's journey through the batch service is the scheduling component. At the top level this one has the responsibilities of applying quotas and resource request validation, as well as providing the executor component with a queue of jobs to pick up. The latter is crucial for the operation of the overall batch service: without it the executors would run idle and the system would fail to perform its core function. For this reason we place this function at the top of the scheduling component's priorities and correspondingly at the root of its sub-component hierarchy as shown in figure 12.5.

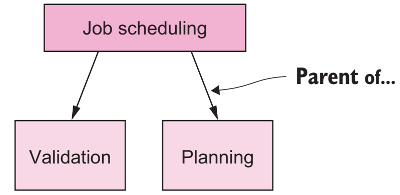


Figure 12.5 Job scheduling sub-hierarchy

While applying the Simple Component Pattern we identified two sub-responsibilities of the scheduling component. The first is to validate jobs against policy rules like per-client quotas[4] or general compatibility with the currently available resource set—it would not do to accept a job that needs 20 executor units when only 15 can be provisioned. Those jobs that pass validation from the input to the second sub-component that performs the job schedule planning for all currently outstanding and accepted jobs. Both of these responsibilities are task-based; they are started periodically and then either complete successfully or fail. Failure modes include hardware failures as well as not terminating within a reasonable time frame. In order to compartmentalize possible failures, these tasks should not directly modify the persistent state of jobs or the planned schedule but instead report back to their parent component who then takes action, be that notifying clients (via the client interface component) of jobs that failed their submission criteria or updating the internal queue of jobs to be picked next.

Whereas restarting the sub-components proved to be trivial, restarting the parent scheduling component is more complex—it will need to initiate one successful schedule planning run before it can reliably resume performing its duties. Therefore we keep the important data and the vital functionality at the root and delegate the potentially risky tasks to the leaves. Here again we note that the Error Kernel Pattern confirms and reinforces the results of the Simple Component Pattern: we frequently find that the boundaries of responsibilities and failure domains coincide and that their hierarchies match as well.

Once a job has reached the head of the scheduler's priority queue it will be picked up for execution as soon as computing resources become available. We have so far considered execution to be an atomic component, but when considering failure we come to the conclusion that we will have to divide its function: the executor needs to keep track of which job is currently running where and it will also have to monitor the health and progress of all worker nodes. The worker nodes are those components that upon receiving a job description will interpret the contained information, contact data sources, and run the analysis code that was specified by the client. Clearly the failure of each worker shall be contained to that node and not spread to other workers or the overall executor, which implies that the execution manager supervises all worker nodes as shown in figure 12.6.

[4] For example one might want to limit the maximal number of jobs queued by one client—both in order to protect the scheduling algorithm and to enforce administrative limits.

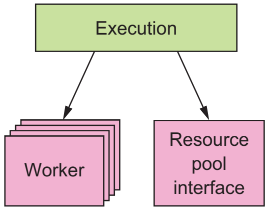


Figure 12.6 Execution component sub-hierarchy

If the system will be elastic, the executor will also make use of the external resource provision mechanism in order to create new worker nodes or shut down unused ones. The execution manager is also in the position to make the decision to enlarge or shrink the worker pool, because it naturally monitors the job throughput and it can easily be informed about the current job queue depth—another approach would be to let the scheduler decide the desired pool size. In any case it is the executor that holds the responsibility of starting, restarting, or stopping worker nodes since it is the only component that knows when it is safe or required to do so.

Analogous to the client interface component, the same reasoning applies that the communication with the external resource provision mechanism should be isolated from the other activities of the execution manager. A communication failure in that regard should not keep jobs from being assigned to already running executor instances or job completion notifications from being processed.

The execution of the job is the main purpose of the whole service, but the journey of our job through the components is not yet complete. After the assigned worker node has informed the manager about the completion status, this result needs to be sent back to the storage component in order to be persisted. If the job's nature was such that it must not be run twice, then the fact that the execution was about to start must also have been persisted in this fashion; in this case a restart of the execution manager will need to include a check of which jobs were already started but not yet completed prior to the crash, and corresponding failure results will have to be generated. In addition to persisting the final job status, the client will need to be informed about the job's result, which completes the whole process.

Now that we have illuminated the function and relationship of the different components, we recognize that we have omitted one in the earlier list of responsibilities. The service itself needs to be orchestrated, composed from its parts, supervised, and coordinated. We need one top-level component that creates the others and arranges for jobs and other messages being passed between them. In essence it is this component's function to oversee the message flow and thereby the business process of the service. This component will be top-level because of its integrating function that is needed at all times, even though it may be completely stateless by itself. The complete resulting hierarchy is shown in figure 12.7.

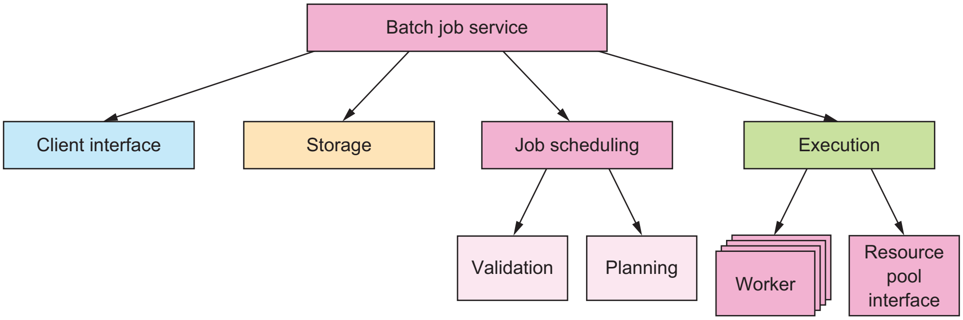


Figure 12.7 The hierarchical decomposition of the batch job service

12.2.3 The Pattern Revisited

The essence of what we did in the preceding example can be summarized in the following strategy: after applying the Simple Component Pattern, pull important state or functionality towards the top of the component hierarchy, and push activities that carry a higher risk for failure downwards towards the leaves. It is expected that the responsibility boundaries will coincide with failure domains and that narrower subresponsibilities will naturally fall towards the leaves of the hierarchy. This process may lead you to introduce new supervising components that tie together the functionality of components that are otherwise siblings in the hierarchy, or it might guide you towards a more fine-grained component structure in order to simplify failure handling or decouple and isolate critical functions to keep them out of harm's way. The quintessential function of this pattern is to integrate the operational constraints of the system into its responsibility-based problem decomposition.

12.2.4 Applicability

The Error Kernel Pattern is applicable if any of the following are true:

- Does your system consist of components that have different reliability requirements?
- Do you expect components to have significantly different failure probabilities and failure severities?
- Does the system have important functionality that it must provide as reliably as possible while also having components that are exposed to failure?
- Is there important information kept in one part of the system that is expensive to recreate while other parts are expected to fail frequently?

The Error Kernel Pattern is not applicable if:

- No hierarchical supervision scheme is used
- The system is a Simple Component
- All components are either stateless or tolerant to data loss

We will discuss the second kind of scenarios in more depth in the next chapter when presenting the Active–Active Replication Pattern.

12.3 The Let-It-Crash Pattern

"Prefer a full component restart to internal failure handling."

In chapter 7 we discussed "principled failure handling," noting that the internal recovery mechanisms of each component are limited because they are not sufficiently separated from the failing parts—everything within a component can be affected by a failure. This is especially clear for hardware failures that take down the component as a whole, but it is also true for corrupted state that is the result of some programming error that is only observable in rare circumstances. For this reason it is necessary to delegate failure handling to a supervisor instead of attempting to solve it within the component itself.

This approach is also called *crash-only software*.[5] The idea is that transient but rare failures are often very costly to diagnose and fix, making it preferable to recover a working system by rebooting parts of it. This way of hierarchical restart-based failure handling allows to greatly simplify the failure model and at the same time leads to a more robust system that even has a chance to survive failures that were entirely unforeseen.

12.3.1 The Problem Setting

We will demonstrate this design philosophy on the example of the worker nodes that perform the bulk of the work in the batch service whose component hierarchy we developed in the previous two patterns. Each of these is presumably deployed on its own hardware—virtualized or not—that it does not share with other components; ideally there is no common failure mode between different worker nodes other than a computing center outage.

The problem that we are trying to solve is that the workers' code may contain programming errors that rarely manifest, but when they do they will impede the ability to process batch jobs. Examples of this kind are very slow resource leaks that can go undetected for a long time but will eventually kill the machine; this could be open files, retained memory, background threads, etc. and the leak might not occur every time but could be caused by a rare coincidence of circumstances. Another example is a security vulnerability that allows the executed batch job to intentionally corrupt the state of the worker node in order to subvert its function and perform unauthorized actions within the service's private network—such subversion often is not completely invisible and leads to spurious failures that should better not be papered over.

[5] Both of the following articles are by George Candea and Armando Fox: "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel," USENIX HotOS VIII, 2001 (http://dslab.epfl.ch/pubs/recursive_restartability.pdf) and "Crash-Only Software," USENIX HotOS IX, 2003 (https://www.usenix.org/legacy/events/hotos03/tech/full_papers/candea/candea.pdf)

THE TASK

Your mission is to consider the components we have identified for the batch service and describe how a crash and restart would affect each of them and which implementation constraints arise from the let-it-crash philosophy.

12.3.2 Applying the Pattern

The Let-It-Crash Pattern by itself is simple: whenever a component—for example a worker node—is detected to be faulty, no attempts are made to repair the damage. Instead of doctoring with its internal state we restart it completely, releasing all its resources and starting up again from scratch. If we obtained the worker nodes by asking an infrastructure service to provision them then we can go back to the most basic state imaginable: we decommission the old worker node and provision an entirely new one. This way no corruption or accumulated failure condition can have survived in the fresh instance since we start out from a known good state again.

Applying this approach to the Client Interface nodes means that all currently active client connections will be severed for the failed node, leading to connection abort errors in the clients. Upon detecting such an abort condition the client should try to reconnect, which is our first conclusion. The second follows immediately when considering that the new connection should not be routed to the failed node: this usually means changing the load balancer configuration to remove the failed node. Then a new node needs to be brought online and added to the load balancer to restore the same processing capacity as before. With these measures we can confidently crash and restart a Client Interface node at any given point in time. We do not need to consider the internal communication because no other components depend on this one: the Client Interface only has dependencies and no dependents—the consequence of this is that any new client requests that a fresh node receives will just refer to the Storage or Scheduling components as the sources of truth; the Client Interface can be "stateless."[6]

For the Storage component, a node failure means that the stored data are invalid or lost—the consequence of either possibility is that the data cannot be relied upon any longer, so these states are fundamentally equivalent. Since the purpose of the component is to store data permanently, we will have to distribute them as per the discussion in section 2.3. We will cover data replication in the next chapter; for now it suffices to assume that there will be other storage nodes that hold copies of the data. After stopping the failed node we will therefore need to start a new one that synchronizes itself with the other replicas, taking on the share of the responsibility that the failed node had. If the new node uses the previous node's permanent storage device then recovery can be sped up by synchronizing only those updates that occurred after

[6] This word has become so widely (mis)used that it does not stand on its own any longer; the author's view is that a truly stateless service which does not contain any mutable internal state does not exist (it would not be a component with a purpose to exist) and a more meaningful interpretation is to equate statelessness with the absence of persistent mutable state.

the failure. It should be noted that failure is not the same as shutting down and starting back up again: the storage devices themselves will keep the data across the shutdown and the system will start up normally afterwards—this can even be done in many cases of infrastructure outages (like network or power failures).

In the case of the Scheduling component, a crash and restart means repopulating the internal state from persistent job storage and resuming operations. This is trivial for an aborted planning run or a failure during job validation and it can also be handled very simply for the top-level Scheduling component: we used the Error Kernel Pattern to keep this piece of software rather simple so that we could assume that a restart cycle takes a sufficiently short time to be deemed an acceptable downtime, unless specific requirements force us to use replication here as well.

The Execution component works similarly in that the worker nodes can crash and be restarted as discussed above, where the supervisor makes sure that the affected batch job is started again on another available node (or on the newly provisioned one). For the Resource Pool Interface we can tolerate short downtime while it is restarted as its services are only rarely needed, and when they are then the reaction times will be of the order of many seconds of even minutes in any case.

12.3.3 The Pattern Revisited

We have looked at each of the components in our system's supervision hierarchy and considered the consequences of a failure and subsequent restart. In some cases we encountered implementation constraints like having to update the request routing infrastructure so that the failed node is no longer considered and the replacement is taken into account once it is ready. In other cases we approached the formulation of service level agreements by saying that a short downtime may be acceptable in some cases: in a real system we would quantify this both in the failure frequency (e.g. by way of the MTBF[7]) and the extent of the outage (also called MTTR[8]).

This pattern can also be turned around so that components are "crashed" intentionally on a regular basis instead of waiting for failures to occur—this could be termed the *Pacemaker Pattern*. Deliberately inducing failures has been standard operation procedure for a long time in high-availability scenarios in order to verify that failover mechanisms are effective and perform according to their specification. The concept has been popularized in recent years by the "Chaos Monkey" employed by Netflix[9] to establish and maintain the resilience of their infrastructure. The chaotic nature of this approach manifests in single nodes being killed at random without prior selection or human consideration. The idea is that in this way failure modes are exercised that could potentially be missed in human enumeration of all possible cases.

[7] Mean Time Between Failures, see https://en.wikipedia.org/wiki/Mean_time_between_failures

[8] Mean Time To Repair, see https://en.wikipedia.org/wiki/Mean_time_to_repair

[9] At the time of writing, Netflix is the largest streaming video provider in the U.S. The Chaos Monkey is part of the SimianArmy project which is available as open-source software at https://github.com/Netflix/SimianArmy; the approach is described in detail at http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html.

On a higher level, whole data centers or geographic regions are taken offline in a more prepared manner to verify global resource reallocation—this is done on the live production system since there is no simulation environment that could practically emulate the load and client dynamics in such a large-scale application.

Another way to look at this is to consider the definition of availability: it is the fraction of time during which the system is not failed, i.e. (MTBF – MTTR) / MTBF. This can be increased either by making MTBF larger—which corresponds to less frequent but possibly extensive failures—or by making MTTR smaller. In the latter case the maximum consecutive downtime period is smaller and the system operates more smoothly, which is the goal of the Let-It-Crash pattern.

12.3.4 Implementation Considerations

While this pattern is deeply ingrained in reactive application design already, it is nevertheless documented here to take note of its important consequences on the design of components and their interaction:

- Each component must tolerate a crash and restart at any point in time, just like a power outage can happen without warning. This means that all persistent state must be managed such that the service can resume processing requests with all necessary information and ideally without having to worry about state corruption.
- Each component must be strongly encapsulated so that failures are fully contained and cannot spread. The practical realization depends on the failure model for the hierarchy level under consideration: the options range from shared-memory message passing over separate O/S processes to separate hardware in possibly different geographic regions.
- All interactions between components must tolerate peers crashing. This means ubiquitous use of timeouts and circuit breakers (described later in this chapter).
- All resources a component uses must be automatically reclaimable by performing a restart. Within an actor system this means that resources are freed by each actor upon termination or that they are leased from their parent. For an O/S process it means that the kernel will release all open file handles, network sockets, etc. when the process exits. For a virtual machine it means that the infrastructure resource manager will release all allocated memory (also persistent filesystems) and CPU resources, to be reused by a different virtual machine image.
- All requests sent to a component must be as self-describing as is practical so that processing can resume with as little recovery cost as possible after a restart.

12.3.5 Corollary: the Heartbeat Pattern

Let-it-crash describes how failures are dealt with. The other side of this coin is that failures must first be detected before they can be acted upon. In particularly catastrophic cases like hardware failures, the supervising component can only detect that something is wrong by observing the absence of expected behavior. This obviously requires that some behavior can be expected, which means that the supervisor and subordinate must communicate with each other on a regular basis. In cases where there would not otherwise be a reason for such interchange, the supervisor needs to send dummy requests whose sole purpose is to see whether the subordinate is still working properly. Due to their regular and vital nature these are called *heartbeats*. The resulting pattern's diagram is shown in figure 12.8.

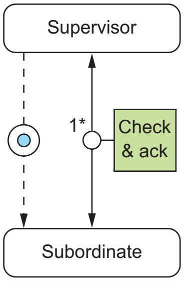


Figure 12.8 The supervisor starts the subordinate, then it performs periodic health checks by exchanging messages with it until no satisfactory answer is returned.

One caveat of using dedicated heartbeat messages is that the subordinate might have failed in a way that allows heartbeats to be processed while nothing else can be answered properly. In order to guard against such unforeseen failures health monitoring should be implemented by monitoring the service quality (failure rate, response latency, etc.) during normal operation where appropriate—sending such statistics to the supervisor on a regular basis can be used as a heartbeat signal at the same time if it is done by the subordinate itself (as opposed to the infrastructure, e.g. by monitoring the state of circuit breakers as discussed in section 12.4).

12.3.6 Corollary: The Proactive Failure Signal Pattern

Applying the Heartbeat Pattern to all failure modes results in a high level of robustness already, but there are classes of failures where patiently counting out the suspected component takes longer than necessary: the component can diagnose some failures itself. A prominent example is that all exceptions that are thrown from an actor implementation will be treated as failures—exceptions that are handled inside the actor usually pertain to error conditions resulting from the use of libraries that use exceptions for this purpose. All uncaught exceptions can be sent by the infrastructure (i.e. the actor library) to the supervisor in a message signaling the failure so that the supervisor can act upon it immediately. Wherever this is possible it should be viewed as an optimization of the supervisor's response time. The messaging pattern between supervisor and subordinate is depicted in figure 12.9 using the conventions established in appendix A.

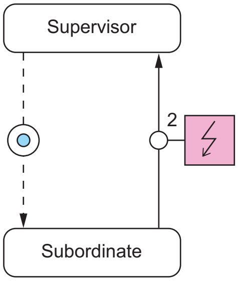


Figure 12.9 The supervisor starts the subordinate and reacts to its failure signals as they occur.

Depending on the failure model it can also be adequate to rely entirely on such measures. This is equivalent to saying that for example an actor is assumed to not have failed until it has sent a failure signal. Monitoring the health of every single actor in a system is typically forbiddingly expensive and relying on these failure signals achieves sufficient robustness at the lower levels of the component hierarchy.

It is not uncommon to combine this pattern and the Heartbeat Pattern to cover all bases. Where the infrastructure supports lifecycle monitoring—for example see the Deathwatch[10] feature of Akka actors—there is an additional way in which the supervisor can learn of the subordinate's troubles: if the subordinate has stopped itself while the supervisor still expected it to do something (or if the component is not expected to ever stop while the application is running) then the resulting termination notification can be taken as a failure signal as well. The full communication diagram for such a relationship is shown in figure 12.10.

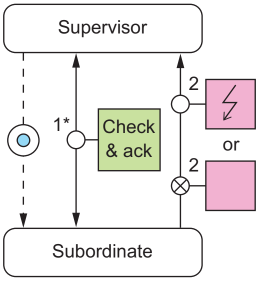


Figure 12.10 The supervisor first starts the subordinate, then it performs periodic health checks by exchanging messages with it (step 1) until either no answer is returned or a failure signal is received (step 2).

It is important to note that these patterns are not specific to Akka or the Actor Model; we use these implementations only to give concrete examples of their implementation. An application based on RxJava would for example use the Hystrix library for health monitoring, allowing components to be restarted as needed. Another example is that the deployment of components as microservices on Amazon EC2 could use the AWS API to learn of some nodes' termination and react in the same fashion as described for the DeathWatch feature here.

12.4 The Circuit Breaker Pattern

"Protect services by breaking the connection to their users during prolonged failure conditions."

In the previous sections we discussed how to segregate a system into a hierarchy of components and sub-components for the purpose of isolating responsibilities and encapsulating failure domains. This pattern describes how to safely connect different parts of the system together so that failures do not spread uncontrollably across them. Its origin lies in electrical engineering: in order to protect electrical circuits from each other and introduce decoupled failure domains, we have established the technique of breaking the connection when the transmitted power exceeds a given threshold.

Translated to a reactive application, this means that the flow of requests from one component to the next may be broken up deliberately when the recipient is overloaded or otherwise failing. This serves two purposes: firstly the recipient gets some breathing room to recover from possible load-induced failures, and secondly the sender decides that requests will fail instead of wasting time with waiting for negative replies.

[10] see http://doc.akka.io/docs/akka/2.4.1/general/supervision.html#What_Lifecycle_Monitoring_Means and http://doc.akka.io/docs/akka/2.4.1/scala/actors.html#Lifecycle_Monitoring_aka_DeathWatch

While circuit breakers have been used in electrical engineering since the 1920s, the use of this principle has been popularized in software design only recently, e.g. by Michael Nygard's book *Release It!* (Pragmatic Programmers, 2007).

12.4.1 The Problem Setting

The batch job execution facility we designed in the previous two sections will serve us yet another time. We already hinted at one place that would do well to include a circuit breaker: the service is offered to external clients who submit jobs at their own rate and schedule, and those are not naturally bounded by the capacity of the batch system.

To visualize what this means, we consider a single client that contacts the batch service to submit a single job. The client will get multiple status updates as the submitted job progresses through the system:

- Upon having received and persisted the job description
- Upon having accepted the job for execution, or upon rejecting it due to policy violations
- Upon starting execution
- Upon finishing execution

The first of these steps is a very important one: it assures the client that there will be further updates on this job because it has been admitted into the system and will at least be examined. Providing this guarantee is quite costly—it involves storing the job description in non-volatile and replicated memory—and therefore a client could easily generate more jobs per second than the system can safely ingest. In this case the client interface would overload the storage subsystem, which has knock-on effects for the job scheduling and execution components, who would now experience degraded performance when accessing job descriptions for their purposes. The system might still work in this state, but its performance characteristics would be quite different from normal operation; it would be in "overload mode."

The Task

Your mission is to sketch the use of circuit breakers between the Client Interface and the Storage component to both ensure that clients cannot willfully overload the storage, as well as to ensure that the Client Interface gives timely responses even when the Storage component is unreachable or failed.

12.4.2 Applying The Pattern

When implementing the client interface module we will have to write one piece of code that sends requests to the storage subsystem. If we make sure that all such requests take this one route then we will have an easy time reacting to the problematic scenarios outlined above. What we will have to do is keep track of the response latency for all requests that are made. When we observe that this latency rises consistently above the agreed limit then we switch into "emergency mode": instead of trying new requests we answer all subsequent ones with negative replies right away—we fabricate the negative replies on behalf of the storage subsystem, since that one cannot do even that within the allowed time window under the current conditions.

In addition we should also monitor the failure rate of replies that come back from the storage subsystem. It does not make much sense to send ever more storage requests when all of them will be answered negatively anyway. Instead of wasting the network bandwidth we should switch into "emergency mode" as well, fabricating these negative replies.

An example implementation of this scheme in Akka would look like the following:

```
private object StorageFailed extends RuntimeException

private def sendToStorage(job: Job): Future[StorageStatus] = {
  // make an asynchronous request to the storage subsystem
  val f: Future[StorageStatus] = …
  // map storage failures to Future failures to alert the breaker
  f.map {
    case StorageStatus.Failed => throw StorageFailed
    case other                => other
  }
}

private val breaker = CircuitBreaker(
    system.scheduler, // used for scheduling timeouts
    5, // number of failures in a row when it trips
    300.millis, // timeout for each service call
    30.seconds, // time before trying to close after tripping
  )

def persist(job: Job): Future[StorageStatus] =
  breaker
    .withCircuitBreaker(sendToStorage(job))
    .recover {
      case StorageFailed                   => StorageStatus.Failed
      case _: TimeoutException             => StorageStatus.Unknown
      case _: CircuitBreakerOpenException => StorageStatus.Failed
    }
```

The other client interface code will call the persist method and get back a Future representing the storage subsystem's reply, but the remote service invocation will only be performed if the circuit breaker is in the *closed* state. Negative replies (of type `StorageStatus.Failed`) or timeouts will be counted by breaker, and if it sees five failures in a row then it will transition into the *open* state in which it immediately provides a response that consists of a CircuitBreakerOpenException. After 30 seconds exactly one request will be let through again to the storage subsystem and if that comes back successfully and in time then the breaker flicks back into the *closed* state.

What we have done so far is illustrated in figure 12.11: the client interface will reply to the external clients within its allotted time, but in case of storage subsystem overload or failure, these replies will be fabricated and negative for all clients alike. Though this protects the system from attacks it is not the best we can do. Just as in electrical engineering we need to break circuits at more than one level—what we have built so far is the main circuit breaker for the whole apartment building, but we are lacking the power distribution boards that limit the damage that each individual tenant can do.

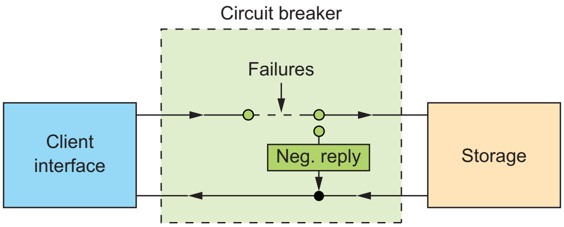


Figure 12.11 A circuit breaker between the client interface and the storage subsystem

There is one difference between these per-client circuit breakers and the main one: they react not primarily to trouble downstream, they enforce a certain maximum current to flow through them. While in computer systems this might rather be called rate limiting, this is exactly the function of circuit breakers in electrical engineering.[11] What this means is that instead of tracking the call latencies, we must remember the times of previous requests and reject new requests that violate a stated limit like "no more than 100 requests in any 2 sec interval." Writing such a facility in Scala is quite straightforward:

```
import scala.concurrent.duration.FiniteDuration
import scala.concurrent.duration.Deadline
import scala.concurrent.Future

case object RateLimitExceeded extends RuntimeException

class RateLimiter(requests: Int, period: FiniteDuration) {
  private val startTimes = {
    val onePeriodAgo = Deadline.now - period
    Array.fill(requests)(onePeriodAgo)
  }

  // the index of the next slot to be used, keeping track of when
  // the last job was enqueued in it to enforce the rate limit
  private var position = 0

  private def enqueue(time: Deadline) = {
    startTimes(position) = time
    position += 1
    if (position == requests) position = 0
  }

  def call[T](block: => Future[T]): Future[T] = {
    val now = Deadline.now // obtain current timestamp
    if ((now - startTimes(position)) < period) {
      Future.failed(RateLimitExceeded)
    } else {
      enqueue(now)
      block
    }
  }
}
```

[11] In fact the use of the term for the failure-related mode of operation may be seen as a bit of a stretch but since the name is already established for this type of use we will leave it at that.

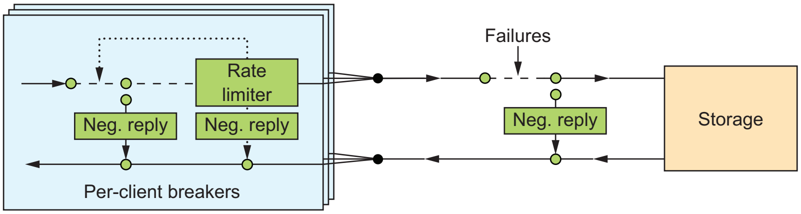


Figure 12.12 Complete circuit breaker setup between client interface and storage subsystem

Now we can combine both kinds of circuit breakers to obtain the full picture shown in figure 12.12. Clients are identified by their authentication credentials, hence we can assign one CircuitBreaker for each user independently of how many network connections they use. For each client we maintain a RateLimiter that protects the client interface from being flooded with requests. On the outgoing side towards the storage component we use one shared CircuitBreaker to guard against the remote subsystem's failures. The per-client code could look like the following:

```
private val limiter = new RateLimiter(100, 2.seconds)

// this is assumed to not be invoked concurrently as it is for a
// single client
def persistForThisClient(job: Job): Future[StorageStatus] =
  limiter
    .call(persist(job))
    .recover {
      case RateLimitExceeded => StorageStatus.Failed
    }
```

ADVANCED USAGE

It is common practice to gate a client that repeatedly violates its rate limit; this is an incentive to client code writers to properly limit the service calls on their end instead of always sending at full speed—with the previous code that would be an efficient tactic for achieving maximum throughput. In order to do that we only need to add another circuit breaker:

```
private val limiter = new RateLimiter(100, 2.seconds)
private val breaker = CircuitBreaker(system.scheduler,
                                     10, Duration.Zero, 10.seconds)

def persistForThisClient(job: Job): Future[StorageStatus] =
  breaker
    .withCircuitBreaker(limiter.call(persist(job)))
    .recover {
      case RateLimitExceeded              => StorageStatus.Failed
      case _: CircuitBreakerOpenException => StorageStatus.Gated
    }
```

In order to trip the circuit breaker the client will have to send 10 requests while being above the rate limit; assuming regular request spacing this means that the client needs to submit at least at 10% higher rate than it is allowed. In this case it will be blocked from service for the next 10 seconds and it will be informed by way of receiving a Gated status reply. The `Duration.Zero` in this code has the function of turning off the timeout tracking for individual request; this is not needed here since it will be performed by the persist call.

12.4.3 The Pattern Revisited

We have decoupled the client interface and the storage subsystem by introducing predetermined breaking points on the path from one to the other. Thus we have protected the storage from being overloaded in general (the main circuit breaker) and we have protected the client interface's function from single misbehaving clients (the rate limiting per-client circuit breakers). Being overloaded is a condition that we should strive to avoid when possible, because running at 100% capacity is in most cases less efficient than leaving a bit of headroom. The reason for this is because at full capacity, more time will be wasted competing for the available resources (CPU time, memory bandwidth, caches, IO channels) than otherwise when requests can travel through the system mostly unhindered by congestion.

The second problem that we considered here was that the client interface cannot acknowledge reception of a job description before it receives the successful reply from the storage subsystem. If this reply does not arrive within the allotted time then the response to the client will be delayed for longer than the SLA allows—the service will violate its latency bound. This means that during time periods when the storage subsystem fails to answer promptly, the client interface will have to come to its own conclusions; if it cannot ask another (non-local) component then it must locally determine the appropriate response to its own clients. We have used the circuit breaker also to protect the client interface from failures of the storage subsystem, fabricating negative responses in case no others are readily available.

We have seen that the circuit breaker that we installed for overload protection also handles the situation where the storage subsystem does not answer successfully or in time. The reaction is independent of the underlying reason. *This makes the system more resilient compared to handling every single error case separately*. And this is what is meant by bulk-heading failure domains to achieve compartmentalization and encapsulation.

12.4.4 Applicability

This pattern is applicable wherever two decoupled components communicate and where failures—foreseen or unexpected—must not travel upstream to infect and slow down other components, or where overload conditions must not travel downstream to induce failure. Decoupling has a cost in that all calls pass through another tracking step and timeouts need to be scheduled, hence it should not be applied on a too small level of granularity; it is most useful between different components in a system. This applies especially to services that are reached via network connections (like authentication providers or persistent storage) where the circuit breaker also reacts appropriately to network failures by concluding that the remote service is not currently reachable.

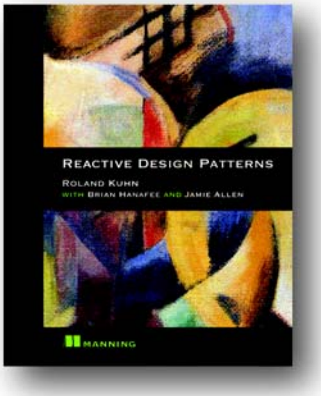
Another important use of circuit breakers is that monitoring their state reveals interesting insight into the runtime behavior and performance of a service. When circuit breakers that protect a given service are tripping, the operations personnel will usually want to be alerted in order to look into the outage.

IMPORTANT A circuit breaker is a means to fail fast: it must not be used to postpone requests and send them later. The problem with such a scheme is that when the circuit breaker closes, the deferred requests will likely overload the target system. This phenomenon is called *thundering herd* and can create feedback loops that lead to a system oscillating between being unavailable and being overloaded.

12.5 Summary

In this chapter we have covered a lot of ground on the design and implementation of resilient systems:

- We started out with describing Simple Components that obey the single responsibility principle.
- We observed the application of hierarchical failure handling in practice while implementing the Error Kernel Pattern.
- We took note of the implications of relying on component restarts to recover from failure when contemplating the Let-It-Crash Pattern.
- We learnt how to decouple components from each other using the Circuit Breaker Pattern for either side's protection.



Reactive Design Patterns is a clearly-written guide for building message-driven distributed systems that are resilient, responsive, and elastic. In it, you'll find patterns for messaging, flow control, resource management, and concurrency, along with practical issues like test-friendly designs. All patterns include concrete examples using Scala and Akka. In some cases, you'll also see examples in Java, JavaScript, and Erlang. Software engineers and architects will learn patterns that address day-to-day distributed development problems in a fault-tolerant and scalable way. Project leaders and CTOs will gain a deeper understanding of the reactive design philosophy.

The design patterns in this book were collected by the consultants and engineers of Typesafe during thousands of hours spent building enterprise-quality applications using Scala and Akka. Although many reactive patterns can be implemented using standard development tools like Java, others require the capabilities offered by a functional programming language like Scala and an Actor-based concurrency system like Akka.

What's inside

- Learn what reactive system design entails and the theory behind its principles
- Discover best practices and patterns for building reactive applications
- Build applications that can withstand hardware or software failure at any level
- Fully utilize multicore hardware using asynchronous and message-driven solutions
- Scale applications under tremendous load up and down, in and out

Readers should be familiar with a standard programming language like Java, C++, or C# and be comfortable with the basics of distributed systems. Although most of the book's examples use the Scala language, no prior experience with Scala or Akka is required.

Your first reactive web application

My book, *Reactive Web Applications*, provides the foundations for building the next generation of web applications with Scala, Akka, the Play Framework, and Reactive Streams. The following chapter gives a hands-on tour of building a simple reactive web application that streams data from Twitter asynchronously to a client's browser using WebSockets. Given the unreliable nature of computer networks, it is essential for the stream handling to be done asynchronously so as not to block computing resources such as threads and memory while the data is being received. This chapter will give you a sense of what reactive data streaming looks like at the level of a simple web application where data is being directly communicated to a user's browser.

Chapter 2 from *Reactive Web Applications*
by Manuel Bernhardt

Your first reactive web application

This chapter covers

- Creating a new Play project
- Streaming data from a remote server and broadcasting it to clients
- Dealing with failure

In the previous chapter, we talked about the key benefits of adopting a reactive approach to web application design and operation, and you saw that the Play Framework is a good technology for this. Now it's time to get your hands dirty and build a reactive web application. We'll build a simple application that connects to the Twitter API to retrieve a stream of tweets and send them to clients using WebSockets.

2.1 Creating and running a new project

An easy way to start a new Play project is to use the Lightbend Activator, which is a thin wrapper around Scala's sbt build tool that provides templates for creating new projects. The following instructions assume that you have the Activator installed on your computer. If you don't, appendix A provides detailed instructions for installing it.

Let's get started by creating a new project called "twitter-stream" in the workspace directory, using the `play-scala-v24` template:

```
~/workspace » activator new twitter-stream play-scala-2.4
```

This will start the process of creating a new project with Activator, using the template as a scaffold:

```
Fetching the latest list of templates...

OK, application "twitter-stream" is being created using the "play-scala-2.4"
➥ template.

To run "twitter-stream" from the command line, "cd twitter-stream" then:
/Users/mb/workspace/twitter-stream/activator run

To run the test for "twitter-stream" from the command line,
➥ "cd twitter-stream" then:
/Users/mb/workspace/twitter-stream/activator test

To run the Activator UI for "twitter-stream" from the command line,
➥ "cd twitter-stream" then:
/Users/mb/workspace/twitter-stream/activator ui
```

You can now run this application from the project directory:

```
~/workspace » cd twitter-stream
~/workspace/twitter-stream » activator run
```

If you point your browser to http://localhost:9000, you'll see the standard welcome page for a Play project. At any time when running a Play project, you can access the documentation at http://localhost:9000/@documentation.

> **PLAY RUNTIME MODES** Play has a number of runtime modes. In *dev mode* (triggered with the `run` command), the sources are constantly watched for changes, and the project is reloaded with any new changes for rapid development. *Production mode*, as its name indicates, is used for the production operation of a Play application. Finally, *test mode* is active when running tests, and it's useful for retrieving specific configuration settings for the test environment.

Besides running the application directly with the `activator run` command, it's possible to use an interactive console. You can stop the running application by hitting Ctrl-C and start the console simply by running `activator`:

```
~/workspace/twitter-stream » activator
```

That will start the console, as follows:

```
[info] Loading project definition from
           /Users/mb/workspace/twitter-stream/project
[info] Set current project to twitter-stream
           (in build file:/Users/mb/workspace/twitter-stream/)
[twitter-stream] $
```

Once you're in the console, you can run commands such as `run`, `clean`, `compile`, and so on. Note that this console is not Play-specific, but common to all sbt projects. Play adds a few commands to it and makes it more suited to web application development.

Table 2.1 lists some useful commands:

Table 2.1 Useful sbt console commands for working with Play

Command	Description
`run`	Runs the Play project in dev mode
`start`	Starts the Play project in production mode
`clean`	Cleans all compiled classes and generated sources
`compile`	Compiles the project
`test`	Runs the tests
`dependencies`	Shows all the library dependencies of the project, including transitive ones
`reload`	Reloads the project settings if they have been changed

When you start the application in the console with `run`, you can stop it and return to the console by pressing Ctrl-D.

> **AUTO-RELOADING** By prepending a command with ~, such as ~ `run` or ~ `compile`, you can instruct sbt to listen to changes in the source files. In this way, every time a source file is saved, the project is automatically recompiled or reloaded.

Now that you're all set to go, let's start building a simple reactive application, which, as you may have guessed from the name of the empty project we've created, has something to do with Twitter.

What we'll build is an application that will connect to one of Twitter's streaming APIs, transform the stream asynchronously, and broadcast the transformed stream to clients using WebSocket, as illustrated in figure 2.1. We'll start by building a small Twitter client to stream the data, and then build the transformation pipeline that we'll plug into a broadcasting mechanism.

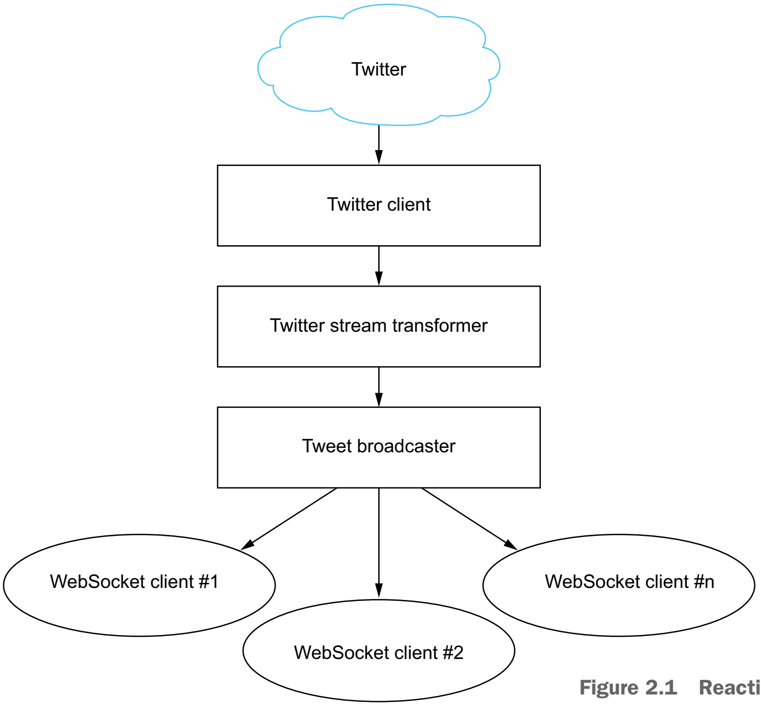


Figure 2.1 Reactive Twitter broadcaster

2.2 Connecting to Twitter's streaming API

To get started, we'll connect to the Twitter filter API.[1] At this point, we'll just focus on getting data from Twitter and displaying it on the console—we'll deal with sending it to clients connecting to our application at a later stage.

Start by opening the project in your favorite IDE. Most modern IDEs have extensions to support Play projects nowadays, and you can find resources on the topic in the Play documentation (www.playframework.com/documentation), so we won't look into setting up various flavors of IDEs here.

2.2.1 Getting the connection credentials to the Twitter API

Twitter uses the OAuth authentication mechanism to secure its API. To use the API, you need a Twitter account and OAuth consumer key and tokens. Register with Twitter (if you haven't already), and then you can go to https://apps.twitter.com where you can request access to the API for an application. This way, you'll get an API key and an API secret, which together represent the consumer key. In addition to these keys, you'll need to generate request tokens (in the Details tab of the Twitter Apps web application). At the end of this process, you should have access to four values:

[1] The Twitter API documentation can be found at https://dev.twitter.com/streaming/reference/post/statuses/filter.

- The API key
- The API secret
- An access token
- An access token secret

Once you have all the necessary keys, you'll need to add them to the application configuration in conf/application.conf. This way, you'll be able to retrieve them easily from the application later on. Add the keys at the end of the file as follows:

```
# Twitter
twitter.apiKey="<your api key>"
twitter.apiSecret="<your api secret>"
twitter.token="<your access token>"
twitter.tokenSecret="<your access token secret>"
```

2.2.2 Working around a bug with OAuth authentication

As a technical book author, I want my examples to flow and my code to look simple, beautiful, and elegant. Unfortunately the reality of software development is that bugs can be anywhere, even in projects with a very high code quality, which the Play Framework definitely is. One of those bugs has its origins in the async-http-client library that Play uses, and it plagues the 2.4.x series of the Play Framework. It can't be easily addressed without breaking binary compatibility, which is why it will likely not be fixed within the 2.4.x series.[2]

More specifically, this bug breaks the OAuth authentication mechanism when a request contains characters that need to be encoded (such as the `@` or `#` characters). As a result, we have to use a workaround in all chapters making use of the Twitter API. Open the build.sbt file at the root of the project, and add the following line:

```
libraryDependencies += "com.ning" % "async-http-client" % "1.9.29"
```

2.2.3 Streaming data from the Twitter API

The first thing we'll do now is add some functionality to the existing `Application` controller in app/controllers/Application.scala. When you open the file, it should look rather empty, like this:

```
class Application extends Controller {

  def index = Action {
    Ok(views.html.index("Your new application is ready."))
  }

}
```

The `index` method defines a means for obtaining a new `Action`. Actions are the mechanism Play uses to deal with incoming HTTP requests, and you'll learn a lot more about them in chapter 4.

[2] https://github.com/playframework/playframework/pull/4826

Start by adding a new `tweets` action to the controller.

Listing 2.1 Defining a new `tweets` action

```
import play.api.mvc._

class Application extends Controller {
  def tweets = Action {
    Ok
  }
}
```

This action won't do anything other than return a 200 Ok response when accessed. To access it, we first need to make it accessible in Play's routes. Open the conf/routes file and add a new route to the newly created action, so you get the following result.

Listing 2.2 Route to the newly created `tweets` action

```
# Routes
# This file defines all application routes
# (Higher priority routes first)
# ~~~~

# Home page
GET     /                 controllers.Application.index
GET     /tweets           controllers.Application.tweets

# Map static resources from the /public folder to the /assets URL path
GET     /assets/*file     controllers.Assets.at(path="/public", file)
```

Now when you run the application and access the /tweets file, you should get an empty page in your browser. This is great, but not very useful. Let's go one step further by retrieving the credentials from the configuration file.

Go back to the app/controllers/Application.scala controller and extend the `tweets` action as follows.

Listing 2.3 Retrieving the configuration

```
import play.api.libs.oauth.{ConsumerKey, RequestToken}
import play.api.Play.current
import scala.concurrent.Future
import play.api.libs.concurrent.Execution.Implicits._

def tweets = Action.async {
  val credentials: Option[(ConsumerKey, RequestToken)] = for {
    apiKey <- Play.configuration.getString("twitter.apiKey")
    apiSecret <- Play.configuration.getString("twitter.apiSecret")
    token <- Play.configuration.getString("twitter.token")
    tokenSecret <- Play.configuration.getString("twitter.tokenSecret")
```

Uses Action.async to return a Future of a result for the next step

Retrieves the Twitter credentials from application.conf

```
  } yield (
    ConsumerKey(apiKey, apiSecret),
    RequestToken(token, tokenSecret)
  )

  credentials.map { case (consumerKey, requestToken) =>
    Future.successful {
      Ok
    }
  } getOrElse {
    Future.successful {
      InternalServerError("Twitter credentials missing")
    }
  }
}
```

Wraps the result in a successful Future block until the next step

Returns a 500 Internal Server Error if no credentials are available

Wraps the result in a successful Future block to comply with the return type

Now that we have access to our Twitter API credentials, we'll see whether we can get anything back from Twitter. Replace the simple `Ok` result in app/controllers/Application.scala with the following bit of code to connect to Twitter.

Listing 2.4 First attempt at connecting to the Twitter API

```
// ...
import play.api.libs.ws._

def tweets = Action.async {
  credentials.map { case (consumerKey, requestToken) =>
    WS
      .url("https://stream.twitter.com/1.1/statuses/filter.json")
      .sign(OAuthCalculator(consumerKey, requestToken))
      .withQueryString("track" -> "reactive")
      .get()
      .map { response =>
        Ok(response.body)
      }
    } getOrElse {
      Future.successful {
        InternalServerError("Twitter credentials missing")
      }
    }
}

def credentials: Option[(ConsumerKey, RequestToken)] = for {
  apiKey <- Play.configuration.getString("twitter.apiKey")
  apiSecret <- Play.configuration.getString("twitter.apiSecret")
  token <- Play.configuration.getString("twitter.token")
  tokenSecret <- Play.configuration.getString("twitter.tokenSecret")
} yield (
  ConsumerKey(apiKey, apiSecret),
  RequestToken(token, tokenSecret)
)
```

The API URL

OAuth signature of the request

Specifies a query string parameter

Executes an HTTP GET request

Play's WS library lets you easily access the API by signing the request appropriately following the OAuth standard. You're currently tracking all the tweets that contain the word "reactive," and for the moment you only log the status of the response from Twitter to see if you can connect with these credentials. This may look fine at first sight, but there's a catch: if you were to execute the preceding code, you wouldn't get any useful results. The streaming API, as its name indicates, returns a (possibly infinite) stream of tweets, which means that the request would never end. The WS library would time out after a few seconds, and you'd get an exception in the console.

What you need to do, therefore, is consume the stream of data you get. Let's rewrite the previous call to `WS` and use an *iteratee* (discussed in a moment) to simply print the results you get back.

Listing 2.5 Printing out the stream of data from Twitter

```
// ...
import play.api.libs.iteratee._
import play.api.Logger

def tweets = Action.async {

  val loggingIteratee = Iteratee.foreach[Array[Byte]] { array =>
    Logger.info(array.map(_.toChar).mkString)
  }

  credentials.map { case (consumerKey, requestToken) =>
    WS
      .url("https://stream.twitter.com/1.1/statuses/filter.json")
      .sign(OAuthCalculator(consumerKey, requestToken))
      .withQueryString("track" -> "reactive")
      .get { response =>
        Logger.info("Status: " + response.status)
        loggingIteratee
      }.map { _ =>
        Ok("Stream closed")
      }
}

def credentials = ...
```

Defines a logging iteratee that consumes a stream asynchronously and logs the contents when data is available

Sends a GET request to the server and retrieves the response as a (possibly infinite) stream

Feeds the stream directly into the consuming loggingIteratee; the contents aren't loaded in memory first but are directly passed to the iteratee

Returns a 200 Ok result when the stream is entirely consumed or closed

QUICK INTRODUCTION TO ITERATEES

An iteratee is a construct that allows you to consume streams of data asynchronously; it's one of the cornerstones of the Play Framework. Iteratees are typed with input and output types: an `Iteratee[E, A]` consumes chunks of `E` to produce one or more `A`'s.

In the case of the `loggingIteratee` in listing 2.5, the input is an `Array[Byte]` (because you retrieve a raw stream of data from Twitter), and the output is of type

Unit, which means you don't produce any result other than the data logged out on the console.

The counterpart of an iteratee is an *enumerator*. Just as the iteratee is an asynchronous consumer of data, the enumerator is an asynchronous producer of data: an Enumerator[E] produces chunks of E.

Finally, there's another piece of machinery that lets you transform streaming data on the fly, called an *enumeratee*. An Enumeratee[From, To] takes chunks of type From from an enumerator and transforms them into chunks of type To.

On a conceptual level, you can think of an enumerator as being a faucet, an enumeratee as being a filter, and an iteratee as being a glass, as in figure 2.2.

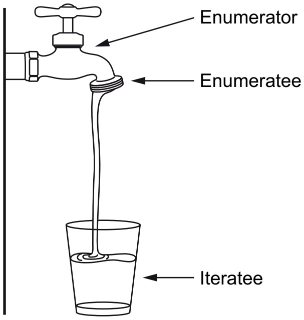


Figure 2.2 Enumerators, enumeratees, and iteratees

Let's go back to our loggingIteratee for a second, defined as follows:

```
val loggingIteratee = Iteratee.foreach[Array[Byte]] { array =>
    Logger.info(array.map(_.toChar).mkString)
}
```

The Iteratee.foreach[E] method creates a new iteratee that consumes each input it receives by performing a side-effecting action (of result type Unit). It's important to understand here that foreach isn't a method of an iteratee, but rather a method of the Iteratee library used to create a "foreach" iteratee. The Iteratee library offers many other methods for building iteratees, and we'll look at some of them later on.

At this point, you may wonder how this is any different from using other streaming mechanisms, such as java.io.InputStream and java.io.OutputStream. As mentioned earlier, iteratees let you manipulate streams of data asynchronously. In practice, this means that these streams won't hold on to a thread in the absence of new data. Instead, the thread that they use will be freed for use by other tasks, and only when there's a signal that new data is arriving will the streaming continue. In contrast, a java.io.OutputStream blocks the thread it's using until new data is available.

> **THE FUTURE OF ITERATEES IN PLAY** At the time of writing, Play is largely built on top of iteratees, enumerators, and enumeratees. *Reactive Streams* is a new standard for nonblocking stream manipulation with backward pressure on the JVM that we'll talk about in chapter 9. Although we use iteratees in this chapter and later in the book, the roadmap for the next major release of Play is to gradually replace iteratees with *Akka Streams*, which implement the Reactive Streams standard. Chapter 9 will cover this toolset as well as how to convert from iteratees to Akka Streams and vice versa.

Let's now get back to our application. Our approach to turning the Array[Byte] into a String is very crude (and, as you'll see later, problematic), but if someone were to tweet about "reactive," we'd be able to see something. If you want to check that things are going well, you can write a tweet yourself, as I just did:

```
[info] application - Status: 200
[info] application - {"created_at":"Fri Sep 19 15:08:07 +0000 2014","id
":512981466662592512,"id_str":"512981466662592512","text":"Writing the
second chapter of my book about #reactive web-applications with #PlayFr
amework. I need a tweet with \"reactive\" for an example.","source":"<a
 href=\"http:\/\/itunes.apple.com\/us\/app\/twitter\/id409789998?mt=12\
" rel=\"nofollow\">Twitter for Mac<\/a>","truncated":false,"in_reply_to
_status_id":null,"in_reply_to_status_id_str":null,"in_reply_to_user_id"
:null,"in_reply_to_user_id_str":null,"in_reply_to_screen_name":null,"us
er":{"id":12876952,"id_str":"12876952","name":"Manuel Bernhardt","scree
n_name":"elmanu","location":"Vienna" ...
```

GETTING MORE TWEETS For all the advantages of reactive applications, the keyword "reactive" is slightly less popular than more common topics on Twitter, so you may want to use another term to get faster-paced data. (One keyword that always works well, and not only on Twitter, is "cat.")

2.2.4 Asynchronously transforming the Twitter stream

Great, you just managed to connect to the Twitter streaming API and display some results! But to do something a bit more advanced with the data, you'll need to parse the JSON representation to manipulate it more easily, as shown in figure 2.3.

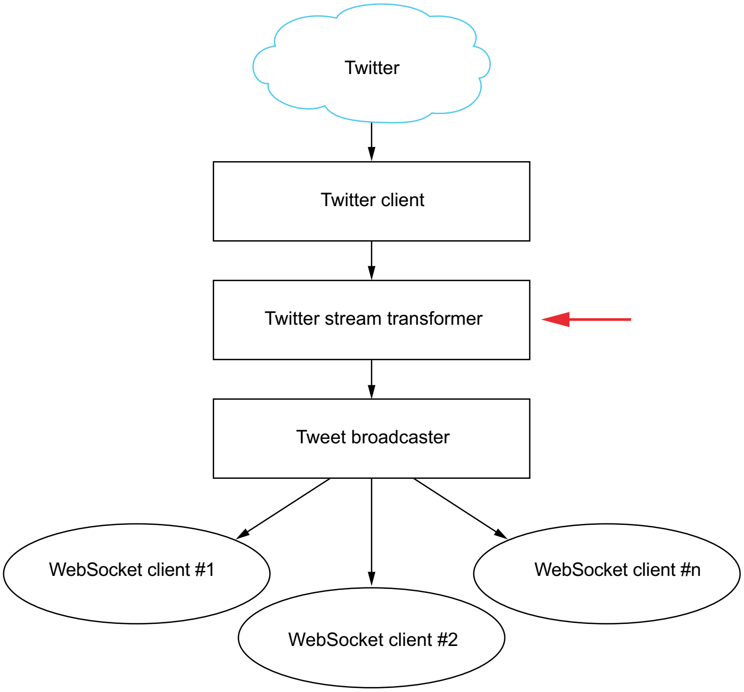


Figure 2.3 Twitter stream transformation step

Play has a built-in JSON library that can take care of parsing textual JSON files into a structured representation that can easily be manipulated. But you first need to pay a little more attention to the data you're receiving, because there are a few things that can go wrong:

- Tweets are encoded in UTF-8, so you need to decode them appropriately, taking into account variable-length encoding.
- In some cases, a tweet is split over several chunks of `Array[Byte]`, so you can't just assume that each chunk can be parsed right away.

These issues are rather complex to solve, and they may take quite some time to get right. Instead of doing it ourselves, let's use the play-extra-iteratees library. Add the following lines to the build.sbt file.

Listing 2.6 Including the play-extra-iteratees library in the project

```
resolvers += "Typesafe private" at
  "https://private-repo.typesafe.com/typesafe/maven-releases"

libraryDependencies +=
  "com.typesafe.play.extras" %% "iteratees-extras" % "1.5.0"
```

To make the changes visible to the project in the console, you need to run the `reload` command (or exit and restart, but `reload` is faster).

Armed with this library, you now have the necessary tools to handle this stream of JSON objects properly:

- `play.extras.iteratees.Encoding.decode` will decode the stream of bytes as a UTF-8 string.
- `play.extras.iteratees.JsonIteratees.jsSimpleObject` will parse a single JSON object.
- `play.api.libs.iteratee.Enumeratee.grouped` will apply the `jsSimpleObject` iteratee over and over again until the stream is finished.

We'll start with a stream of `Array[Byte]`, decode it into a stream of `CharString`, and finally parse it into JSON objects of kind `play.api.libs.JsObject` by continuously parsing one JSON object out of the incoming stream of `CharString`. `Enumeratee.grouped` continuously applies the same iteratee over and over to the stream until it's finished.

You can set up the necessary plumbing by evolving your code in app/controllers/Application.conf as follows.

Listing 2.7 Reactive plumbing for the data from Twitter

```
// ...
import play.api.libs.json._
import play.extras.iteratees._
```

```
def tweets = Action.async {
    credentials.map { case (consumerKey, requestToken) =>
      val (iteratee, enumerator) = Concurrent.joined[Array[Byte]]

      val jsonStream: Enumerator[JsObject] =
        enumerator &>
        Encoding.decode() &>
        Enumeratee.grouped(JsonIteratees.jsSimpleObject)

      val loggingIteratee = Iteratee.foreach[JsObject] { value =>
        Logger.info(value.toString)
      }

      jsonStream run loggingIteratee

      WS
        .url("https://stream.twitter.com/1.1/statuses/filter.json")
        .sign(OAuthCalculator(consumerKey, requestToken))
        .withQueryString("track" -> "reactive")
        .get { response =>
          Logger.info("Status: " + response.status)
          iteratee
        }.map { _ =>
          Ok("Stream closed")
        }
    }
}

def credentials = ...
```

Sets up a joined iteratee and enumerator

Defines the stream transformation pipeline; each stage of the pipe is connected using the &> operation

Plugs the transformed JSON stream into the logging iteratee to print out its results to the console

Provides the iteratee as the entry point of the data streamed through the HTTP connection. The stream consumed by the iteratee will be passed on to the enumerator, which itself is the data source of the jsonStream. All the data streaming takes place in a nonblocking fashion.

The first thing you have to do in this setup is get an enumerator to work with. Iteratees are used to consume streams, whereas enumeratees produce them, and you need a producing pipe so you can add adapters to it. The `Concurrent.joined` method provides you with a connected pair of iteratee and enumerator: whatever data is consumed by the iteratee will be immediately available to the enumerator.

Next, you want to turn the raw `Array[Byte]` into a proper stream of parsed `JsObject` objects. To this end, start off with your enumerator and pipe the results to two transforming enumeratees:

- `Encoding.decode()` to turn the `Array[Byte]` into a UTF-8 representation of type `CharString` (an optimized version of a String proper for stream manipulation, and part of the play-extra-iteratees library)
- `Enumeratee.grouped(JsonIteratees.jsSimpleObject)` to have the stream consumed over and over again by the `JsonIteratees.jsSimpleObject` iteratee

The `jsSimpleObject` iteratee ignores whitespace and line breaks, which is convenient in this case because the tweets coming from Twitter are separated by a line break.

Set up a logging iteratee to print out the parsed JSON object stream, and connect it to the transformation pipeline you just set up using the `run` method of the enumerator.

This method tells the enumerator to start feeding data to the iteratee as soon as some is available.

Finally, by providing the `iteratee` reference to the `get()` method of the WS library, you effectively put the whole mechanism into motion.

If you run this example, you'll now get a stream of tweets printed out, ready to be manipulated for further use.

> **FASTER JSON PARSING** Although the play-extra-iteratees library is very convenient, the JSON tooling it offers isn't optimized for speed; it serves as more of a showcase of what can be done with iteratees. If I wanted to build a pipeline for production use, or where performance matters a lot more than low memory consumption, I'd probably create my own enumeratee and make use of a fast JSON parsing library such as Jackson.

2.3 Streaming tweets to clients using a WebSocket

Now that we have streaming data being sent by Twitter, let's make it available to users of our web application using WebSockets. Figure 2.4 provides an overview of what we want to achieve.

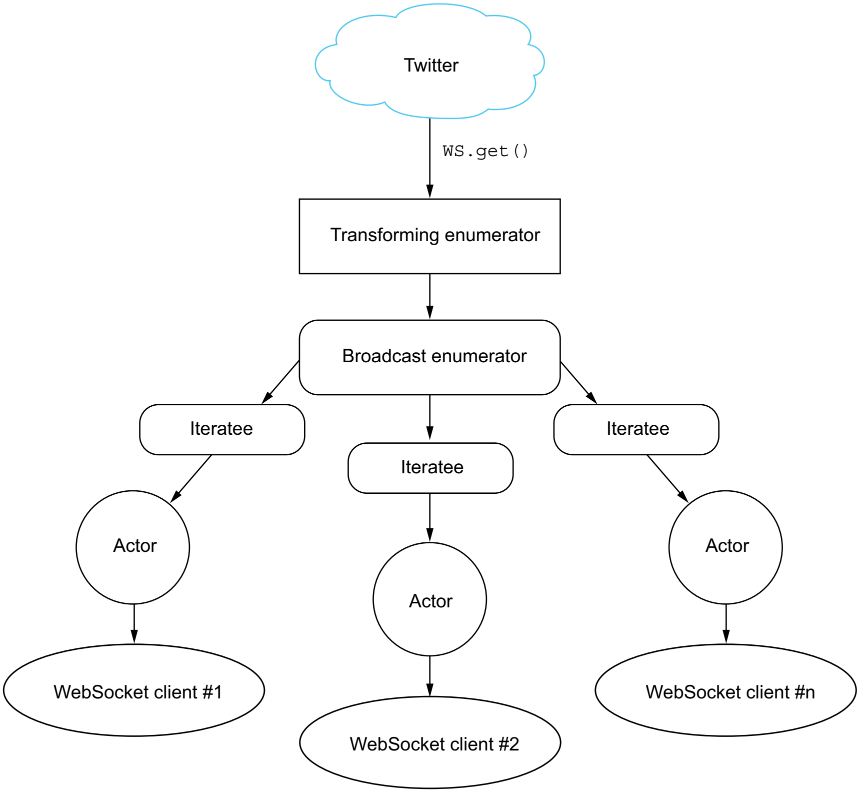


Figure 2.4 Reactive pipeline from Twitter to the client's browser

We want to connect once to Twitter and broadcast the stream we receive to the user's browser using the WebSocket protocol. We'll use an actor to establish the WebSocket connection for each client and connect it to the same broadcasted stream.

We'll proceed in two steps: first, we'll move the logic responsible for retrieving the stream from Twitter to an Akka actor, and then we'll set up a WebSocket connection that makes use of this actor.

2.3.1 Creating an actor

An actor is a lightweight object that's capable of sending and receiving messages. Each actor has a mailbox that keeps messages until they can be dealt with, in the order of reception. Actors can communicate with each other by sending messages. In most cases, messages are sent asynchronously, which means that an actor doesn't wait for a reply to its message, but will instead eventually receive a message with the answer to its question or request. This is all you need to know about actors for now—we'll talk about them more thoroughly in chapter 6.

To see an actor in action, start by creating a new file in the `actors` package, app/actors/TwitterStreamer.scala, with the following content.

Listing 2.8 Setting up a new actor

```
package actors

import akka.actor.{Actor, ActorRef, Props}
import play.api.Logger
import play.api.libs.json.Json

class TwitterStreamer(out: ActorRef) extends Actor {
  def receive = {
    case "subscribe" =>
      Logger.info("Received subscription from a client")
      out ! Json.obj("text" -> "Hello, world!")
  }
}

object TwitterStreamer {
  def props(out: ActorRef) = Props(new TwitterStreamer(out))
}
```

The receive method handles messages sent to this actor.

Handles the case of receiving a "subscribe" message

Sends out a simple Hello World message as a JSON object

Helper method that initializes a new Props object

You want to use your actor to represent a WebSocket connection with a client, managed by Play. You need to be able to receive messages, but also to send them, so you pass the `out` *actor reference* in the constructor of the actor. Play will take care of initializing the actor using the `akka.actor.Props` object, which you provide in the `props` method of the companion object `TwitterStreamer`. It will do so every time a new WebSocket connection is requested by a client.

An actor can send and receive messages of any kind using the `receive` method, which is a so-called *partial function* that uses Scala's pattern matching to figure out which case statement will deal with the incoming message. In this example, you're only concerned with messages of type String that have the value "subscribe" (other messages will be ignored).

When you receive a subscription, you first log it on the console, and then (for the moment) send back the JSON object { "message": "Hello, world!" }. The exclamation mark (!) is an alias for the tell method, which means that you "fire and forget" a message without waiting for a reply or a delivery confirmation.

> **SCALA TIP: PARTIAL FUNCTIONS** In Scala, a partial function p(x) is a function that's defined only for some values of x. An actor's receive method won't be able to handle every type of message, which is why this kind of function is a good fit for this method. Partial functions are often implemented using pattern matching with case statements, wherein the value is matched against several case definitions (like a switch expression in Java).

2.3.2 Setting up the WebSocket connection and interacting with it

To make use of your freshly baked actor, you need to create a WebSocket endpoint on the server side and a view on the client side that will initialize a WebSocket connection.

SERVER-SIDE ENDPOINT

We'll start by rewriting the tweets method of the Application controller (you may want to keep the existing method as a backup somewhere, because we'll reuse most of its parts later on). You'll notice that we're not creating a Play Action this time, because actions only deal with the HTTP protocol, and WebSockets are a different kind of protocol. Play makes initializing WebSockets really easy.

Listing 2.9 Setting up the WebSocket endpoint in app/controllers/Application.scala

```
// ...
import actors.TwitterStreamer

// ...

def tweets = WebSocket.acceptWithActor[String, JsValue] {
  request => out => TwitterStreamer.props(out)
}
```

That's it! You don't need to adjust the route in the routes file either, because you're essentially reusing the existing mapping to the /tweets route.

The acceptWithActor[In, Out] method lets you create a WebSocket endpoint using an actor. You specify the type of the input and output data (in this case, you want to send strings from the client and receive JSON objects) and provide the Props of the actor, given the out actor reference that you're using to communicate with the client.

> **SIGNATURE OF THE ACCEPTWITHACTOR METHOD** The acceptWithActor method has a slightly uncommon signature of type f: RequestHeader =>ActorRef => Props. This is a function that, given a RequestHeader, returns another function that, given an ActorRef, returns a Props object. This construct allows you to access the HTTP request header information for purposes such as performing security checks before establishing the WebSocket connection.

CLIENT-SIDE VIEW

We'll now create a client-side view that will establish the WebSocket connection using JavaScript. Instead of creating a new view template, we'll simply reuse the existing view template, app/views/index.scala.html, as follows.

Listing 2.10 Client-side connection to the WebSocket using JavaScript

```
@(message: String)(implicit request: RequestHeader)

@main(message) {
    <div id="tweets"></div>
    <script type="text/javascript">
        var url = "@routes.Application.tweets().webSocketURL()";
        var tweetSocket = new WebSocket(url);

        tweetSocket.onmessage = function (event) {
            console.log(event);
            var data = JSON.parse(event.data);
            var tweet = document.createElement("p");
            var text = document.createTextNode(data.text);
            tweet.appendChild(text);
            document.getElementById("tweets" ).appendChild(tweet);
        };

        tweetSocket.onopen = function() {
            tweetSocket.send("subscribe");
        };
    </script>
}
```

Initializes the WebSocket connection using a URL generated by Play

The container in which the tweets will be displayed

The handler called when a message is received

The handler called when the connection is opened

Sends a subscription request to the server

You start by opening a WebSocket connection to the tweets handler. The URL is obtained using Play's built-in reverse routing and resolves to ws://localhost:9000/tweets. Then you add two handlers: one for handling new messages that you receive, and one for handling the new WebSocket connection once a connection with the server is established.

> **USING URLS IN VIEWS** It's also possible to make use of reverse routing natively in JavaScript. We'll look into that in chapter 10.

When a new connection is established, you immediately send a subscribe message using the send method, which is matched in the receive method of the TwitterStreamer on the server side.

Upon receiving a message on the client side, you append it to the page as a new paragraph tag. To do this, you need to parse the `event.data` field, as it's the string representation of the JSON object. You can then access the `text` field, in which the tweet's text is stored.

There's one change you need to make for your project to compile, which is to pass the `RequestHeader` to the view from the controller. In app/controllers/Application .scala, replace the `index` method with the following code.

Listing 2.11 Declaring the implicit `RequestHeader` to make it available in the view

```
def index = Action { implicit request =>
  Ok(views.html.index("Tweets"))
}
```

You need to take this step because in the index.scala.html view you've declared two parameter lists: a first one taking a message, and a second implicit one that expects a `RequestHeader`. In order for the `RequestHeader` to be available in the implicit scope, you need to prepend it with the `implicit` keyword.

Upon running this page, you should see "Hello, world!" displayed. If you look at the developer console of your browser, you should also see the details of the event that was received.

Scala tip: implicit parameters

Implicit parameters are a language feature of Scala that allows you to omit one or more arguments when calling a method. Implicit parameters are declared in the last parameter list of a function. For example, the index.scala.html template will be compiled to a Scala function that has a signature close to the following:

```
def indexTemplate(message: String)(implicit request: RequestHeader)
```

When the Scala compiler tries to compile this method, it will look for a value of the correct type in the implicit scope. This scope is defined by prepending the `implicit` keyword when declaring anonymous functions, as here with `Action`:

```
def index = Action { implicit request: RequestHeader =>
  // request is now available in the implicit scope
}
```

You don't need to explicitly declare the type of `request`; the Scala compiler is smart enough to do so on its own and to infer the type.

2.3.3 Sending tweets to the WebSocket

Play will create one new `TwitterStreamer` actor for each WebSocket connection, so it makes sense to only connect to Twitter once, and to broadcast our stream to all connections. To this end, we'll set up a special kind of broadcasting enumerator and provide a method to the actor to make use of this broadcast channel.

We first need an initialization mechanism to establish the connection to Twitter. To keep things simple, let's set up a new method in the companion object of the `TwitterStreamer` actor in app/actors/TwitterStreamer.scala.

Listing 2.12 Initializing the Twitter feed

```
object TwitterStreamer {
 def props(out: ActorRef) = Props(new TwitterStreamer(out))

 private var broadcastEnumerator: Option[Enumerator[JsObject]] = None

 def connect(): Unit = {
  credentials.map { case (consumerKey, requestToken) =>
   val (iteratee, enumerator) = Concurrent.joined[Array[Byte]]

   val jsonStream: Enumerator[JsObject] = enumerator &>
    Encoding.decode() &>
    Enumeratee.grouped(JsonIteratees.jsSimpleObject)

   val (be, _) = Concurrent.broadcast(jsonStream)
   broadcastEnumerator = Some(be)

   val url = "https://stream.twitter.com/1.1/statuses/filter.json"
   WS
    .url(url)
    .sign(OAuthCalculator(consumerKey, requestToken))
    .withQueryString("track" -> "reactive")
    .get { response =>
     Logger.info("Status: " + response.status)
     iteratee
    }.map { _ =>
     Logger.info("Twitter stream closed")
    }

  } getOrElse {
   Logger.error("Twitter credentials missing")
  }
 }
}
```

Initializes an empty variable to hold the broadcast enumerator

Sets up a joined set of iteratee and enumerator

Sets up the stream transformation pipeline, taking data from the joined enumerator

Initializes the broadcast enumerator using the transformed stream as a source

Consumes the stream from Twitter with the joined iteratee, which will pass it on to the joined enumerator

With the help of the broadcasting enumerator, the stream is now available to more than just one client.

A WORD ON THE CONNECT METHOD Instead of encapsulating the `connect()` method in the `TwitterStreamer` companion object, it would be better practice to establish the connection in a related actor. The methods exposed in the `TwitterStreamer` connection are publicly available, and misuse of them may seriously impact your ability to correctly display streams. To keep this example short, we'll use the companion object; we'll look at a better way of handling this case in chapter 6.

You can now create a subscribe method that lets your actors subscribe their WebSocket clients to the stream. Append it to the TwitterStreamer object as follows.

Listing 2.13 Subscribing actors to the Twitter feed

```
object TwitterStreamer {

  // ...

  def subscribe(out: ActorRef): Unit = {
    if (broadcastEnumerator.isEmpty) {
      connect()
    }
    val twitterClient = Iteratee.foreach[JsObject] { t => out ! t }
    broadcastEnumerator.foreach { enumerator =>
      enumerator run twitterClient
    }
}
```

In the subscribe method, you first check if you have an initialized broadcastEnumerator at your disposal, and if not, establish a connection. Then you create a twitterClient iteratee, which sends each JSON object to the browser using the actor reference.

Finally, you can make use of this method in your actor when a client subscribes.

Listing 2.14 TwitterStreamer actor subscribing to the Twitter stream

```
class TwitterStreamer(out: ActorRef) extends Actor {
  def receive = {
    case "subscribe" =>
      Logger.info("Received subscription from a client")
      TwitterStreamer.subscribe(out)
  }
}
```

When running the chain, you should now see tweets appearing on the screen, one after another. You can open multiple browsers or tabs to see more client connections being established.

This setup is very resource-friendly given that you only make use of asynchronous and lightweight components that don't block threads: when no data is sent from Twitter, you don't unnecessarily block threads waiting or polling. Instead, each time new data comes in, the parsing and subsequent communication with clients happen asynchronously.

PROPER DISCONNECTION HANDLING One thing we haven't done here is properly handle client disconnections. When you close the browser tab or otherwise disconnect the client, your twitterClient iteratee will continue trying to send new messages to the out actor reference, but Play will have closed the WebSocket connection and stopped the actor, which means that messages will be sent to the void. You can observe this behavior by seeing Akka complain in the log about "dead letters" (actors sending messages to no-longer-existing endpoints). To properly handle this situation, you'd need to keep track of subscribers and check if each actor is still in the list of subscribers prior to sending each message. You can find an example of how this is done in the source code for this chapter, available on GitHub.

2.4 Making the application resilient and scaling out

We've built a pretty slick and resource-efficient application to stream tweets from our server to many clients. But to meet the failure-resilience criterion of a reactive web application, we need to do a bit more work: we need a good mechanism to detect and deal with failure, and we need to be able to scale out to respond to higher demand.

2.4.1 Making the client resilient

To be completely resilient, our application would need to be able to deal with a multitude of failure scenarios, ranging from Twitter becoming unavailable to our server crashing. We'll look into a first level of failure handling on the client side here, in order to alleviate the pain inflicted on our users if the stream of tweets were to be interrupted. We'll cover the topic of responsive clients in depth in chapter 8.

If the connection with the server is lost, we should alert the user and attempt to reconnect. This can be achieved by rewriting the `<script>` section of our index.scala.html view, as follows.

Listing 2.15 Resilient version of the JavaScript

```
function appendTweet(text) {
    var tweet = document.createElement("p");
    var message = document.createTextNode(text);
    tweet.appendChild(message);
    document.getElementById("tweets").appendChild(tweet);
}

function connect(attempt) {
    var connectionAttempt = attempt;
    var url = "@routes.Application.tweets().webSocketURL()";
    var tweetSocket = new WebSocket(url);
    tweetSocket.onmessage = function (event) {
        console.log(event);
        var data = JSON.parse(event.data);
        appendTweet(data.text);
    };
    tweetSocket.onopen = function() {
        connectionAttempt = 1;
        tweetSocket.send("subscribe");
    };
    tweetSocket.onclose = function() {
        if (connectionAttempt <= 3) {
```

Encapsulates the WebSocket connection logic in a reusable function

Attempts up to three connection retries

The onclose handler, called when the WebSocket connection is closed

```
      appendTweet("WARNING: Lost server connection,
  attempting to reconnect. Attempt number " + connectionAttempt);
      setTimeout(function() {
        connect(connectionAttempt + 1);
      }, 5000);
    } else {
      alert("The connection with the server was lost.");
    }
  };
}

connect(1);
```

Executes the wrapped function call after a delay of 5000 milliseconds

Attempts reconnection and increments the number of retries

In case of failure, alerts the user with a more prominent alert

Initiates the first connection attempt

To avoid repeating the same code twice, you start by moving the logic for displaying a new message into the `appendTweet` method and the logic for establishing a new WebSocket connection into the `connect` method. The latter now takes as its argument the connection attempt count, so you know when to give up trying and can then inform the user about the progress.

The `onclose` handler of the WebSocket API is invoked whenever the connection with the server is lost (or can't be established). This is where you plug in your failure-handling mechanism: when the connection is lost, you inform the user in an unobtrusive manner (by appending a warning message to the existing tweet stream) and then attempt to reconnect after a waiting period of five seconds. If you haven't succeeded after three reconnection attempts, you alert the user in a more direct fashion (in this example, by using a native browser alert). If you succeed at reconnecting, you reset the connection attempt count to 1.

> **FURTHER COPING MECHANISMS** It's not uncommon for a web application to lose connection with the server. One popular mechanism implemented in many clients, such as Gmail, is to wait for increasing amounts of time between two reconnection attempts (first a few seconds, then a minute, and so on), while still informing the user and also giving them a means to reestablish the connection manually by clicking a link or button. This disconnection scenario is quite frequent with mobile devices and laptops, so it's good for an application to have an automated reconnection mechanism in place to optimize the user experience.

> **SERVER-SIDE FAILURE HANDLING** So far we've only handled failures on the client side; we haven't looked into mechanisms to deal with failure handling on the server side. This is not, unfortunately, because there are no failures on the server side, but rather because this topic is too big to cover in this chapter's example application. Don't worry, though. We'll revisit this aspect of the application in detail in chapters 5 and 6.

2.4.2 Scaling out

We now have a pretty slick and resource-efficient application that can stream tweets to many clients. But what if we were to build a fairly popular application, and we wanted

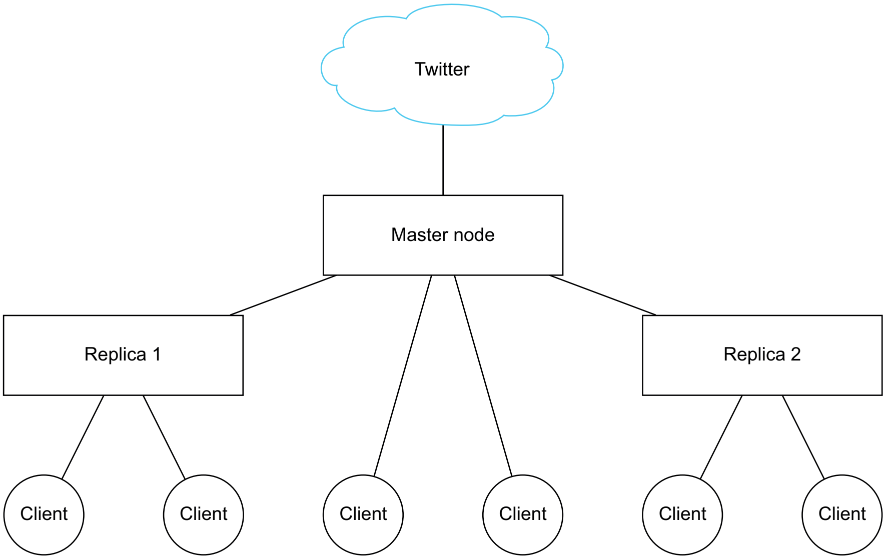


Figure 2.5 Scaling out using replica nodes

to handle more connections than a single node could manage? One mechanism we'll consider is *replica nodes* that could replicate our initial connection, as shown in figure 2.5.

Let's say we wanted to reuse the same connection to Twitter (because Twitter doesn't let us reuse the same credentials many times, and we don't want to create a new user and get new API credentials for each node). We already have a mechanism in place that lets clients view the stream using WebSockets, and we also have a mechanism to broadcast an incoming Twitter stream to WebSocket clients. The only thing we need in order to have working replica nodes that connect to a master node is a means to configure them and get them to connect to our master node instead of Twitter.

To achieve this, we'll set up a new subscription mechanism that allows other nodes to consume data from the initial stream (the one coming from Twitter). We'll set up a new controller action to stream out the content and make the necessary modifications to run the application in *replica* mode.

First, you need to set up a means for the `controller` method to subscribe to the stream.

Listing 2.16 Subscribing other nodes to the broadcast Twitter feed

```
def subscribeNode: Enumerator[JsObject] = {
  if (broadcastEnumerator.isEmpty) {
    connect()
  }
  broadcastEnumerator.getOrElse {
    Enumerator.empty[JsObject]
  }
}
```

This method, like the existing subscribe method, first makes sure that the connection to Twitter is initialized, and then simply returns the broadcasting enumeratee. You can now use the enumeratee in a controller method in your Application controller.

Listing 2.17 Streaming the replicated Twitter feed in the controller

```
class Application extends Controller {

  // ...

  def replicateFeed = Action { implicit request =>
    Ok.feed(TwitterStreamer.subscribeNode)
  }
}
```

The feed method simply feeds the stream provided by the enumerator as an HTTP request.

You now need to provide a new route for this action in conf/routes:

```
GET     /replicatedFeed         controllers.Application.replicateFeed
```

If you now visit http://localhost:9000/replicatedFeed, you'll see the stream of JSON documents displayed with continuous additions to the page.

You now have almost everything in place to set up a replica node. The last thing you need to do is connect to the master node instead of the original Twitter API. You can do this very easily by replacing the URL used in a replica node with the master node's URL. In a production setup, you'd use the application configuration for this. To keep things simple for this example, we'll use a JVM property that can easily be passed along. Add the following logic in the connect() method of the TwitterStreamer companion object, replacing the existing URL declaration:

```
val maybeMasterNodeUrl = Option(System.getProperty("masterNodeUrl"))
val url = maybeMasterNodeUrl.getOrElse {
  "https://stream.twitter.com/1.1/statuses/filter.json"
}
```

Now, start a new terminal window and start another Activator console (don't close the existing running application):

```
activator -DmasterNodeUrl=http://localhost:9000/replicatedFeed
```

Then run the application on another port:

```
[twitter-stream] $ run 9001
```

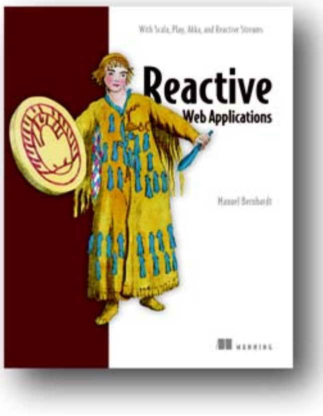
Upon visiting http://localhost:9001, you'll see the stream from the other node. You can start more of those nodes on different ports to check if the replication works as expected. Given how the setup works, you can also chain more replicating nodes by passing the URL of a replicating node as masterNodeUrl to another node.

FAILURE HANDLING IN A REPLICATED SETUP Although scaling out makes your application capable of handling a higher demand in terms of connections, it also makes failure handling quite a bit more complicated. Given the limitation of only one node being able to connect to Twitter, you're in a situation where there is a single point of failure—if this node were to go down, you'd be in trouble. In a real system, you'd seek to avoid having a single point of failure, and instead have a number of master nodes. You'd also need to devise a mechanism to cope with the loss of a master server.

2.5 Summary

In this chapter, we built a reactive web application using Play and Akka. We used a few key techniques for reactive applications:

- Using asynchronous actions for handling incoming HTTP requests
- Streaming and transforming tweets asynchronously using iteratees, enumeratees, and enumerators
- Establishing WebSocket connections using an Akka actor and connecting it to the stream
- Dealing with failure on the client side
- Scaling out using a simple replication model



The emerging reactive model is ideal for high-performance web applications that need to manage the unpredictably-bursty behavior of the web, along with the potential instability of running on networks you don't fully control. By using application components that communicate asynchronously as they react to user and system events, reactive applications are more scalable, responsive, and fault-tolerant than standard monolithic applications. For web developers working in Java or Scala, the Play Framework makes it easy to implement reactive applications without taking on the overhead of building everything from scratch.

Reactive Web Applications teaches web developers how to benefit from the reactive application architecture and presents hands-on examples using the Play Framework. This book quickly introduces Play as a framework to handle the plumbing of your application. As you move through the book, you'll alternate between chapters that introduce reactive ideas like asynchronous programming, managing distributed state, and fault tolerance and examples that show you how to build such applications using Play. Readers new to Play will be able to learn from the ground up. If you're already using Play, you'll get a deeper look at how to implement reactive web applications effectively.

What's inside

- Introduces the reactive application architecture
- Learn the basics of Play Framework
- Examples in Scala
- Asynchronous programming with Futures, Actors, and Reactive Streams

This book assumes only that you're comfortable programming with a higher-level language such as Java or C#, and can read Scala code. No prior experience with Play Framework or reactive applications is required.

Getting smart with MLlib

The Apache Spark data-processing platform handles many of the complex aspects of processing large data streams in applications. The following chapter introduces MLlib, a library that makes it possible to apply machine learning techniques on incoming data. The particular technique introduced in this chapter is linear regression.

Machine learning has regained popularity since the recent evolution in hardware made it available to a much broader set of developers and organizations, enabling them to make more accurate decisions based on the data at hand. After reading this chapter, you'll have a sense of how machine learning works in practice and how Apache Spark supports it.

Chapter 7 from *Spark in Action*
by Petar Zečević and Marko Bonaći

Getting smart with MLlib

This chapter covers

- Machine learning basics
- Performing linear algebra in Spark
- Scaling and normalizing features
- Training and applying a linear regression model
- Evaluating the model's performance
- Using regularization
- Optimizing linear regression

Machine learning is a scientific discipline studying the use and development of algorithms that make computers accomplish complicated tasks without explicitly programming them. That is, the algorithms eventually learn how they can solve a given task. These algorithms include various methods and techniques from statistics, probability, and information theory.

Today, machine learning is ubiquitous. Some examples include online stores that offer you similar items that other users have viewed or bought, email clients that automatically move emails to spam, advances in autonomous driving recently developed by several car manufacturers, and speech and video recognition. It's also becoming a big part of online business: finding hidden relationships in user habits and actions (and learning from them) can bring critical added value to existing products and services.

But with the advent of companies handling huge amounts of data (known as Big Data), more scalable machine learning packages are needed. Spark provides distributed and scalable implementations of various machine learning algorithms and makes it possible to handle those continuously growing data sets[1].

Spark offers distributed implementations of the most important and most often used machine learning algorithms, and new implementations are constantly being added. Spark‘s distributed nature helps you apply machine learning algorithms on very large data sets with adequate speed. Spark, as a unifying platform, lets you perform most of the machine learning tasks (such as data collection, preparation, analysis, model training, and evaluation), all inside the same system and using the same API.

In this chapter, you'll use *linear regression* to predict Boston housing prices. *Regression analysis* is a statistical process of modeling relationships between variables, and *linear regression,* as a special type of the regression analysis, assumes those relationships to be linear. It's historically one of the most widely used and simplest regression methods in statistics.

While using linear regression to predict housing prices, you'll learn about linear regression itself: how to prepare the data, train the model, use the model to make predictions, and evaluate the model's performance and optimize it. We'll begin with a short introduction to machine learning and a primer on using linear regression in Spark.

First, a disclaimer. Machine learning is such a vast subject that it's impossible to fully cover it here. To learn more about machine learning in general, check out *Real-World Machine Learning*, by Henrik Brink and Joseph W. Richards (Manning, 2016 [est.]), and *Machine Learning in Action*, by Peter Harrington (Manning, 2012). A sea of other resources can be found online; Stanford's "*Machine Learning*" course by Andrew Ng is an excellent starting point.[2]

7.1 Introduction to machine learning

Let's start with an example of using machine learning in real life. Let's say you're running a website that lets people sell their cars online. And let's say you'd like your system to automatically propose to your sellers reasonable starting prices when they post their ads. You know that regression analysis can be used for that purpose by taking data of previous sales, analyzing characteristics of the cars and their selling prices, and modeling the relation between them. But you don't have enough ads in your database, so you decide to get car prices from publicly available sources. You find a lot of interesting car sale records online, but most of the data is available in CSV files, and large parts of it are PDF and Word documents (containing car sale offers).

[1] Spark is not the only framework that provides a distributed machine learning package. There are other frameworks, such as Graphlab, Flink, and Parameter Server.

[2] See http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=MachineLearning

You first parse PDFs and Word documents to identify and match similar fields (manufacturer, model, make, and so on). You know that a regression analysis model can't handle string values of various fields ("*automatic*" and "*manual*," for example), so you come up with a way to convert these values to numeric ones. Then you notice that important fields are missing from some of the records ("*year manufactured*," for example) and you decide to remove those records from your data set.

When you finally have the data cleaned up and stored somewhere, you start examining various fields—how they are correlated and what their distributions look like (this is important for better understanding the hidden dependencies of the data). Then you decide which regression analysis model to use.

Let's say you choose linear regression, because based on the correlations you calculated, you assume the main relations to be linear.

Before building a model, you normalize and scale the data (more on how and why this is done soon) and split it into training and validation data sets. You finally train your model using the training data (you use the historical data to set weights of the model to predict the future data where the price is not known; we will explain this later), and you get a usable linear regression model. But when you test it on your validation data set, the results are horrible. You change some of the parameters used for

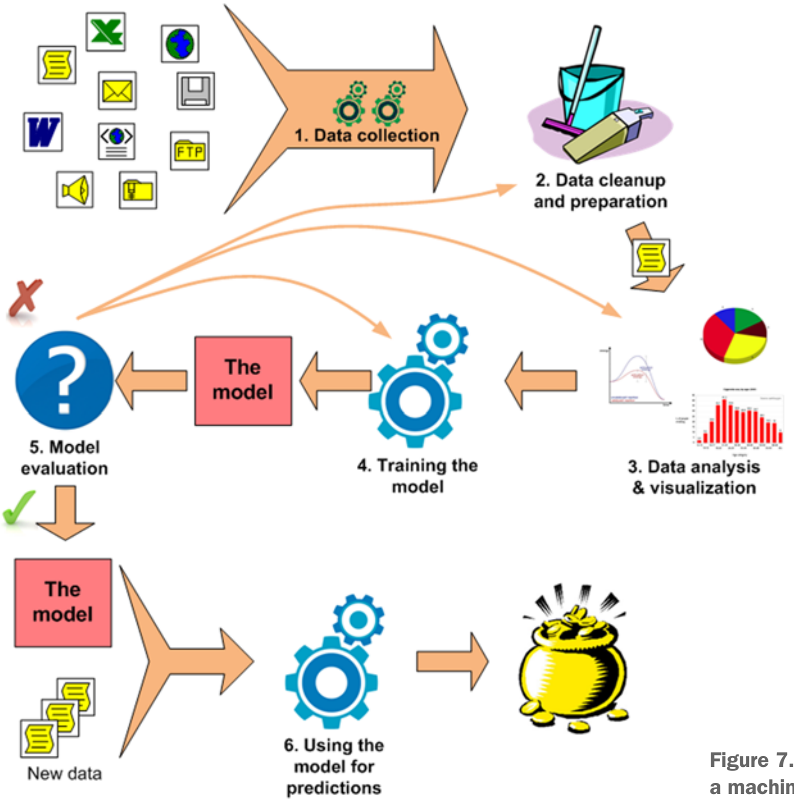

FTP
1. Data collection
2. Data cleanup and preparation
3. Data analysis & visualization
4. Training the model
The model
5. Model evaluation
The model
New data
6. Using the model for predictions

Figure 7.1 Typical steps in a machine learning project

training the model, test it again, and repeat the process until you get a model with acceptable performance.

Finally, you incorporate the model in your web application and start getting emails from your clients wondering how you‘re doing that (or clients complaining about bad predictions).

What this example illustrates is that a machine learning project consists of multiple steps. Although typical steps are shown on figure 7.1, the whole process can usually be broken down into the following:

1. *Data collection*—First, the data needs to be gathered from various sources. The sources can be log files, database records, signals coming from sensors, and so on. Spark can help in loading the data from relational databases, CSV files, remote services, and distributed file systems like HDFS, or from real-time sources using Spark Streaming.
2. *Data cleaning and preparation*—Data is not always available in a structured format appropriate for machine learning (text, images, sounds, binary data, and so forth), so you need to devise and carry out a method of transforming this unstructured data into numerical features. Additionally, you need to handle missing data and the different forms in which the same values can be entered (for example, “VW” and “Volkswagen” are the same carmaker). Often data also needs to be scaled so that all dimensions are of comparable ranges.
3. *Data analysis and feature extraction*—Next you analyze the data, examine its correlations, and visualize them (using various tools) if necessary. (The number of dimensions might be reduced in this step if some of them don’t bring any extra information, for example, if they are redundant.) You then choose the appropriate machine learning algorithm (or set of algorithms) and split the data into training and validation subsets—this is important because you would like to see how the model behaves on the data not seen during the training phase. Or you decide on a different cross-validation method, where you continuously split the data set into different training and validation data sets and average the results over the rounds.
4. *Training the model*—Here you train a model by running an algorithm that *learns* a set of algorithm-specific parameters from the input data.
5. *Model evaluation*—You then put the model to use on the validation data set and evaluate its performance according to some criteria. At this point, you may decide you need more input data or that you need to change the way features were extracted. You may also change the feature space or switch to a different model. In any of these cases, you would go back to step 1 or step 2.
6. *Using the model*—Finally, you deploy the built model to the production environment of your website.

The mechanics of using an API (Spark or some other machine learning library) to train and test the models is only the last and the shortest part of the process. Equally important are collection, preparation, and analysis of data, where knowledge about the problem domain is needed. Therefore, these two machine learning chapters are mostly about steps 4 and 5 described previously.

7.1.1 Definition of machine learning

Machine learning is one of the largest research areas within *artificial intelligence*, a scientific field that studies algorithms for simulating intelligence. Ron Kohavi and Foster Provost in their article "*Glossary of Terms*" describe machine learning in these words:

> *Machine learning is a scientific discipline that explores the construction and study of algorithms that can learn from and make predictions on data.*[3]

This is in contrast to traditional programming methods where an algorithm that needs to do what it's explicitly programmed for (like parsing an XML file with a certain structure) is explicitly programmed into it. Such traditional methods can't be easily expanded to cover similar tasks, like parsing XML files with a similar structure. As another example, making a speech-recognition program that recognizes different accents and voices would be impossible by explicitly programming it, because the sheer number of variations in the way a single word can be pronounced would necessitate that many versions of the program.

Instead of incorporating the explicit knowledge about the problem area in the program itself, machine learning relies on methods from the fields of statistics, probability, and information theory to discover and use the knowledge inherent in data and then change the behavior of a program accordingly in order to be able to solve the initial task (such as recognizing speech).

7.1.2 Classification of machine learning algorithms

The most basic classification of machine learning algorithms divides them into two classes called *supervised* and *unsupervised learners*. A data set for supervised learning is pre-labeled (information about the expected prediction output is provided with the data), whereas one for unsupervised learning contains no labels and the algorithm needs to determine them itself.

Supervised learning is used for many practical machine learning problems today, such as spam detection, speech and handwriting recognition, computer vision, and more. A spam-detection algorithm, for example, is trained on examples of emails manually marked as spam or not spam (labeled data) and learns how to classify future emails.

Unsupervised learning is also a powerful tool that is is widely used. Among other purposes, it's used for discovering structure within data—for example, groups of similar items known as *clusters*)—anomaly detection, image segmentation, and so on.

[3] Ron Kohavi; Foster Provost (1998). "Glossary of terms", *Editorial for the Special Issue on Applications of Machine Learning and the Knowledge Discovery Process, vol.* 30, 271–274.

Classification to supervised and unsupervised algorithms

In *supervised* learning, an algorithm is given a set of known inputs and matching outputs, and it has to find a function that can be used to transform the given inputs to the true outputs even in the case of input data not seen during the training phase. The same function can then be used to predict outputs of any future input. The typical supervised learning tasks are regression and classification.

Regression attempts to predict the *values* of continuous output variables based on a set of input variables. *Classification* aims to classify sets of inputs into two or more *classes (discrete output variables)*. Both regression and classification models are trained based on a set of inputs with known outputs—where known outputs are the output variables values or classes, which are supervised problems.

In the case of unsupervised learning, the output is not known in advance, and the algorithm has to find some structure in the data without additional information provided. A typical unsupervised learning task is clustering. With clustering, the goal of the algorithm is to discover dense regions, called clusters, in the input data by analyzing similarities between the input examples. There are no known classes used as a reference.

For an example of differences between supervised and unsupervised learning, consider figure 7.2. It shows the often used Iris flower data set[4] created in 1936. The data set contains widths and lengths of petals and sepals[5] of 150 flowers of three iris flower species: *Iris setosa*, *Iris versicolor,* and *Iris virginica* (50 flowers of each species). For the sake of simplicity, only sepal length and width are given in figure 7.2. That way we can plot the data set in two dimensions.

Sepal length and sepal width are *features* (or *dimensions*) of input, and the flower species is the output (or *target variable, a label*). We would like our algorithm to find a mapping function that correctly maps sepal length and sepal width to flower species for existing and future examples.

> **NOTE** For historical reasons, and because of many possible application areas, a single concept in machine learning can have several different names. Inputs are also called *examples*, *points*, *data samples*, *observations,* or *instances*. In Spark, training examples for supervised learning are called *labeled points*. Features (sepal length and sepal width in the Iris data set, for example) are also called *dimensions, attributes*, *variables,* or *independent variables*.

On the graph on the lefthand side of figure 7.2, flower species corresponding to each input are marked with dots, circles, and x marks, which means that the flower species are known in advance. We call this the *training set* because it can be used to *train* (or *fit*) the parameters of the machine learning model to determine the mapping function. You would then test the accuracy of your trained model using a *test set* containing

[4] Iris flower data set, Wikipedia (http://en.wikipedia.org/wiki/Iris_flower_data_set)

[5] A *sepal* is a part of flower that supports its petals and protects the flower in bud.

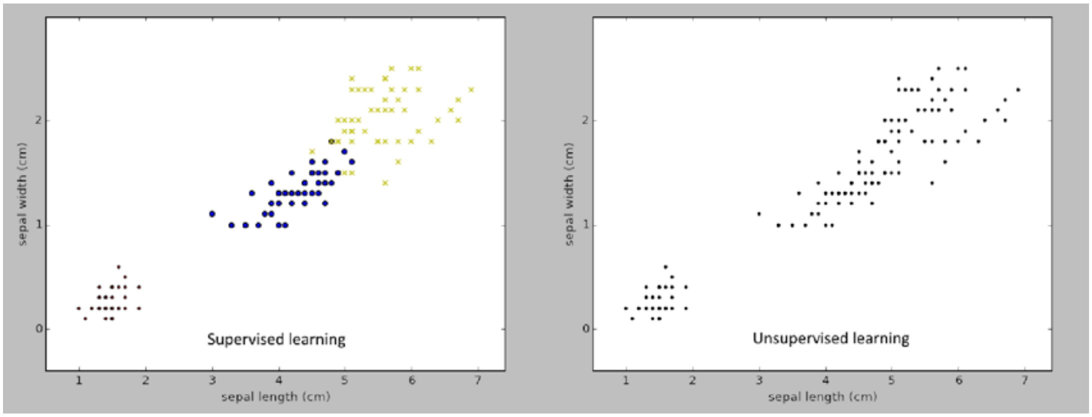


Figure 7.2 Supervised and unsupervised learning in the Iris flower data set. Data set for supervised learning is pre-labeled, whereas the one for unsupervised learning contains no labels because the algorithm needs to determine them itself.

a different set of labeled examples. If satisfied with its performance, you would then let it predict labels for some real data.

The graph on the right in figure 7.2, on the other hand, shows clustering (a form of unsupervised learning), which requires the algorithm to find the mapping function *and* the categories. As you can see, all the examples are marked with the same symbol (a dot), and the algorithm needs to find the "most likely" grouping system for the given examples.

In the graph showing clustering, there is obviously a clear separation between the group of examples in the lower left-hand corner of the graph and the rest of the examples, but the separation between the other two categories is not that clear. You can probably already guess that an unsupervised learning algorithm will be less successful in correctly separating this data set into the three flower categories—that's because the supervised learning algorithm has much more data to learn from.

ALGORITHM CLASSIFICATION BASED ON THE TYPE OF TARGET VARIABLE

Besides classifying machine learning algorithms as supervised and unsupervised, they can also be classified according to the type of the target variable into *classification* and *regression* algorithms.

The Iris data set mentioned in the last section is an example of a *classification* problem because target variables are *categorical* (or *qualitative*), which means that they can take on a limited number of values (*discrete* values). In classification algorithms, the target variable is also called a *label, class,* or *category* and the algorithm itself is called a *classifier, recognizer,* or *categorizer.*

In the case of *regression* algorithms, the target variable is *continuous* or *quantitative,* (a real number).

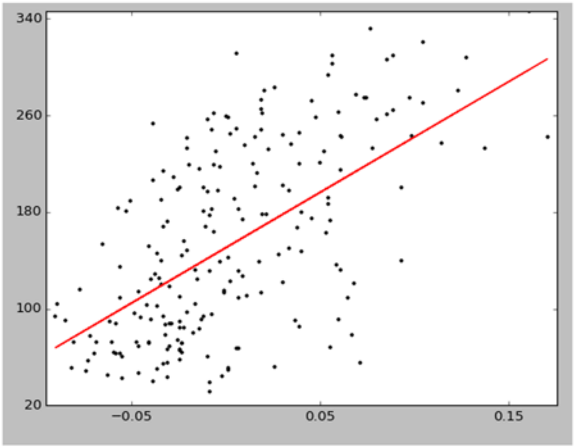


Figure 7.3 Example of a simple linear regression problem

Both regression and classification are plainly supervised learning algorithms because the estimation function is fitted according to *a priori* known values. Figure 7.3 shows an example of linear regression with only one feature, shown on axis x. The output value is shown on axis y.

The goal of regression is to find, based on a set of examples, a mathematical function that will be as close an approximation of the relationship between features and the target variable as possible. The regression in figure 7.3 is a simple linear regression because there is only one independent variable (making it simple), and the hypothesis function is modeled as a linear function (a straight line). If there were two variables, you could plot the estimation function in 3D space as a plane. When there are more features, the function becomes a *hyperplane*.

7.1.3 Machine learning with Spark

All the advantages of Spark extend to machine learning, too. The most important aspect of Spark is its distributed nature. It enables you to train and apply machine learning algorithms on very large data sets with adequate speed.

The second advantage is Spark‘s unifying nature; it offers a platform for performing most of tasks. You can collect, prepare, and analyze the data and train, evaluate, and use the model—all inside the same system and using the same API.

Spark offers distributed implementations of the most popular machine learning algorithms, and new ones are constantly added. Spark’s primary machine learning API is called MLlib. It based on the MLbase project in Berkeley, California. Since its inclusion in Spark version 0.8, MLlib has been expanded and developed by the open source community.

Spark version 1.2 introduced a new machine learning API called Spark ML. The idea behind Spark ML is to provide a generalized API that can be used for training and tuning different algorithms in the same way. It also provides *pipelines,* sequences of machine learning–related processing steps that are collected together and handled as a unit.

The new Spark ML API is developed in parallel with the "old" Spark MLlib API. Spark MLlib will continue to be supported and expanded.

SUPPORTING LIBRARIES

Spark relies on several low-level libraries for performing optimized linear algebra operations. These are Breeze and jblas for Scala and Java and NumPy for Python. Refer to the official documentation (https://spark.apache.org/docs/latest/mllib-guide.html#dependencies) for how to configure these. We will use the default Spark build, but that decision shouldn't influence functional aspects described in this chapter.

7.2 Linear algebra in Spark

Linear algebra is a branch of mathematics focusing on vector spaces and linear operations and mappings between them expressed mainly by matrices. Linear algebra is essential for understanding the math behind most machine learning algorithms, so if you don't know much about vectors and matrices, you should have a peek at appendix C for a primer on linear algebra.

Matrices and vectors in Spark can be manipulated locally (inside the driver or executor processes) or in a distributed manner. Implementations of distributed matrices in Spark enable you to perform linear algebra operations on huge amounts of data, spanning numerous machines. For local linear algebra operations, Spark uses the very fast Breeze and jblas libraries (and NumPy in Python), and it has its own implementations of distributed ones.

Sparse and dense vectors and matrices

Spark supports *sparse* and *dense* vectors and matrices. A vector or matrix is sparse if it contains mostly zeros. It's more efficient to represent such data with pairs of indices and values at those indices. A sparse vector or matrix can be likened to a map (or dictionary in Python).

On the other hand, a dense vector or matrix contains all of the data—values at all index positions not storing the indices, similar to an array or a list.

7.2.1 Local vector and matrix implementations

Local vector and matrix implementations in Spark are located in the package `org.apache.spark.mllib.linalg`. We'll examine Spark's linear algebra API with a set of examples you can run in your Spark shell. To start the shell in local mode, use the command `spark-shell --master local[*]`. We'll assume you're running Spark in the *spark-in-action* virtual machine.

GENERATING LOCAL VECTORS

Local vectors in Spark are implemented with two classes: DenseVector and SparseVector, implementing a common interface called Vector, making sure that both implementations support exactly the same set of operations. The main class for creating vectors is the Vectors class and its dense and sparse methods. The dense method has two versions: it can take all elements as inline arguments or it can take an array of elements. For the sparse method, you need to specify a vector size, an array with indices, and an array with values. The following three vectors (dv1, dv2, and sv) contain the same elements and, hence, represent the same mathematical vectors:

```
import org.apache.spark.mllib.linalg.{Vectors,Vector}
val dv1 = Vectors.dense(5.0,6.0,7.0,8.0)
val dv2 = Vectors.dense(Array(5.0,6.0,7.0,8.0))
val sv = Vectors.sparse(4, Array(0,1,2,3), Array(5.0,6.0,7.0,8.0))
```

> **NOTE** Make sure to always use sorted indices for constructing your sparse vectors (the second argument of the sparse method). Otherwise, you may get unexpected results.

You can access a specific element in the vector by its index like this:

```
scala> dv2(2)
res0: Double = 7.0
```

You can get the size of the vector with the size method:

```
scala> dv1.size
res1: Int = 4
```

To get all elements of a vector as an array, you can use the toArray method or:

```
scala> dv2.toArray
res2: Array[Double] = Array(5.0, 6.0, 7.0, 8.0)
```

LINEAR ALGEBRA OPERATIONS ON LOCAL VECTORS

Linear algebra operations on local vectors can be done using the Breeze library, which Spark uses internally for the same purposes. toBreeze functions exist in Spark vector and matrix local implementations, but they are declared as private. The Spark community has decided not to allow end users access to this library, because they do not want to depend on a third-party library. But you will most likely need a library for handling local vectors and matrices.

An alternative would be to create your own function for converting Spark vectors to Breeze classes, which is not really that hard to do. We propose the following solution:

```
import org.apache.spark.mllib.linalg.{DenseVector, SparseVector, Vector}
import breeze.linalg.{DenseVector => BDV,SparseVector => BSV,Vector => BV}
def toBreezeV(v:Vector):BV[Double] = v match {
    case dv:DenseVector => new BDV(dv.values)
    case sv:SparseVector => new BSV(sv.indices, sv.values, sv.size)
}
```

Now you can use this function (toBreezeV) and the Breeze library to add vectors and calculate their dot products. For example:

```
scala> toBreezeV(dv1) + toBreezeV(dv2)
res3: breeze.linalg.Vector[Double] = DenseVector(10.0, 12.0, 14.0, 16.0)
scala> toBreezeV(dv1).dot(toBreezeV(dv2))
res4: Double = 174.0
```

The Breeze library offers more linear algebra operations, and we invite you to examine its rich set of functionalities. You should note that the names of Breeze classes conflict with the names of Spark classes, so be careful when using both in your code. One solution is to change the class names during import, as in the preceding toBreezeV function example.

Generating local dense matrices

Similar to the Vectors class, the Matrices class also has the methods dense and sparse for creating matrices. The dense method expects number of rows, number of columns, and an array with the data (elements of type Double). The data should be specified column-wise, which means that the elements of the array will be used sequentially to populate columns. For example, to create the following matrix as a DenseMatrix:

$$\boldsymbol{M} = \begin{bmatrix} 5 & 0 & 1 \\ 0 & 3 & 4 \end{bmatrix}$$

Use a code snippet similar to this one:

```
scala> import org.apache.spark.mllib.linalg.{DenseMatrix, SparseMatrix,
     Matrix, Matrices}
scala> import breeze.linalg.{DenseMatrix => BDM,CSCMatrix => BSM,Matrix =>
     BM}
scala> val dm = Matrices.dense(2,3,Array(5.0,0.0,0.0,3.0,1.0,4.0))
dm: org.apache.spark.mllib.linalg.Matrix =
5.0  0.0  1.0
0.0  3.0  4.0
```

A Matrices object provides some "shortcut" methods for quickly creating identity and diagonal matrices and matrices with all zeros and ones. The eye(n) method[6] creates a dense identity matrix of size *n x n*. The method speye is the equivalent for creating a sparse identity matrix. Methods ones(m, n) and zeros(m, n) create dense matrices with all ones or zeros of size *m x n*. The diag method takes a Vector and creates a diagonal matrix (its elements are all zeros, except the ones on its main diagonal) with elements from the input Vector placed on its diagonal. Its dimensions will be equal to the size of the input Vector.

Additionally, you can generate a DenseMatrix filled with *random numbers in a range from* 0 to 1 using the rand and randn methods of the Matrices object. The first

[6] Identity matrices are usually denoted with the letter I, pronounced the same as the word "eye," hence the pun in the method name.

method generates numbers according to a uniform distribution, and the second one according to Gaussian distribution. (Gaussian distribution, also known as *normal* distribution, has that familiar bell-shaped curve you have most probably seen.) Both distributions take the number of rows, the number of columns, and an initialized `java.util.Random` object as arguments. The `sprand` and `sprandn` methods are equivalent methods for generating `SparseMatrix` objects.

> **NOTE** These methods (`eye`, `rand`, `randn`, `zeros`, `ones`, and `diag`) are not available in Python.

GENERATING LOCAL SPARSE MATRICES

Generating sparse matrices is a bit more involved than generating dense ones. You also pass number of rows and columns to the `sparse` method, but the nonzero element values (in sparse matrices only, nonzero elements are needed) are specified in CSC (Compressed Sparse Column) format[7]. CSC format is made of three arrays, containing column pointers, row indices, and the nonzero elements. A row indices array contains the row index of each element in the elements array. The column pointers array contains ranges of indices of elements that belong to the same column.

> **NOTE** `SparseMatrix` is not available in Python.

For the previous **M** matrix example (the same matrix used previously), the arrays for specifying the matrix in CSC format are as follows:

$$colPtrs = [0\ \ 1\ \ 2\ \ 4], \qquad rowIndices = [0\ \ 1\ \ 0\ \ 1], \qquad elements = [5\ \ 3\ \ 1\ \ 4]$$

The `colPtrs` array tells us that the elements from index 0 (inclusive) to 1 (non-inclusive), which is only element '5' belong to the first column. Elements from index 1 to 2, which is only element '3', belong to the second column. Finally, elements from index 2 to 4 (elements '1' and '4') belong to the third column. The row index of each element is given in the `rowIndices` array.

To create the `SparseMatrix` object corresponding to the matrix **M**, you would use this line of code:

```
val sm = Matrices.sparse(2,3,Array(0,1,2,4), Array(0,1,0,1),
     Array(5.,3.,1.,4.))
```

(Note that the indices are specified as `Int`s, and the values as `Double`s.)

You can convert `SparseMatrix` to `DenseMatrix` and vice versa with the corresponding `toDense` and `toSparse` methods. But you will need to explicitly cast the `Matrix` object to the appropriate class:

```
scala> import org.apache.spark.mllib.linalg. {DenseMatrix,SparseMatrix}
scala> sm.asInstanceOf[SparseMatrix].toDense
res0: org.apache.spark.mllib.linalg.DenseMatrix =
```

[7] Compressed Column Storage, Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, Jack Dongarra et al., http://netlib.org/linalg/html_templates/node92.html

```
5.0  0.0  1.0
0.0  3.0  4.0
scala> dm.asInstanceOf[DenseMatrix].toSparse
2 x 3 CSCMatrix
(0,0) 5.0
(1,1) 3.0
(0,2) 1.0
(1,2) 4.0
```

LINEAR ALGEBRA OPERATIONS ON LOCAL MATRICES

Similarly to vectors, you can access specific elements of a Matrix by indexing it like this:

```
scala> dm(1,1)
res1: Double = 3.0
```

You can efficiently create a transposed matrix using the transpose method:

```
scala> dm.transpose
res1: org.apache.spark.mllib.linalg.Matrix =
5.0  0.0
0.0  3.0
1.0  4.0
```

For other local matrix operations similar to vectors, conversion to Breeze matrices is necessary. The online repository contains `toBreezeM` and `toBreezeD` functions that you can use for converting local and distributed matrices to Breeze objects.

Once converted to Breeze matrices, you can use operations like element-wise addition and matrix multiplication. We leave it up to you to further explore the Breeze API.

7.2.2 Distributed matrices

Distributed matrices are necessary when you're using machine learning algorithms on huge datasets. They are stored across many machines, and they can have a large number of rows and columns. Instead of using `Int`s for indexing rows and columns, for distributed matrices use `Long`s. There are four types of distributed matrices in Spark, defined in the package `org.apache.spark.mllib.linalg.distributed`: `RowMatrix`, `IndexedRowMatrix`, `BlockMatrix`, and `CoordinateMatrix`.

ROWMATRIX

`RowMatrix` stores the rows of a matrix in an RDD of `Vector` objects. This RDD is accessible as the `rows` member field. Number of rows and columns can be obtained with `numRows` and `numCols`. `RowMatrix` can be multiplied by a local matrix (producing another `RowMatrix`) using the method `multiply`. `RowMatrix` also provides other useful methods, not available for other distributed implementations. We describe those later.

Every other type of Spark distributed matrix can be converted to a `RowMatrix` using the built-in `toRowMatrix` methods, but there are no methods for converting a `RowMatrix` to other distributed implementations.

IndexedRowMatrix

IndexedRowMatrix is an RDD of IndexedRow objects, each containing an index of the row and a Vector with row data. Although there is no built-in method for converting a RowMatrix to IndexedRowMatrix, it's fairly easy to do:

```
import org.apache.spark.mllib.linalg.distributed.IndexedRowMatrix
import org.apache.spark.mllib.linalg.distributed.IndexedRow
val rmind = new IndexedRowMatrix(rm.rows.zipWithIndex().map(x =>
     IndexedRow(x._2, x._1)))
```

CoordinateMatrix

CoordinateMatrix stores its values as an RDD of MatrixEntry objects, which contain individual entries and their *(i,j)* positions in the matrix. This is not an efficient way of storing data, so you should use CoordinateMatrix for storing only sparse matrices. Otherwise, it could consume too much memory.

BlockMatrix

BlockMatrix is the only distributed implementation with methods for adding and multiplying other distributed matrices. It stores its values as RDDs of tuples *((i,j), Matrix)*. In other words, BlockMatrix contains local matrices (blocks) referenced by their position in the matrix. Sub-matrices take up blocks of the same sizes (of "rows per block" and "columns per block" dimensions), except for the last sub-matrices, which can be smaller (to allow the total matrix to be of any dimensions). The validate method checks whether all blocks are of the same size (except the last ones).

Linear algebra operations with distributed matrices

Linear algebra operations with distributed matrix implementations are somewhat limited, so you will need to implement some of these yourself. For example, element-wise addition and multiplication of distributed matrices is available only for BlockMatrix matrices. The reason is that only BlockMatrices offer a way to efficiently handle these operations for matrices with many rows and columns.

Transposition is available only for CoordinateMatrix and BlockMatrix.

The other operations, like matrix inverse, for example, have to be done manually.

7.3 Linear regression

Now we finally get to do some machine learning. In this section, you'll learn how linear regression works and apply it on a sample data set. In the process, you'll learn how to analyze and prepare the data for linear regression and how to evaluate your model's performance. You will also learn some important concepts, such as bias-variance tradeoff, cross-validation, and regularization.

7.3.1 About linear regression

Historically, linear regression has been one of the most widely used regression methods and one of the basic analytical methods in statistics, and it's still widely used today. That's because modeling linear relationships is much easier than modeling nonlinear ones. Interpretation of resulting models is also easier. The theory behind linear regression also forms the basis for more advanced methods and algorithms in machine learning.

Like other types of regression, linear regression lets you use a set of independent variables to make predictions about a target variable and quantify the relationship between them. Linear regression makes an assumption that there is a *linear* relationship (hence the name) between the independent and target variables. Let's see what this means when we have only one independent and one target variable, which is also called *simple linear regression.* Using simple linear regression, we can plot the problem in two dimensions: the x-axis being the independent variable and the y-axis being the target variable. Later, we'll expand this into a model with more independent variables, which is called *multiple linear regression.*

7.3.2 Simple linear regression

As an example, we'll use the UCI Boston housing data set[8]. Although the data set is rather small and, as such, does not represent a Big Data problem, it's nevertheless appropriate for explaining machine learning algorithms in Spark. Besides, this enables you to use it on your local machine if you want to do so.

The data set contains mean values of owner-occupied homes in the suburbs of Boston and 13 features that can be used to predict home values. These features include the crime rate, number of rooms per dwelling, accessibility to highways, and so on.

For our simple linear regression example, we'll predict home prices based on the average number of rooms per dwelling. You might not really need to use linear regression to find out that the price of a house probably rises if there are more rooms in it. That is something obvious and intuitive. But linear regression does enable you to quantify that relationship—to say what is the expected price for a certain number of rooms? If we were to plot the average number of rooms on the x-axis and the average price on the y-axis, we would get output similar to that shown in figure 7.4.

There is obviously a correlation between the two variables: almost no expensive houses have a small number of rooms, and no inexpensive houses have a large number of rooms. Linear regression enables us to find a line that goes through the middle of these data points and, in that way, to approximate the most likely home price we could expect given an average number of rooms. We've already calculated this line as shown in figure 7.4. Let's see what the method is for finding it.

Generally, if you want to draw a line in two-dimensional space, you need two values: the *slope* of the line and the value at which the line intersects the y-axis, also called the *intercept.* If we denote the number of rooms as x, the function for calculating the home price as h (which stands for *hypothesis*), and the intercept and slope as w_0 and w_1, respectively, the line can be described with the following formula:

$$h(x) = w_0 + w_1 x$$

[8] UCI Machine Learning Repository, Housing Data Set, https://archive.ics.uci.edu/ml/datasets/Housing

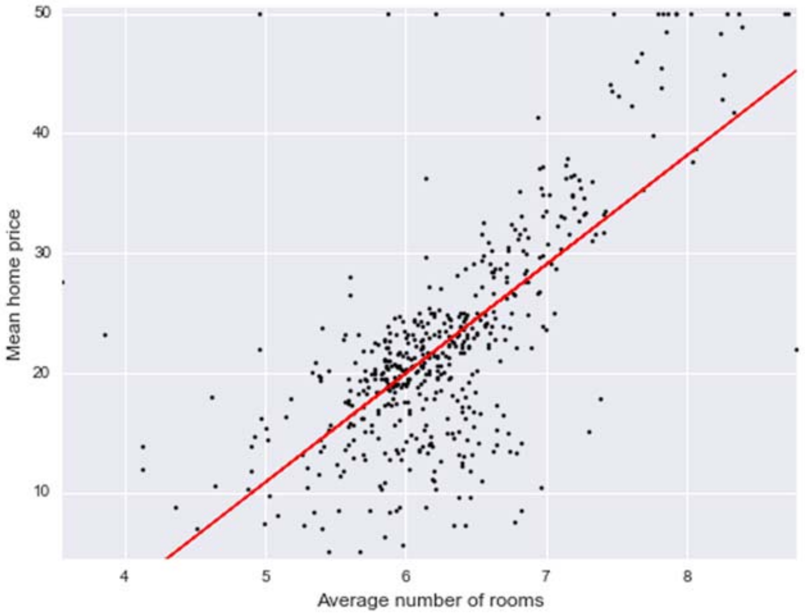


Figure 7.4 Mean home prices in Boston by average number of rooms per dwelling. Linear regression was used to find a best-fit line for this data (shown on the graph).

Our goal is to find the *weights* w_0 and w_1 that would best fit our data. Linear regression's method for finding appropriate weight values is to minimize the so-called *cost function*. The cost function returns a single value that can be used as a measure of how well the line, determined by weights, fits all examples in a data set. Different cost functions could be used. The one used in linear regression is the mean of the squared differences between predicted and real values of the target variable for all the *m* examples in the data set (mean squared error). The cost function (we call it *C* in the equation) can be written like this:

$$c(w_0, w_1) = \frac{1}{2m}\sum_{i=1}^{m}(h(x^{(i)}) - y^{(i)})^2 = \frac{1}{2m}\sum_{i=1}^{m}(w_0 + w_1 x^{(i)} - y^{(i)})^2$$

If we give this function a set of *m* examples $x^{(1)}$ to $x^{(m)}$ (with matching target values $y^{(1)}$ to $y^{(m)}$) and the weights w_0 and $w_{1,}$ which we think would be most appropriate for the data, the function would give us a single error value. If this value is lower than a second one, obtained for a different set of weights, that means our first model (determined by chosen weights w_0 and w_1) better fits the data set.

But how do we get the "best-fit weights?" We can find the minimum of the cost function. If we were to plot the cost function with respect to weights w_0 and w_1, it would form a curved plane in a 3-dimensional space, similar to the one in figure 7.5. The shape of the cost function depends on your data set. In our example, the cost function has a "valley" along which many points correspond to low error values. That

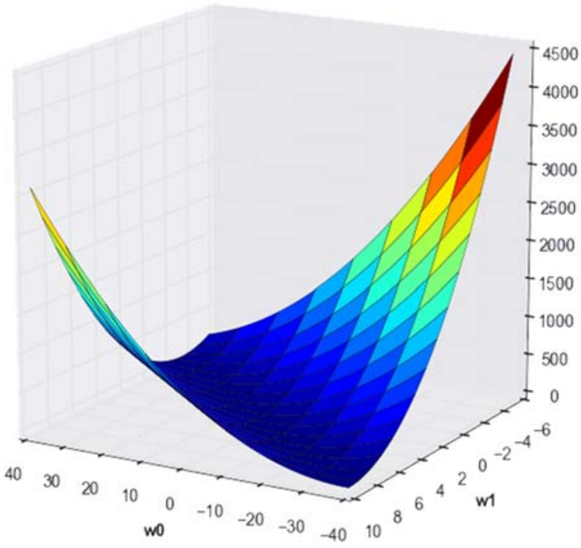


Figure 7.5 The cost function of the housing data set, depending on weights w_0 and w_1. The "valley" in the middle shows that many combinations of w_0 and w_1 can fit the data equally well.

means we could draw many lines (defined by weights w_0 and w_1) in figure 7.4 that would fit our data set equally well.

You use the mean squared error cost function in linear regression because it offers certain benefits: the squares of individual deviations can't cancel each other out (they are always positive), even though the corresponding deviations might be negative; the function is convex, which means there are no "local minima," only a global minimum; and an analytical solution for finding its minimum exists.

7.3.3 Expanding the model to multiple linear regression

There is a nice vectorized solution for finding the minimum of the cost function *C*, but let's first expand the model to a multiple linear regression one. As we said previously, expanding the model to a multiple linear regression means that examples will have more *dimensions* (independent variables). In our example, we need to add the remaining 12 dimensions of our housing data set. This adds additional information to the data set and enables the model to make better predictions based on that additional information. It also means that, from this point on, we will not be able to plot the data nor the cost function anymore, because the linear regression solution now becomes a 13-dimensional hyperplane (instead of a line in two dimensions).

After adding the remaining 12 dimensions to our data set, our hypothesis function becomes

$$h(x) = w_0 + w_1 x_1 + \ldots + w_n x_n = \boldsymbol{w}^T \boldsymbol{x}$$

where *n*, in our example, is equal to 12. On the right side, you can see the vectorized version of the same expression. To be able to introduce the vectorized notation (because the intercept value w_0 is multiplied by 1), we need to extend the original vector x with an additional component, x_0, which has a constant value of 1:

$$\boldsymbol{x}^T = [1 \; x_1 \; \dots \; x_n]$$

We can now rewrite the cost function of our multiple linear regression model like this:

$$c(\boldsymbol{w}) = \frac{1}{2m}\sum_{i=1}^{m}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} - y^{(i)})^2$$

This is also a vectorized version of the cost function (as indicated by bold letters in this equation).

Finding the minimum with the normal equation method

Finally, the vectorized solution to the problem of minimizing the cost function, in respect to the weights w_0 to w_n, is given by the *normal equation method* formula:

$$\boldsymbol{w} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}$$

X here is a matrix with m rows (m examples) and $n + 1$ columns (n dimensions plus 1s for x_0). **w** and **y** are vectors with $n + 1$ weights and m target values, respectively. Unfortunately, the scope of this book does not allow us to explain the math behind this formula.

Finding the minimum with gradient descent

Directly solving this equation with the previous formula can be very expensive and not easy to do (because of the matrix multiplications and matrix inversion calculations required), especially if there are a large number of dimensions and rows in the data set. So we'll use the *gradient descent method, which* is more commonly employed—and you can use it in Spark, too.

Gradient descent algorithms work iteratively. Such an algorithm starts from a certain point, representing a best-guess of the weight parameters' values (this point can also be randomly chosen), and for each weight parameter w_j, calculates a partial derivative of the cost function with respect to that weight parameter. Partial derivative tells the algorithm how to change the weight parameter in question to descend to the minimum of the cost function as quickly as possible. The algorithm then updates the weight parameters according to the calculated partial derivatives and calculates the value of the cost function at the new point. If the new value is less than some *tolerance* value, we say that the algorithm *converged,* and the process stops. See figure 7.6 for an illustration.

As an example of gradient descent algorithm, let's return to our simple linear regression example and its cost function, pictured again in figure 7.6. The dots on the white line in the figure are points the algorithm visits in each step. The black line itself is the shortest path from the starting point to the minimum of the cost function.

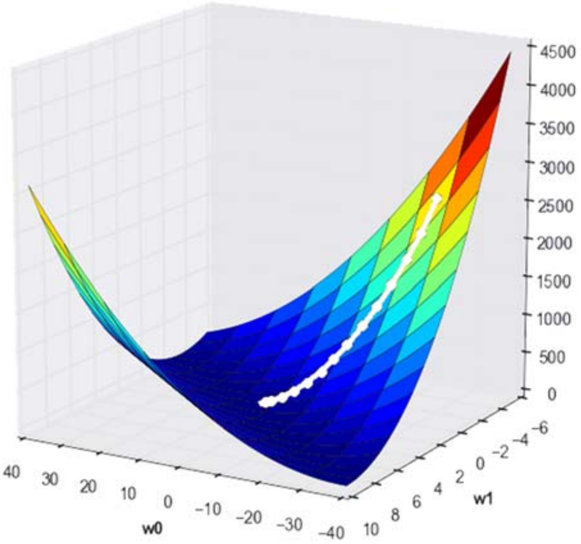


Figure 7.6 A gradient descent algorithm determines the minimum value for the cost function in the simple linear regression model for the housing data set. The white line connects the points the algorithm visits in each step.

The partial derivative of the cost function C, with respect to any weight parameter w_j, is given by this formula:

$$\frac{\partial}{\partial w_j} C(\boldsymbol{w}) = \frac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}$$

(Please note that x_0 is equal to 1 for all examples, as mentioned previously.)

If the partial derivative is negative, the cost function *decreases* with an *increase* of weight parameter w_j. You can now use this value to update the weight parameter w_j to *decrease* the value of your cost function. And you would do it for all weight parameters as part of a single step:

$$w_j: = w_j - \Upsilon\frac{\partial}{\partial w_j}C(\boldsymbol{w}) = \boldsymbol{w}_j - \Upsilon\frac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}, \textit{ for every } j$$

After updating all weight parameters, you would calculate the cost function again (the second point along the black line in figure 7.6), and if it's still unacceptably high, you would update the weights again using partial derivatives. You would repeat this process until *convergence* (the value of cost function remains stable).

Parameter γ (Greek letter *gamma*) is the *step size parameter that helps stabilize the algorithm. We will have more to say about the step size parameter later*.

7.4 Analyzing and preparing the data

That was a good dose of theoretical background for our linear regression example. Now it's time to implement all that using Spark's API. You will now download the housing data set, prepare the data, fit a linear regression model, and use the model to predict target values of some examples.

To begin, download the housing data set (housing.data) from our online repository (use our GitHub repository[9] and not the one from UCI machine learning repository because we changed the data set a bit). We will assume you cloned the GitHub repository to the /home/spark/first-edition folder in the virtual machine. You can find the description of the data set in the file ch07/housing.names.

First, start the Spark shell in your home directory and load the data with the following code:

```
import org.apache.spark.mllib.linalg.Vectors
val housingLines = sc.textFile("first-edition/ch07/housing.data", 6)
val housingVals = housingLines.map(x =>
     Vectors.dense(x.split(",").map(_.trim().toDouble)))
```

We are using six partitions for the housingLines RDD, but you can choose another value, depending on your cluster environment[10].

Now you have your data parsed and available as Vector objects. But before doing anything useful with it, acquaint yourself with the data first. The first step when dealing with any machine learning problem is to analyze the data and notice its distribution and interrelationships among different variables.

7.4.1 Analyzing data distribution

To get a feeling for the data you just loaded, you can calculate its *multivariate statistical summary*. You can obtain that value from the corresponding RowMatrix object like this:

```
import org.apache.spark.mllib.linalg.distributed.RowMatrix
val housingMat = new RowMatrix(housingVals)
val housingStats = housingMat.computeColumnSummaryStatistics()
```

Or you can use the Statistics object for the same purpose:

```
import org.apache.spark.mllib.stat.Statistics
val housingStats = Statistics.colStats(housingVals)
```

You can now use the obtained MultivariateStatisticalSummary object to examine average (the mean method), maximum (the max method), and minimum (the min method) values in each column of the matrix. For example, minimum values in the columns are these:

```
scala> housingStats.min
res0: org.apache.spark.mllib.linalg.Vector = [0.00632,0.0,0.46,0.0,0.385,
     3.561,2.9,1.1296,1.0,187.0,12.6,0.32,1.73,5.0]
```

You can also get the L1 norm (sum of absolute values of all elements per column) and L2 norm (also called *Euclidian norm*; equal to the length of a vector/column) for each

[9] https://github.com/spark-in-action/first-edition/blob/master/ch07/housing.data

[10] If you need more information about the number of partitions required, chapter 4 would be a good place to look for it.

column, with methods normL1 and normL2. Variance of each column can be obtained with the variance method.

DEFINITION *Variance* is a measure of dispersion of a data set and is equal to the average of squared deviations of values from their mean value. *Standard deviation* is calculated as the square root of variance. *Covariance* is a measure of how much two variables change relative to each other.

All of this can be very useful when examining data for the first time, especially when deciding whether *feature scaling* is necessary (described shortly).

7.4.2 Analyzing column cosine similarities

Understanding column cosine similarities is another thing that helps when analyzing data. *Column cosine similarities* represent an angle between two columns, viewed as vectors. A similar procedure can be used for other purposes as well (for example, for finding similar products, or similar articles).

You obtain column cosine similarities from the RowMatrix object:

```
val housingColSims = housingMat.columnSimilarities()
```

PYTHON The columnSimilarities method is not available in Python.

The resulting object is a distributed CoordinateMatrix containing an upper-triangular matrix (*upper-triangular matrices* contain data only above their diagonal). Value at *i*-th row and *j*-th column in the resulting housingColSims matrix gives a measure of similarity between *i*-th column and *j*-th column in the housingMat matrix. The values in the housingColSims matrix can go in value from –1 to 1. A value of –1 means the two columns have completely opposite orientations (directions), a value of 0 means they are orthogonal to one another, and a value of 1 means the two columns (vectors) have the same orientation.

The easiest way to see the contents of this matrix is to convert it to a Breeze matrix using our toBreezeD method and then print the output with our utility method printMat that you can find in our repository listing, which we omit due to brevity. To do this, first paste the printMat method definition into your shell and execute the following:

```
printMat(toBreezeD(housingColSims))
```

This will pretty-print the contents of the matrix (you can also find the expected output in our online repository). If you look at the last column of the result, it gives you a measure of how well each dimension in the data set corresponds to the target variable (average price). This is the contents of the last column: 0.224, 0.528, 0.693, 0.307, 0.873, 0.949, 0.803, 0.856, 0.588, 0.789, 0.897, 0.928, 0.670, 0.000. The biggest value here is the sixth value (0.949), which corresponds to the column containing the average number of rooms. Now you can see that it was no coincidence that we chose that exact column for our previous simple linear regression example—it has the strongest similarity with the target value and thus represents the most appropriate candidate for simple linear regression.

7.4.3 Computing the covariance matrix

Another method for examining similarities between different columns (dimensions) of the input set is the covariance matrix. It's important in statistics for modeling linear correspondence between variables. In Spark, you compute the covariance matrix similarly to column statistics and column similarities, using the `RowMatrix` object:

```
val housingCovar = housingMat.computeCovariance()
printMat(toBreezeM(housingCovar))
```

PYTHON The `computeCovariance` method is not available in Python.

The expected output is also available in our online repository. If you spend a moment studying it, you'll notice that there is a large range of values in the matrix and that some of them are negative and some are positive. You'll also probably notice that the matrix is symmetric (that is, each *(i, j)* element is the same as a *(j, i)* element).

This is because the variance-covariance matrix contains the variance of each column on its diagonal and covariance of the two matching columns on all other positions. If a covariance of two columns is zero, there is no linear relationship between them. Negative values mean that the values in the two columns move in opposite directions from their averages, whereas the opposite is true for positive values.

Spark also offers two other methods for examining correlations between series of data: Spearman's and Pearson's methods. Because an explanation of those methods is beyond the scope of this book, you can access them through the `org.apache.spark.mllib.stat.Statistics` object.

7.4.4 Transforming to labeled points

Now that we've examined the data set, we can get on to preparing the data for linear regression. First, we have to put each example in the data set in a structure called `LabeledPoint, which` is used in most of Spark's machine learning algorithms. It contains the target value and the vector with the features. `housingVals` containing Vector objects with all variables, and the equivalent `housingMat RowMatrix` object, were useful when we were examining the data set as a whole (in the previous sections), but now we need to separate the target variable (the label) from the features.

To do that, we can just transform the `housingVals` RDD (the target variable is in the last column):

```
import org.apache.spark.mllib.regression.LabeledPoint
val housingData = housingVals.map(x => {
  val a = x.toArray
  LabeledPoint(a(a.length-1), Vectors.dense(a.slice(0, a.length-1)))
})
```

7.4.5 Splitting the data

The second important step is splitting the data into training and validation sets. A *training* set is used to train the model, and a *validation* set is used to see how well the model performs on data that wasn't used to train it. The usual split ratio is 80% for the training set and 20% for the validation set.

We can split the data very easily in Spark with the RDD's built-in `randomSplit` method:

```
val sets = housingData.randomSplit(Array(0.8, 0.2))
val housingTrain = sets(0)
val housingValid = sets(1)
```

The method returns an array of RDDs, each containing approximately the requested percentage of orginal data.

7.4.6 Feature scaling and mean normalization

We're not done with data preparation yet. As you probably noticed when we were examining distribution of our data, there are large differences in data spans between the columns. For example, data in the first column goes from 0.00632 to 88.9762, and the data in the fifth column from 0.385 to 0.871.

Interpreting results from a linear regression model trained with data like this can be difficult and can render some data transformations (which we're going to perform in the coming sections) problematic. It's often a good idea to standardize the data first, which can't hurt your model. There are two ways you can do this: with *feature scaling* and with *mean normalization.*

Feature scaling means that the ranges of data are scaled to comparable sizes. *Mean normalization* means that the data is translated so that the averages are roughly zero. We can do both in a single pass, but we need a `StandardScaler` object to do that. In the constructor, you specify which of the standardization techniques you want to use (we will use both) and then you fit it according to some data:

```
import org.apache.spark.mllib.feature.StandardScaler
val scaler = new StandardScaler(true, true).
  fit(housingTrain.map(x => x.features))
```

Fitting finds column summary statistics of the input data and uses these statistics (in the next step) to do the actual scaling. We fitted the scaler according to the training set, and we'll then use the same statistics to scale both the training and validation sets (only data from the training set should be used for fitting the scaler):

```
val trainScaled = housingTrain.map(x => LabeledPoint(x.label,
     scaler.transform(x.features)))
val validScaled = housingValid.map(x => LabeledPoint(x.label,
     scaler.transform(x.features)))
```

Now you're finally ready to use the housing data set for linear regression.

7.5 Fitting and using a linear regression model

A linear regression model in Spark is implemented by the class `LinearRegressionModel` in the package `org.apache.spark.mllib.regression`. It's produced by fitting a model and holds the fitted model's parameters. When you have fitted a `LinearRegressionModel` object, you can use its `predict` method on individual `Vector` examples to predict the corresponding target variables. You construct the model using the `LinearRegressionWithSGD` class, which implements the algorithm used for training the model. But you can do this in two ways. The first is the standard Spark way of invoking the static `train` method:

```
val model = LinearRegressionWithSGD.train(trainScaled, 200, 1.0)
```

Unfortunately, this does not allow you to find the intercept value (only the weights), so use the second, nonstandard method:

```
import org.apache.spark.mllib.regression.LinearRegressionWithSGD
val alg = new LinearRegressionWithSGD()           ◁— Instantiate object
alg.setIntercept(true)                         ◁— Set option to find intercept value
alg.optimizer.setNumIterations(200)               ◁— Set number of iterations to run
trainScaled.cache()                              | Caching input data is important
validScaled.cache()                              |
val model = alg.run(trainScaled)                  ◁— Start training of model
```

Within a few seconds of executing this code, you'll have your Spark linear regression model ready to use for predictions. The data sets are cached, which is important for iterative algorithm, such as machine learning ones, because they tend to reuse the same data many times.

7.5.1 Predicting the target values

You can now use the trained model to predict the target values of vectors in the validation set by running `predict` on every element. Our validation set contains labeled points, but you only need the features. You also need the predictions together with the original labels, so you can compare them. This is how you can map labeled points to pairs of predicted and original values:

```
val validPredicts = validScaled.map(x => (model.predict(x.features),
     x.label))
```

The moment of truth has arrived. You can see how well your model is doing on the validation set by simply examining the contents of `validPredicts`:

```
scala> validPredicts.collect()
res123: Array[(Double, Double)] = Array((28.250971806168213,33.4),
     (23.050776311791807,22.9), (21.278600156174313,21.7),
     (19.067817892581136,19.9), (19.463816495227626,18.4), ...
```

Some predictions are close to original labels and some are further off. To quantify the success of your model, calculate the root mean squared error (the root of the cost function defined previously):

```
scala> math.sqrt(validPredicts.map{case(p,l) => math.pow(p-l,2)}.mean())
res0: Double = 4.775608317676729
```

The average value of target variables (home prices) is 22.5—which we learned earlier when we were calculating column statistics—so a root mean squared error of 4.78 seems rather large. But if we take into account that the variance of home prices is 84.6, the number suddenly looks much better.

7.5.2 Evaluating the model's performance

This is not the only way you can evaluate the performance of your regression model. Spark offers the `RegressionMetrics` class for this purpose. You give it an RDD with pairs of predictions and labels, and it returns several useful evaluation metrics:

```
scala> import org.apache.spark.mllib.evaluation.RegressionMetrics
scala> val validMetrics = new RegressionMetrics(validPredicts)
scala> validMetrics.rootMeanSquaredError
res1: Double = 4.775608317676729
scala> validMetrics.meanSquaredError
res2: Double = 22.806434803863162
```

Beside the root mean squared error that you previously calculated yourself, `RegressionMetrics` also gives you the following:

- `meanAbsoluteError`—Average absolute difference between a predicted and real value (3.044 in our case).
- `r2`—Coefficient of determination R^2 (0.71 in our case) is a value between 0 and 1 and represents the *fraction of variance explained*. It's a measure of how much a model accounts for the variation in the target variable (predictions) and how much of it is "unexplained." A value close to 1 means that the model explains a large part of variance in the target variable.
- `explainedVariance`—A value similar to R^2 (0.711 in our case).

All of these are used in practice, but coefficient of determination can give you somewhat misleading results (it tends to rise when the number of features increases, whether they are relevant or not). For that reason, we'll use the root mean squared error (RMSE) from now on.

7.5.3 Interpreting the model parameters

The set of weights the model has learned can tell you something about the influence of individual dimensions on the target variable. If a particular weight is near zero, the corresponding dimension does not contribute to the target variable (price of housing) in a significant way (assuming the data has been scaled—otherwise even low-range features might be important).

You can inspect absolutes of the individual weights with the following snippet of code:

```
scala> println(model.weights.toArray.map(x => x.abs).
  | zipWithIndex.sortBy(_._1).mkString(", "))
(0.112892822124492423,6), (0.163296952677502576,2), (0.588838584855835963,3),
     (0.939646889835077461,0), (0.994950411719257694,11),
     (1.263479388579985779,1), (1.660835069779720992,9),
     (2.030167784111269705,4), (2.072353314616951604,10),
     (2.419153951711214781,8), (2.794657721841373189,5),
     (3.113566843160460237,7), (3.323924359136577734,12)
```

The model's `weights` Vector is first converted to a Scala `Array`, then the absolute values are calculated, an index is attached to each weight using the Scala's `zipWithIndex` method, and finally, the weights are sorted by their values.

You can see that the most influential dimension of the data set is the one with index 12, which corresponds to the LSTAT column, or the "percentage of lower status of the population." (You can find the column descriptions in the `housing.names` file in the book's online repository.) The second-most influential dimension is the column with index 7, or "weighted distances to five Boston employment centers." And so on.

The two least influential dimensions are the "proportion of owner-occupied units built prior to 1940" and the "proportion of non-retail business acres per town." Those dimensions can be removed from the dataset without influencing the model's performance significantly. In fact, that might even improve it a bit because in that way, the model would "get focused" on the important features more.

7.5.4 Loading and saving the model

Because training a model using lots of data can be an expensive and lengthy operation, Spark offers a way to save the model to a file system as a Parquet file (covered in chapter 5) and simply load it later, when needed. Most Spark MLlib models can be saved using the `save` method. You just pass a `SparkContext` instance and a file system path to it, similar to this:

```
model.save(sc, "chapter07output/model")
```

Spark uses the path for creating a directory and creates two Parquet files in it: `data` and `metadata`.

In case of linear regression models, the `metadata` file contains the model's implementation class name, the implementation's version, and the number of features in the model. The `data` file contains the weights and the intercept of the linear regression model.

To load the model, use the corresponding `load` method, again passing to it a `SparkContext` instance and the path to the directory with the saved model. For example:

```
import org.apache.spark.mllib.regression.LinearRegressionModel
val model = LinearRegressionModel.load(sc, "ch07output/model")
```

The model can then be used for predictions.

7.6 Tweaking the algorithm

In section 7.3.3, you saw the gradient descent formula:

$$w_j\!: = w_j - \Upsilon\frac{\partial}{\partial w_j}C(\boldsymbol{w}) = \boldsymbol{w}_j - \Upsilon\frac{1}{m}\sum_{i=1}^{m}(h(\boldsymbol{x}^{(i)}) - y^{(i)})x_j^{(i)}, \textit{ for every } j$$

Parameter γ (Greek letter *gamma*) in the formula is the *step size parameter,* which helps to stabilize the gradient descent algorithm. But it can be difficult to find the optimal value for this parameter. If it's too small, the algorithm will take too many small steps to converge. If it's too large, the algorithm might never converge. The right value depends on the data set.

It's similar with number of iterations. If it's too large, fitting the model will take too much time. If it's too small, the algorithm might not reach the minimum.

Although we only set the number of iterations for our previous run of the linear regression algorithm (the step size we used had the default value of 1.0), you can set both of these parameters when using `LinearRegressionWithSGD`. But you can't tell Spark "iterate until the algorithm converges" (which would be ideal). You have to find the optimal values for these two parameters yourself.

7.6.1 Finding the right step size and number of iterations

One way to find satisfactory values for these two parameters is to experiment with several combinations and find the one that gives the best results. We put together a function that can help you do this. Find the `iterateLRwSGD` function in our online repository (in the `ch07-listings.scala` and `ch07-listings.py` files) and paste it into your Spark shell. This is the complete function:

```
import org.apache.spark.rdd.RDD
def iterateLRwSGD(iterNums:Array[Int], stepSizes:Array[Double],
    train:RDD[LabeledPoint], test:RDD[LabeledPoint]) = {
  for(numIter <- iterNums; step <- stepSizes) {
    val alg = new LinearRegressionWithSGD()
    alg.setIntercept(true).optimizer.setNumIterations(numIter).
        setStepSize(step)
    val model = alg.run(train)
    val rescaledPredicts = train.map(x =>
        (model.predict(x.features), x.label))
    val validPredicts = test.map(x => (model.predict(x.features), x.label))
    val meanSquared = math.sqrt(rescaledPredicts.map(
        {case(p,l) => math.pow(p-l,2)}).mean())
    val meanSquaredValid = math.sqrt(validPredicts.map(
        {case(p,l) => math.pow(p-l,2)}).mean())
    println("%d, %5.3f -> %.4f, %.4f".format(numIter,
        step, meanSquared, meanSquaredValid))
  }
}
```

The `iterateLRwSGD` function takes two arrays, containing different numbers of iterations and step size parameters, and two RDDs, containing training and validation data.

For each combination of step size and number of iterations in the input arrays, the function returns the root mean squared error (RMSE) of training and validation sets. Here's what the printout should look like:

```
scala> iterateLRwSGD(Array(200, 400, 600), Array(0.05, 0.1, 0.5, 1, 1.5, 2,
     3), trainScaled, validScaled)
200, 0.050 -> 7.5420, 7.4786
200, 0.100 -> 5.0437, 5.0910
200, 0.500 -> 4.6920, 4.7814
200, 1.000 -> 4.6777, 4.7756
200, 1.500 -> 4.6751, 4.7761
200, 2.000 -> 4.6746, 4.7771
200, 3.000 -> 108738480856.3940, 122956877593.1419
400, 0.050 -> 5.8161, 5.8254
400, 0.100 -> 4.8069, 4.8689
400, 0.500 -> 4.6826, 4.7772
400, 1.000 -> 4.6753, 4.7760
400, 1.500 -> 4.6746, 4.7774
400, 2.000 -> 4.6745, 4.7780
400, 3.000 -> 25240554554.3096, 30621674955.1730
600, 0.050 -> 5.2510, 5.2877
600, 0.100 -> 4.7667, 4.8332
600, 0.500 -> 4.6792, 4.7759
600, 1.000 -> 4.6748, 4.7767
600, 1.500 -> 4.6745, 4.7779
600, 2.000 -> 4.6745, 4.7783
600, 3.000 -> 4977766834.6285, 6036973314.0450
```

You can see several things from this output. First, the testing RMSE is always greater than training RMSE (except for some corner cases). That's to be expected. Furthermore, for every number of iterations, both errors decline rapidly as step size increases, following some inverse exponential function. That makes sense because for smaller numbers of iterations and smaller step sizes, there were simply not enough iterations to get to the minimum.

Then the error values flatten out, more quickly for larger numbers of iterations. This also makes sense because there are some limitations to how well you can fit a data set. And models fitted with larger numbers of iterations will perform better. For step size value of 3, the error values explode. This step size value is simply too large, and the algorithm misses the minimum. It seems that step size of 0.5 or 1.0 gives the best results if the number of iterations stays the same.

You may also have noticed that running more iterations does not help much. For example, a step size of 1.0 with 200 iterations gives you almost the same training RMSE as with 600 iterations.

7.6.2 Adding higher-order polynomials

It seems that the testing RMSE of 4.7760 is the lowest error you can get for the housing dataset. But your model can actually do better. Very often data does not follow a simple linear formula (a straight line in a 2-dimensional space) but may be some kind of a curve. Curves can often be described with functions containing higher-order polynomials. For example:

$$h(x) = w_0 x^3 + w_1 x^2 + w_2 x + w_3$$

This hypothesis is capable of matching data governed by some nonlinear relationship. You will see an example of this in the next section.

Spark doesn‘t offer a method of training a nonlinear regression model that includes higher-order polynomials, such as the preceding hypothesis. Instead, you can employ a little trick and do something that has a similar effect: you can expand your data set with additional features obtained by multiplying the existing ones. For example, if you have features x_1 and x_2, you could expand the data set to include x_1^2 and x_2^2. Adding the *interaction term* $x_1 x_2$ helps in cases when x_1 and x_2 influence the target variable together.

Let‘s do that now with our data set. You will use a simple function for mapping each `Vector` in the data set to include the square of each feature:

```
def addHighPols(v:Vector): Vector =
{
  Vectors.dense(v.toArray.flatMap(x => Array(x, x*x)))
}
val housingHP = housingData.map(x => LabeledPoint(x.label,
     addHighPols(x.features)))
```

Add squares to a Vector

Map original data set

`housingHP` RDD now contains `LabeledPoints` from our original `housingData` RDD, but expanded with additional features containing second order polynomials. We now have 26 features instead of the previous 13:

```
scala> housingHP.first().features.count()
res0: Int = 26
```

Now it’s necessary to once again go through the process of splitting the data set for training and testing subsets and to scale the data in the same way we did previously:

```
val setsHP = housingHP.randomSplit(Array(0.8, 0.2))
val housingHPTrain = setsHP(0)
val housingHPValid = setsHP(1)
val scalerHP = new StandardScaler(true, true)
scalerHP.fit(housingHPTrain.map(x => x.features))
val trainHPScaled = housingHPTrain.map(x => LabeledPoint(x.label,
     scalerHP.transform(x.features)))
val validHPScaled = housingHPValid.map(x => LabeledPoint(x.label,
     scalerHP.transform(x.features)))
trainHPScaled.cache()
validHPScaled.cache()
```

You can now see how the new model behaves with different numbers of iterations and step sizes:

```
iterateLRwSGD(Array(200, 400), Array(0.4, 0.5, 0.6, 0.7, 0.9, 1.0, 1.1, 1.2,
     1.3, 1.5), trainHPScaled, validHPScaled)
```

As you can see from the results (omitted for brevity, but available in our online repository), RMSE explodes for the step size of 1.3, and you get the best results for the step size of 1.1. The error values are lower than before. The best RMSE is 3.9836 (for 400 iterations), compared to 4.776 before. You can conclude that adding higher-order polynomials helped the linear regression algorithm find a better performing model.

But is this the lowest RMSE you can get with this data set? Let‘s see what happens if you increase the number of iterations (and use the best-performing step size of 1.1):

```
scala> iterateLRwSGD(Array(200, 400, 800, 1000, 3000, 6000), Array(1.1),
     trainHPScaled, validHPScaled)
200, 1.100 -> 4.1605, 4.0108
400, 1.100 -> 4.0378, 3.9836
800, 1.100 -> 3.9438, 3.9901
1000, 1.100 -> 3.9199, 3.9982
3000, 1.100 -> 3.8332, 4.0633
6000, 1.100 -> 3.7915, 4.1138
```

With more iterations, the testing RMSE is even starting to increase. (Depending on your data set split, you may get different results.) So which step size should you choose? And why is RMSE increasing?

7.6.3 Bias-variance tradeoff and model complexity

The situation where the testing RMSE is increasing while the training RMSE is decreasing is known as *overfitting*. What happens is that the model gets too attuned to the "noise" in the training set and becomes less accurate when analyzing new, real-world data that does not possess the same properties as the training set. There is also an opposite term—*underfitting*—where the model is too simple and is incapable of adequately capturing the complexities of the data. Understanding these phenomena is important for correctly using machine learning algorithms and getting the most out of your data.

For example, figure 7.7 shows a sample data set (red circles) following a quadratic function. The linear model (left-hand graph) is not capable of properly modeling the data. The quadratic function in the middle is just about right, and the function with higher-order polynomials on the right overfits the data set.

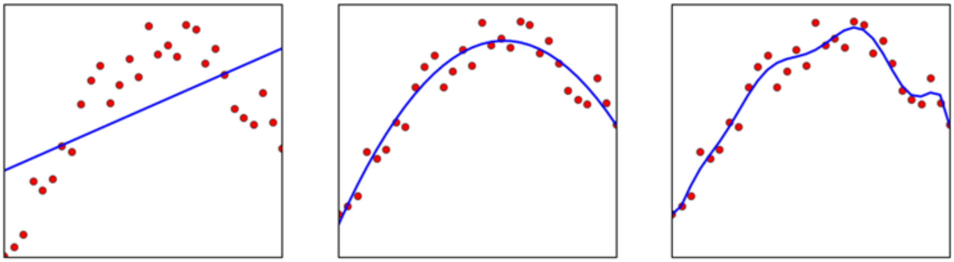


Figure 7.7 The linear model (left) underfits the data set, a model with higher-order polynomials (right) overfits it, and a quadratic model (middle) fits nicely.

You normally want your model to fit the data in your training data set, but also to be expandable to some other, presently unknown data. It's not necessarily possible to do both perfectly.

And that leads us to the *bias-variance tradeoff*. Bias here pertains to the model. For example, the linear model on the left-hand side of figure 7.7 has a high bias: it's assuming the linear relationship between the independent and target variables, so it's biased. The model on the right-hand side has a high variance because the values it predicts are oscillating more. Bias-variance tradeoff says that you can't necessarily have both at the same time and that you need to seek an equilibrium, or middle ground.

How do you know if your model has high bias (it's underfitted) or high variance (it's overfitted)?

Let us return to our example. Generally, overfitting occurs when the ratio of model complexity and training set size gets large. If you have a complex model, but also a relatively large training set, overfitting is less likely to occur. You saw that RMSE on our validation set started to rise when you added higher-order polynomials and trained the model with more iterations. Higher-order polynomials bring more complexity to the model, and more iterations overfit the model to the data while the algorithm is converging. Let's see what happens if we try even more iterations:

```
scala> iterateLRwSGD(Array(10000, 15000, 30000, 50000), Array(1.1),
     trainHPScaled, validHPScaled)
10000, 1.100 -> 3.7638, 4.1553
15000, 1.100 -> 3.7441, 4.1922
30000, 1.100 -> 3.7173, 4.2626
50000, 1.100 -> 3.7039, 4.3163
```

You can see that the training RMSE continues to decrease while the testing RMSE continues to rise. And that's typical for an overfitting situation: training error falls and then plateaus (which would happen for even more iterations), and testing error falls and then starts to rise, meaning that the model learns training set–specific properties, instead of characteristics representative of the whole population. If you were to plot this, you would get a graph similar to figure 7.8.

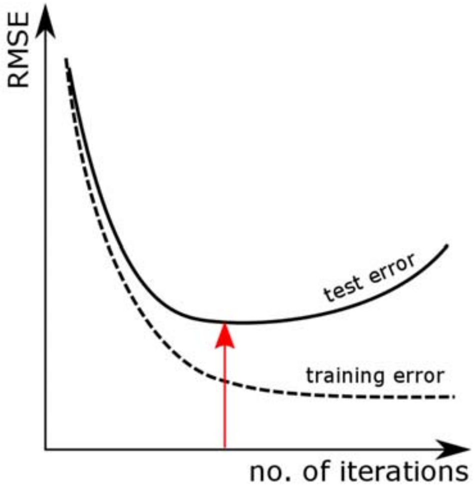


Figure 7.8 Error as a function of the number of iterations used. Test root means square error falls but then starts to rise at a certain point. Parameters corresponding to that point should be chosen for the model because the model is starting to overfit the data.

To answer the question of which values for the number of iterations and step size to choose: choose the values corresponding to the minimum of the testing RMSE curve, at the point before it starts to rise. In this case, 400 iterations and step size of 1.1 give very good results (testing RMSE of 3.98).

7.6.4 Plotting residual plots

But how can you tell if you need to keep adding higher-order polynomials? Or if you need to add any in the first place? And where do you stop? Examining *residual plots* can help you answer those questions.

Residual is the difference between the predicted and actual values of the target variable. In other words, for a single example in your training data set, the residual is the difference between its label's value and what your model says the label's value should be. Residual plots have residuals on the *y*-axis and the predicted values on the *x*-axis.

Residual plot should show no noticeable patterns—it should have the same height at all points on the *x*-axis, and if you were to plot a best-fit line (or a curve) through the plotted values, the line should stay flat. If it shows a shape similar to the letter *u* (or inverted *u*), that means a nonlinear model would be more appropriate for some of the dimensions.

The two residual plots for our two models (the original linear regression model and the one with added second-order polynomials) are shown in figure 7.9. The one on the left shows a shape of inverted *u*-curve. The one on the right, although still not perfect, shows an improvement as the shape is more balanced.

As we said, the new residual plot is still not perfect, and further dimension transformations might help, but probably not much. A line in the lower right-hand part of both figures is also visible. This is due to several *outliers*, or points that represent some

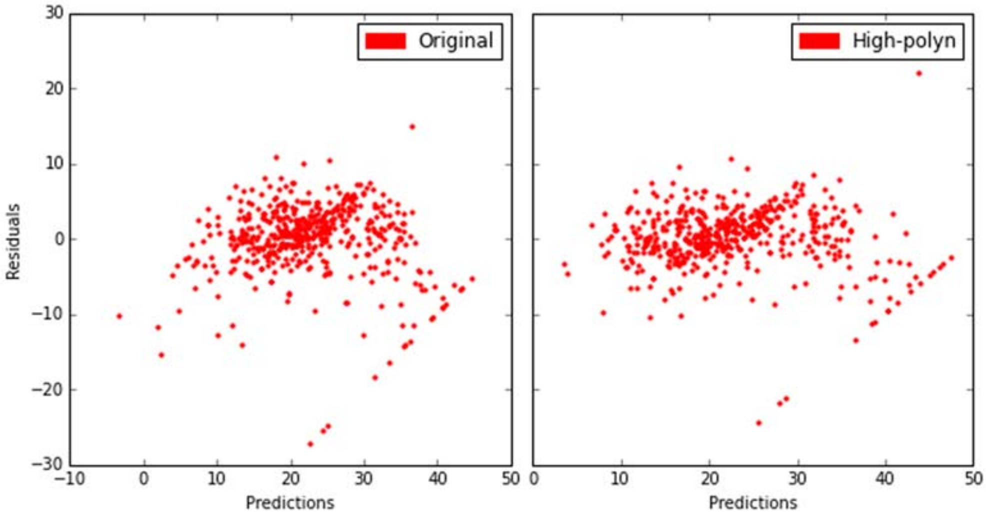


Figure 7.8 Residual plots for two linear regression models. The model on the left was fitted using the original housing data set and shows an inverted *u*-curve shape. The one on the right was fitted using the data set with added second-order polynomials and shows a more balanced pattern.

kind of exceptions. In this case, there are several instances of expensive houses ($50,000) that should otherwise be not as expensive. This could also be caused by a missing variable—some factor making a house expensive (aesthetics, for example), but not present in the data set.

Residual plots can also help you in a number of other situations. For example, if the plot shows a *fan-in* or a *fan-out* shape (the residuals show greater variance at one end of the plot than at the other, a phenomenon called *heteroscedasticity*[11]), one solution besides adding higher-order polynomials might be to transform the *target variable* logarithmically so that your model predicts *log(y)* (or some other function) and not *y*.

Further discussion of this important topic is beyond the scope of this book. The main thing to remember is that you should always spend some time studying residual plots to get further information on how your model is actually doing.

7.6.5 Avoiding overfitting by using regularization

Let's get back to overfitting. You have seen how it decreases your model's performance. You can avoid overfitting using a method called *regularization*, which increases the bias of your model and decreases variance by penalizing large values in the model parameters.

Regularization adds an additional element (we denote it as ?) to the cost function that penalizes complexity in the model. There are different regularization types. The most common ones are L1 and L2 regularizations (named after L1 and L2 norms discussed in section 7.4.2), and they are the ones available in Spark. Linear regression with L1 regularization is called *Lasso regression*, and the one with L2 regression is called *Ridge regression*.

The cost function with the regularization element ? looks like this:

$$C(w) = \frac{1}{2m}\sum_{i=1}^{m}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} - y^{(i)})^2 + \beta = \frac{1}{2m}\sum_{i=1}^{m}(\boldsymbol{w}^T\boldsymbol{x}^{(i)} - y^{(i)})^2 + \lambda\|\boldsymbol{w}\|_{\mathrm{I/II}}$$

β is a product of two elements: γ, the regularization parameter, and L1 ($\|w\|_{\mathrm{I}}$) or L2 ($\|w\|_{\mathrm{II}}$) norm of the weight vector. As we said in section 7.4.2, L1 norm is the sum of absolute values of vector's elements, and L2 norm is the square root of the sum of squares of vector's elements, which is equal to the length of the vector.

What the regularization actually does is increase the error in proportion to absolute weight values. In that way, the optimization function tries to deemphasize individual dimensions and slow down the algorithm as the weights get larger. L1 regularization (Lasso regression) is more aggressive in this process. It's capable of reducing individual weights to zero and thus completely removing some of the features from the data set.

In addition, both L1 and L2 regularizations in Spark decrease the step size in proportion to the number of iterations. This means that the longer the algorithm runs,

[11] For further information, see the Wikipedia page on heteroscedasticity (https://en.wikipedia.org/wiki/Heteroscedasticity).

the smaller steps it will take. (This is not related to regularization per se, but is part of L1 and L2 regularization implementations in Spark.)

Regularization can help you in situations where you're overfitting the model to the data set. By increasing the regularization parameter (λ), you can decrease overfitting. Besides that, regularization can help you get to lower error values quicker, when you have many dimensions, because it lowers the influence of dimensions that have less impact on performance.

But the downside is that regularization requires you to configure an extra parameter, which adds additional complexity to the process.

USING LASSO AND RIDGE REGRESSIONS IN SPARK

In Spark, you can set Lasso and Ridge regressions manually by changing the `regParam` and `updater` properties of `LinearRegressionWithSGD.optimizer` object or by using `LassoWithSGD` and `RidgeRegressionWithSGD` classes. The latter is what we did.

You can find two additional methods in our online repository: `iterateLasso` and `iterateRidge`. They are similar to the `iterateLRwSGD` we used before, but they take an additional `regParam` argument and train different models.

You can try these two methods and see the RMSE values Lasso and Ridge regression give on the data set with second-order polynomials we used before (`trainHPScaled` and `validHPScaled`) with the same step size as before and with the value of regression parameter of 0.01, which gives the best results:

```
iterateRidge(Array(200, 400, 1000, 3000, 6000, 10000), Array(1.1), 0.01,
     trainHPScaled, validHPScaled)
iterateLasso(Array(200, 400, 1000, 3000, 6000, 10000), Array(1.1), 0.01,
     trainHPScaled, validHPScaled)
```

The results (available in our repository online) show that Ridge gives lower MRSE than Lasso regression and that Ridge is better even than the ordinary least squares (OLS) regression used previously (3.966 for 1000 iterations instead of 3.984 for 400 iterations). Note that the increase in test RMSE, which was happening for numbers of iterations larger than 400 and which was the effect of overfitting, also happens for Ridge and Lasso regressions. Only for Ridge regressions, the overfitting kicks in later. If we were to increase the regularization parameter, we would see MRSE increase later, but the MRSE levels would be greater.

Which regularization method and which regularization parameter you should choose is difficult to say because it depends on your data set. You should apply an approach similar to the one we used for finding the number of iterations and the step size (train several models with different parameters and pick the ones with the lowest error).

The most common method for doing this is *k-fold cross-validation.*

7.6.6 K-fold cross-validation

K-fold cross-validation is a method of model validation. It consists of dividing the data set into *k* subsets of roughly equal sizes and training *k* models, excluding a different subset each time. The excluded subsets are used as the validation set and the union of all the remaining subsets as the training set.

For each set of parameters you want to validate, train all *k* models and calculate the mean error across all *k* models. Finally, you choose the set of parameters giving you the smallest average error.

Why is this important? Because fitting a model depends very much on the training and validation sets used. If you take our housing data set, split it randomly into training and validation sets again, and then go through all the actions we did in this chapter, you will notice that the results and parameters will be different, maybe even dramatically so. K-fold cross-validation can help you decide which of the parameter combinations to choose.

We will have more to say about k-fold cross-validation when we talk about Spark's new ML Pipeline API in the next chapter.

7.7 Optimizing linear regression

We have a couple more things to say about linear regression optimization. As you saw in earlier examples, `LinearRegressionSGD` (and its parent class `GeneralizedLinearAlgorithm`) has an `optimizer` member object you can configure. We previously used the default `GradientDescent` optimizer and configured it with the number of iterations and the step size.

There are two additional methods you can employ to make linear regression find the minimum of the cost function faster. The first is to configure the `GradientDescent` optimizer as a mini-batch stochastic gradient descent. The second is to use Spark's LBFGS optimizer (see section 7.7.2).

7.7.1 Mini-batch stochastic gradient descent

As explained in section 7.4.2, gradient descent updates the weights in each step by going through the whole data set. If you recall, the formula used for updating each weight parameter is this:

$$w_j\colon = w_j - \Upsilon \frac{1}{m} \sum_{i=1}^{m} \left(h(\boldsymbol{x}^{(i)}) - y^{(i)} \right) x_j^{(i)}$$

This is also called *batch gradient descent* (or BGD, for short). In contrast, mini-batch stochastic gradient descent uses only a subset of data in each step, and instead of *i* going from 1 to *m* (the whole data set), it only goes from 1 to *k* (as some fraction of *m*). If *k* is equal to 1—which means the algorithm considers only one example in each step—the optimizer is simply called stochastic gradient descent (SGD).

Mini-batch SGD is much less computationally expensive, especially when parallelized, but it compensates for this parallelization with more iterations. It has more difficulties to converge, but it gets to the minimum close enough (except in some rare cases). If mini-batch size (*k*) is small, the algorithm is more *stochastic,* meaning it will have a more random route toward the cost function minimum. If *k* is larger, the algorithm will be more stable. In both cases, though, it reaches the minimum and can get very close to BGD results.

Let us now see how to use mini-batch SGD in Spark. The same `GradientDescent` optimizer we used before is used for mini-batch SGD, but you need to specify an additional parameter (`miniBatchFraction`). `miniBatchFraction` takes a value between 0 and 1. If it's equal to 1 (which is the default), a mini-batch SGD becomes a BGD because the whole data set is considered in each step.

Parameters for mini-batch SGD can be chosen similar to how we did it previously, only now there is one more parameter to be configured. If a step size parameter worked on BGD, that does not mean it will work on mini-batch SGD, so the parameter‘s value has to be chosen in the same way we did it before, or preferably, using the k-fold cross-validation.

A good starting point for the mini-batch fraction parameter is 0.1, but it will probably have to be fine-tuned further. The number of iterations can be chosen so that the data set as a whole is iterated about 100 times in total (and sometimes even less). For example, if the fraction parameter is 0.1, specifying 1,000 iterations guarantees that elements in the data set are taken into account 100 times (on average). For performance reasons, in order to balance computation and communication between nodes in the cluster, the mini-batch size (absolute size, not the fraction parameter) must typically be at least two orders of magnitude larger than the number of machines in the cluster[12].

In our online repository, you'll find the method `iterateLRwSGDBatch`, which is a variation of `iterateLRwSGD` with one additional line:

```
alg.optimizer.setMiniBatchFraction(miniBFraction)
```

The signature of the method is also different as its parameter takes three arrays: besides number of iterations and step sizes, it also takes an array with mini-batch fractions. The method tries all combinations of the three values and prints the results (training and testing MRSE). You can try it out on our data set expanded with feature squares (`trainHPScaled` and `validHPScaled` RDDs). First, to get a feeling for the step size parameter in context of the other two, execute this command:

```
iterateLRwSGDBatch(Array(400, 1000), Array(0.05, 0.09, 0.1, 0.15, 0.2, 0.3,
    0.35, 0.4, 0.5, 1), Array(0.01, 0.1), trainHPScaled, validHPScaled)
```

The results (available online) show that the step size of 0.4 works best. Now let's use that value and see how the algorithm behaves when we change other parameters:

```
iterateLRwSGDBatch(Array(400, 1000, 2000, 3000, 5000, 10000), Array(0.4),
    Array(0.1, 0.2, 0.4, 0.5, 0.6, 0.8), trainHPScaled, validHPScaled)
```

The results (again, available online) show that 2,000 iterations are enough to get the best MRSE of 3.965, which is slightly better even than our previous best MRSE of 3.966

[12] Adding vs. Averaging in Distributed Primal-Dual Optimization, Chenxin Ma et al., www.cs.berkeley.edu/~vsmith/docs/cocoap.pdf

(for Ridge regression). The results also show that with more than 5,000 iterations, we get into overfitting territory. This lowest MRSE was accomplished with mini-batch fraction of 0.5.

If your data set is huge, mini-batch fraction of 0.5 might be too large to get good performance results. You should experiment with lower mini-batch fractions and more iterations. Some experimenting will be needed.

We can conclude that mini-batch SGD can give you the same MRSE as BGD. Because of its performance improvements, you should prefer it to BGD.

7.7.2 LBFGS optimizer

LBFGS is a limited-memory approximation of the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm for minimizing multidimensional functions. Classic BFGS algorithm calculates an approximate inverse of the so-called Hessian matrix, which is a matrix of second-degree derivatives of a function, and keeps an *n x n* matrix in memory, where *n* is the number of dimensions. Limited-memory BFGS only keeps less than 10 last calculated *corrections* and is that more memory efficient, especially for larger numbers of dimensions.

PYTHON LBFGS regression optimizer is not available in Python.

LBFGS can give you really good performance. And it's much simpler to use because, instead of requiring the number of iterations and the step size, its stopping criterion is the *convergence tolerance* parameter. It stops if the MRSE after each iteration changes less than the value of the convergence tolerance parameter. This is a much more natural and simpler criterion.

You also need to give it the maximum number of iterations to run (in case it does not converge), the number of corrections to keep (this should be less than 10, which is the default), and the regularization parameter (it gives you freedom to use L1 or L2 regularization).

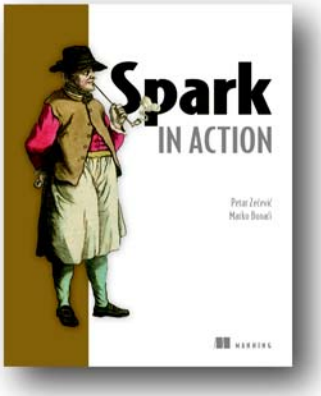
You can find our `iterateLBFGS` method in our online repository and paste it into your Spark Scala shell to try it out like this—but before running it, you might want to set the Breeze library logging level to WARN (the snippet is available online):

```
iterateLBFGS(Array(0.005, 0.007, 0.01, 0.02, 0.03, 0.05, 0.1), 10, 1e-5,
     trainHPScaled, validHPScaled)
0.005, 10 -> 3.8335, 4.0383
0.007, 10 -> 3.8848, 4.0005
0.010, 10 -> 3.9542, 3.9798
0.020, 10 -> 4.1388, 3.9662
0.030, 10 -> 4.2892, 3.9996
0.050, 10 -> 4.5319, 4.0796
0.100, 10 -> 5.0571, 4.3579
```

Now wasn't that fast? It just flew by. And it was simple, too. The only parameter we needed to tweak is the regularization parameter because the other two do not influence the algorithm much, and these defaults can be used safely. Obviously, a regularization parameter of 0.02 gives us the best MRSE of 3.9662. And that is almost the same as our previous best MRSE, which took us great effort to get to.

7.8 Summary

- Supervised learning uses labeled data for training. Unsupervised learning algorithms discover the inner structure of unlabeled data through model fitting.
- Regression and classification differ by the type of target variable: it's continuous (a real number) for regression and categorical (a set of discreet numbers) for classification.
- Before using the data for linear regression, it's a good idea to analyze its distribution and similarities. You should also normalize and scale the data and split it into training and validation data sets.
- A root mean squared error (RMSE) is commonly used for evaluating a linear regression model's performance.
- The learned parameters of a linear regression model can give you insight into how each feature affects the target variable.
- Adding higher-order polynomials to the data set enables you to apply linear regression to nonlinear problems and can yield better results on some data sets.
- Increasing a model's complexity can lead to overfitting. Bias-variance tradeoff says that you can either have high bias or high variance, but not both.
- Ridge and Lasso regularizations help in reducing overfitting for linear regression.
- Mini-batch stochastic gradient descent optimizes performance of the linear regression algorithm.
- The LBFGS optimizer in Spark takes much less time to train and offers great performance.



Working with big data can be complex and challenging, in part because of the multiple analysis frameworks and tools required. Apache Spark is a big-data-processing framework perfect for analyzing near-real-time streams and discovering historical patterns in batched datasets. But Spark goes much further than do other frameworks. By including machine learning and graph-processing capabilities, it makes many specialized data-processing platforms obsolete. Spark's unified framework and programming model significantly lowers the initial infrastructure investment, and Spark's core abstractions are intuitive for most Scala, Java, and Python developers.

Spark in Action teaches you to use Spark for stream and batch data processing. It starts with an introduction to the Spark architecture and ecosystem followed by a taste of Spark's command line interface. You then discover the most fundamental concepts and abstractions of Spark, particularly resilient distributed datasets (RDDs) and the basic data transformations that RDDs provide. The first part of the book also introduces you to writing Spark applications using the core APIs. Next, you'll learn about different Spark components: how to work with structured data using Spark SQL, how to process near-real-time data with Spark Streaming, how to apply machine learning algorithms with Spark MLlib, how to apply graph algorithms on graph-shaped data using Spark GraphX, and a clear introduction to Spark clustering.

What's inside

- Spark code in Java, Scala, and Python
- Spark installation and configuration guided tour
- Scaffolding a new Eclipse Spark Scala project (zero-configuration)
- How to process structured data with Spark SQL
- How to ingest data with Spark Streaming
- Mastering graph computation with GraphX
- Two real-life case studies
- Configuring, monitoring, and tuning Spark
- Spark DevOps with Docker

Readers should be familiar with Java, Scala, or Python. No knowledge of Spark or streaming operations is assumed, but some acquaintance with machine learning is helpful.

Managing datacenter resources with Mesos

The operational infrastructure of a large application can become quite complex. In reactive applications, allocating computing resources efficiently and dynamically is crucial. The following chapter introduces Apache Mesos, an emerging framework designed to manage and allocate hardware resources for executing large data-handling jobs. In it, you'll learn how to use Mesos to run Spark jobs.

Given that it is often impossible to predict exactly how much data a reactive application needs to be able to handle at a given time, there must be a mechanism in place that allows new hardware resources to be automatically provisioned when they're needed and discarded when the job is done. Dynamic hardware resource allocation is crucial for reacting to changes in the load of a reactive application.

Chapter 2 from *Mesos in Action*
by Roger Ignazio

Managing datacenter resources with Mesos

This chapter covers

- Introducing Mesos with a real-world example
- Comparing standalone and general-purpose clusters
- Launching a Spark job on a Mesos cluster
- Exploring a framework's interaction with Mesos

The previous chapter introduced the Apache Mesos project, how it works, and how it compares to the architecture of a traditional datacenter. This chapter explores the benefits of Mesos by applying a real-world scenario: demonstrating multiple applications using Mesos cluster resources. The chapter demonstrates Apache Spark, a popular data-processing framework.

If you're not familiar with Spark, don't worry: the following sections use Spark as a demonstration of how Mesos distributes workloads and shares resources among multiple applications. I use Spark as an example to teach you about resource sharing and workload scheduling on a general-purpose Mesos cluster, and how Mesos compares to statically partitioned clusters within a datacenter. You'll also get a brief introduction to the Mesos and Spark web interfaces, and, who knows, maybe you'll even learn a thing or two about Spark in the process. Let's get started.

2.1 A brief introduction to Spark

To quote the project's website, "Apache Spark is a fast and general engine for large-scale data processing." It lives in the "big data" space along with other popular projects, such as Hadoop, and is often used for data science and analytics. In many cases, Spark performs tasks faster and more efficiently than Hadoop's MapReduce, both in memory and on disk.

Spark also provides APIs for several popular programming languages, including Python, Scala, and Java, and supports streaming workloads, interactive queries, and machine-learning libraries, in addition to MapReduce-like batch processing.

A brief history of Spark

In 2009, Matei Zaharia began development on Spark in the AMPLab at the University of California, Berkeley, the same organization that supported the development of Mesos. In fact, Matei is also one of the co-creators of Mesos.

After being open sourced in 2010, Spark was donated to the Apache Software Foundation and entered the Apache Incubator in 2013. It graduated to become a top-level project in 2014.

Databricks, a company co-founded by Matei in 2013, seeks to commercialize on Spark's successes and help clients with big data problems. Databricks remains a large contributor to the open source Spark project.

At the most basic level, Spark requires a cluster manager for distributing work, and access to a Hadoop-compatible data source. Out of the box, Spark includes support for several cluster managers:

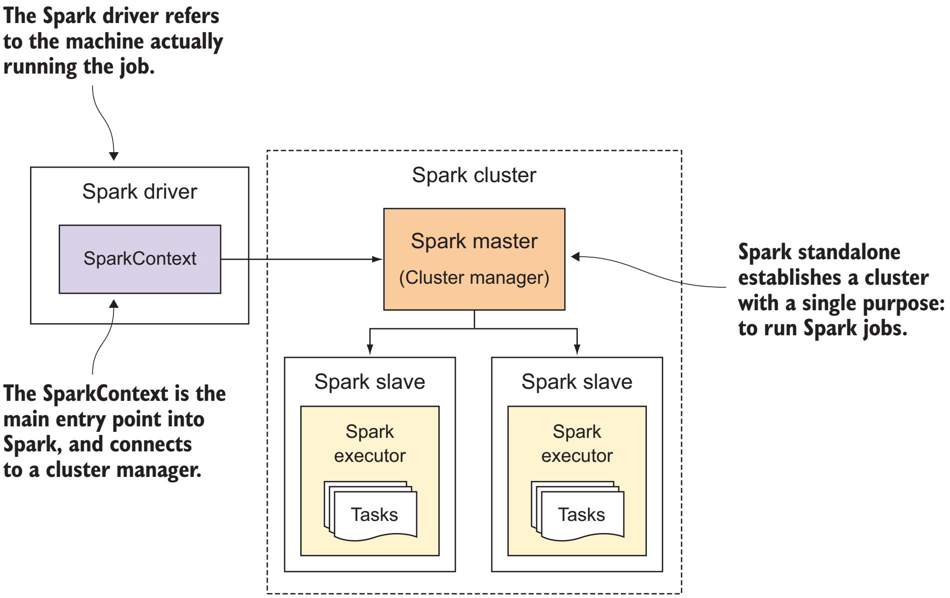
- Spark standalone
- Mesos
- Hadoop YARN
- Pseudo-distributed (running locally on your laptop or workstation)

Although it's possible to run Spark locally and use the CPU cores on your laptop or workstation, that's useful only for development purposes: the number of CPU cores limits the number of executors. When you set up a production Spark cluster, you have two options: deploy a standalone, statically partitioned Spark cluster on a predetermined number of servers, or use a cluster manager such as Mesos or YARN to run the Spark job's tasks for you.

To best illustrate what a general-purpose cluster manager such as Mesos can offer, I'll compare and contrast Spark standalone with Mesos in the next few sections.

2.1.1 Spark on a standalone cluster

In figure 2.1, you see that a Spark driver program connects to a cluster manager—the Spark master—that in turn distributes tasks to various worker nodes.



The Spark driver refers to the machine actually running the job.

The SparkContext is the main entry point into Spark, and connects to a cluster manager.

Spark standalone establishes a cluster with a single purpose: to run Spark jobs.

Figure 2.1 Components and architecture for a standalone Spark cluster

In the graphic, the *Spark Driver* refers to the machine running the Spark job, and the *SparkContext* is the main entry point to Spark. The SparkContext is responsible for connecting to a cluster manager and running tasks on the cluster. It's also responsible for creating Spark's distributed data sets. As you can also see, the two worker nodes in the Spark cluster are single-purpose: they are machines dedicated to running Spark tasks, and nothing else.

As you learned in the previous chapter, Mesos provides an excellent means for running multiple applications on a single cluster, and launching multiple tasks on a single worker node. Instead of setting up one or more statically partitioned Spark clusters, you can use Mesos to share cluster resources across multiple applications. Let's see what it looks like to run Spark on Mesos.

2.1.2 Spark on Mesos

Although setting up Spark to use a standalone cluster isn't a problem, consider the needs of multiple teams needing their own Spark clusters, or consider the bigger picture: multiple, statically partitioned clusters in a single datacenter.

If you're deploying these static clusters on physical hardware, you're clearly dedicating a certain amount of capital to that workload—and only that workload—without the possibility of sharing resources. Likewise, if you set up statically partitioned clusters on an Infrastructure as a Service (IaaS) provider like Amazon Web Services (AWS), you might be wasting money due to cloud instances sitting idle. Regardless of whether your workloads are running on premises or in the cloud, fine-grained resource sharing can help increase a system's utilization, and therefore improve datacenter efficiency.

To illustrate this point, let's take a look at figure 2.2. You have two standalone clusters serving two applications: Spark (the data-processing example used up to this point) and Jenkins, a popular, open source continuous integration framework. The use of Jenkins itself isn't particularly important for this example; what's important is that it's *some other* application that needs to run on multiple servers.

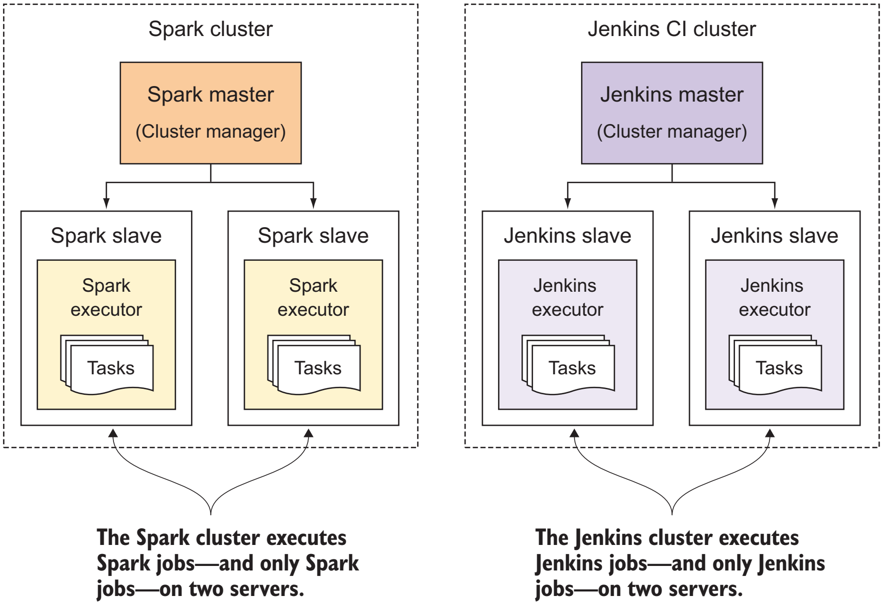
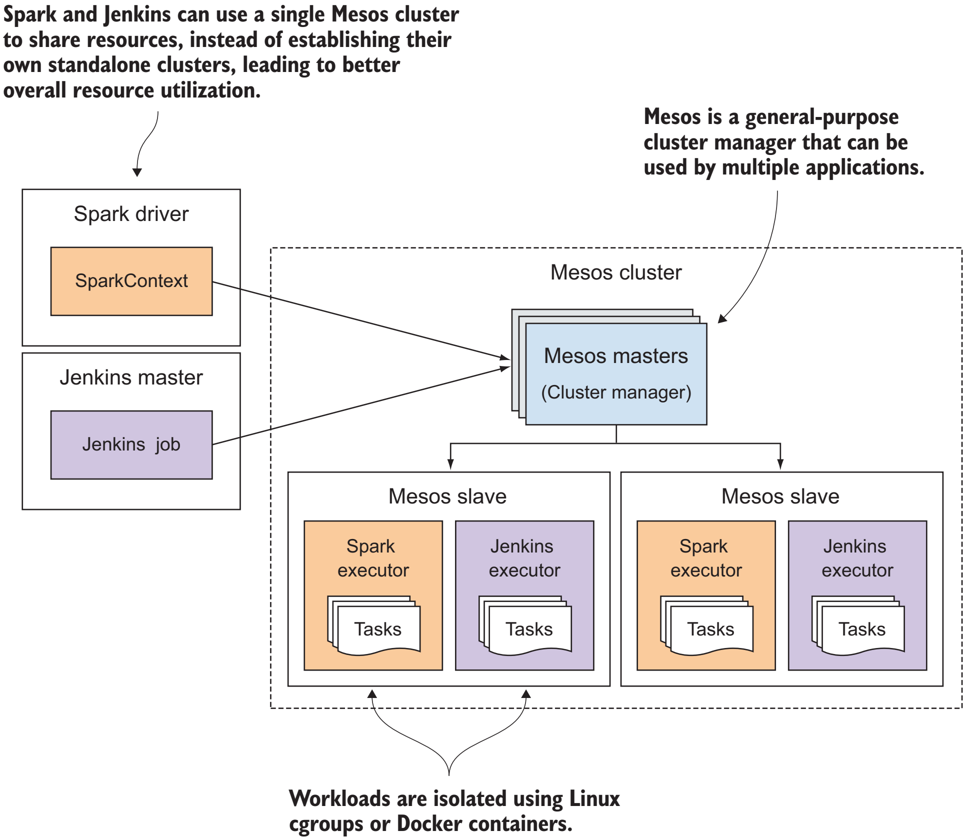


Figure 2.2 Visualizing two statically partitioned, or siloed, clusters

As figure 2.2 illustrates, you now have two statically partitioned clusters: one for Spark and one for Jenkins. Each cluster includes its own cluster manager (Spark master and Jenkins master) and two worker nodes on which to launch tasks or builds. You can also clearly see the static partitions (or silos, if you will) that these two services fall into, and that isn't any way to share compute resources between the two clusters. Chances are that neither of these services is using their computing resources 100% of the time. If the Spark cluster was 50% underpowered, and the Jenkins cluster 50% overpowered, Spark—and the data scientists using the Spark cluster—would benefit by being able to use the resources of three machines instead of just two.

Now let's consider running each of these systems atop a general-purpose cluster manager like Mesos that allows for this sort of fine-grained resource sharing. In figure 2.3, you're able to share compute resources and run multiple workloads on a single Mesos slave by allowing Mesos to isolate each framework's executors using Linux control groups (cgroups). At scale, this will lead to better resource utilization across the many machines within a modern datacenter.

Now that you've taken the time to understand how Spark can use Mesos for its cluster manager, and why adopting a general-purpose cluster manager like Mesos can lead to increased efficiency by sharing compute resources, let's take a look at what it's like to run a Spark job on a Mesos cluster. This will give you a better idea of how Mesos runs tasks before we get into installation and configuration of the cluster in chapter 3.



Spark and Jenkins can use a single Mesos cluster to share resources, instead of establishing their own standalone clusters, leading to better overall resource utilization.

Mesos is a general-purpose cluster manager that can be used by multiple applications.

Workloads are isolated using Linux cgroups or Docker containers.

Figure 2.3 Mesos managing cluster resources for two applications

2.2 Running a Spark job on Mesos

The standalone Spark cluster discussed earlier in this chapter follows an architecture just as you might expect with any other distributed system: a master schedules work on one or more worker nodes. Figure 2.3 demonstrated how you could use Mesos to avoid statically partitioning your datacenter into multiple clusters, and instead declare the compute resources your workload requires of a single, general-purpose cluster. Now let's take a look at Mesos in action by demonstrating how it distributes work for a framework like Spark.

> **NOTE** This section is about running Spark in the context of Mesos, not necessarily a primer on Spark itself. Although I show you how to run the example job on a Mesos cluster, you shouldn't expect to learn about using Spark for real-world data-processing jobs in this text. If you're interested in learning more about Spark, please check out the Spark project page at http://spark.apache.org and *Spark in Action* by Petar Zečević and Marko Bonaći (Manning, 2016).

2.2.1 Finding prime numbers in a set

To demonstrate how Spark connects to a Mesos cluster, accepts resource offers, launches executors, and executes tasks, I'll demonstrate a simple job in Spark. The job will create a data set of integers between 1 and 100,000,000, and then use Spark to determine which integers in the set are prime numbers (numbers that are not equal to 1 and are divisible only by 1 and themselves).

Instead of setting up a standalone Spark cluster for this job, Spark will use Mesos as a cluster manager for scheduling and distributing the individual tasks to available compute resources in the cluster. But before you get into running Spark on a Mesos cluster, let's discuss the order of events that takes place when a framework interacts with a Mesos cluster. Figure 2.4 maps out Spark registering as a Mesos framework, accepting resource offers from the Mesos master, and finally, launching tasks on a Mesos slave.

Several things are happening in this figure, and you'll see a breakdown of what's happening a little later in the chapter. For now, it's important to understand the following:

1. The SparkContext connects to ZooKeeper to determine the leading Mesos master.
2. The SparkContext registers with the leading Mesos master as a new framework.
3. The SparkContext receives resource offers from the leading Mesos master, with which it can launch tasks to perform its data-processing workloads.

Having learned the events—and the order they occur—from figure 2.4, let's take the time now to launch the Spark job and observe real output from the cluster. After you have a Mesos cluster up and running (which you'll learn about in the next chapter), feel free to install Spark and run the example.

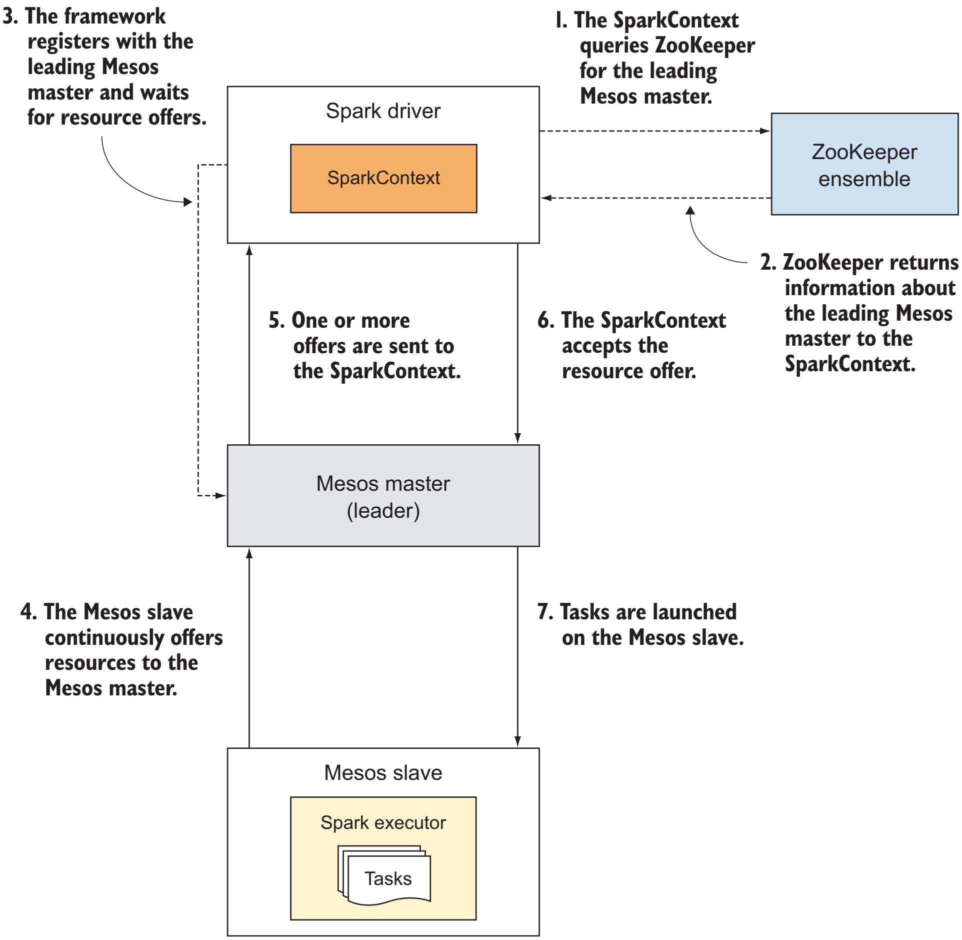

3. The framework registers with the leading Mesos master and waits for resource offers.
1. The SparkContext queries ZooKeeper for the leading Mesos master.
Spark driver
SparkContext
ZooKeeper ensemble
2. ZooKeeper returns information about the leading Mesos master to the SparkContext.
5. One or more offers are sent to the SparkContext.
6. The SparkContext accepts the resource offer.
Mesos master (leader)
4. The Mesos slave continuously offers resources to the Mesos master.
7. Tasks are launched on the Mesos slave.
Mesos slave
Spark executor
Tasks

Figure 2.4 Events that occur when Spark runs tasks on a Mesos cluster

TIP Installation instructions for Spark on Mesos are available on the Spark website, http://spark.apache.org/docs/latest/running-on-mesos.html.

2.2.2 Getting and packaging up the code

I've included the example code for this Spark job with the book's supplementary materials, available on GitHub and on manning.com. The easiest way to get the example code is to clone the repository by using Git:

```
$ git clone https://github.com/rji/mesos-in-action-code-samples
$ cd mesos-in-action-code-samples/chapter02/spark-primes-example
```

Next, you need to package the job and its dependencies into a single Java Archive (JAR) file that can be used with the `spark-submit` command-line tool. Because this particular example is written in the Scala programming language, you'll need to ensure that a recent Java Development Kit (JDK) and Scala are both present on the system you're using to submit the job. I'll refer to this as the *gateway* machine.

After those prerequisites are met, package up the example by using sbt, a build tool for Scala that's similar to Maven or Ant in the Java community. If sbt isn't already installed on your system, you can find installation instructions for Linux, Mac OS X, and Windows at www.scala-sbt.org/release/tutorial.

Proceed to package the example by running the following command:

```
$ sbt package
```

After packaging has completed, you're ready to submit the job to a Mesos cluster. Although I won't cover the Mesos installation and configuration process until the next chapter, I thought it might be beneficial for you to understand how the cluster works before we dive in to deploying it.

2.2.3 Submitting the job

Having already packaged the example code into a simple JAR file, let's go ahead and submit the job. The following example assumes that Spark is installed at /opt/spark:

```
/opt/spark/bin/spark-submit --class com.manning.mesosinaction.PrimesExample
➥ target/scala-2.10/spark-primes-example_2.10-0.1.0-SNAPSHOT.jar
➥ 100000000
```

This job should only take a few minutes to complete.

2.2.4 Observing the output

After submitting the job by using the `spark-submit` command, you observe a decent amount of output on your console; by default, Spark is logging to the console with `INFO`-level verbosity. The following listing includes some of the more important log messages, and I'll explain what they mean in the context of Mesos.

Listing 2.1 Spark job output when running on a Mesos cluster

```
15/04/12 22:35:56 INFO Utils: Successfully started service 'sparkDriver'
    on port 45957.

15/04/12 22:35:56 INFO Utils: Successfully started service
    'HTTP file server' on port 49444.

15/04/12 22:35:56 INFO SparkUI: Started SparkUI at
    http://10.132.171.224:4040
```

Spark queries ZooKeeper for the leading Mesos master.

```
I0412 22:35:57.401646  8991 sched.cpp:157] Version: 0.22.2

2015-04-12 22:35:57,415:8901(0x7f8ed93eb700):ZOO_INFO@check_events@1703:
    initiated connection to server [10.132.171.224:2181]

I0412 22:35:57.418431  8993 detector.cpp:452] A new leading master
    (UPID=master@10.132.171.224:5050) is detected

I0412 22:35:57.418504  8993 sched.cpp:254] New master detected at
    master@10.132.171.224:5050
```

Our SparkContext registers itself as a Mesos framework.

```
I0412 22:35:57.420454  8993 sched.cpp:448] Framework registered with
    20150412-214000-3769336842-5050-2832-0005

15/04/12 22:35:57 INFO MesosSchedulerBackend: Registered as framework ID
    20150412-214000-3769336842-5050-2832-0005
```

The Spark driver executes tasks on the Mesos cluster.

```
15/04/12 22:35:57 INFO SparkContext: Starting job: collect at
    PrimesExample.scala:22

15/04/12 22:39:34 INFO SparkContext: Job finished: collect at
    PrimesExample.scala:22, took 217.099354417 s
```

Output omitted for brevity

```
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79

99999839 99999847 99999931 99999941 99999959 99999971 99999989

15/04/12 22:40:10 INFO SparkUI: Stopped Spark web UI at http://
    10.132.171.224:4040
```

The Spark-Context shuts down its scheduler.

```
15/04/12 22:40:10 INFO DAGScheduler: Stopping DAGScheduler
```

The Spark framework is unregistered from the Mesos master.

```
I0412 22:40:10.902202  8931 sched.cpp:1589] Asked to stop the driver

I0412 22:40:10.902323  8997 sched.cpp:831] Stopping framework
    '20150412-214000-3769336842-5050-2832-0005'

15/04/12 22:40:10 INFO MesosSchedulerBackend: driver.run() returned
    with code DRIVER_STOPPED

15/04/12 22:40:11 INFO SparkContext: Successfully stopped SparkContext
```

Although a lot more activity exists in the Spark logs, and you probably don't need INFO-level verbosity on a regular basis, the lines selected for this listing serve as a good example to understand how Spark is using Mesos to handle its workload, or at least the order of events when launching a Spark workload on Mesos, just as we visualized previously in figure 2.4.

Now that you understand this order of events for a real-world workload, let's take a look at more ways you can observe the output and status of the frameworks and tasks running on a Mesos cluster.

2.3 Exploring further

Having just submitted and run a Spark job on a Mesos cluster—and observed the output from the job on the console—perhaps it's a good time to start introducing you to the various happenings under the hood. In this section, you'll take a quick look at the Mesos and Spark web interfaces so you can observe the work being performed on your cluster in real time. Although you won't install and configure Mesos until the next chapter, the next few sections serve as a starting point for topics that you'll learn in part 2 of this book.

2.3.1 Mesos UI

The Mesos masters provide a web interface for viewing the status of the cluster and the work being performed. This web interface provides information on the cluster, including the following:

- Overview of all tasks and their current status
- Registered frameworks and associated tasks
- Mesos slaves, their resources, and the tasks they're currently executing
- Outstanding resource offers (offers that haven't yet been accepted or rejected)

In figure 2.5, you can see that the Spark Primes Example framework is registered to the cluster, is currently consuming 6 CPUs and 2.6 GB memory, and is running several tasks. For now, don't worry about every feature in the web interface; you're concerned only with observing how the cluster responds to a single Spark job running on it, in an attempt to familiarize you with the features available in Mesos. Chapter 5 revisits the web interface in greater depth.

Clicking the Sandbox link for any of those tasks allows you to drill down into the files and log output present in the Mesos *sandbox*, or working directory, for individual tasks. In figure 2.6, the sandbox contains your Spark job's JAR file and files that have captured the console output from stdout and stderr.

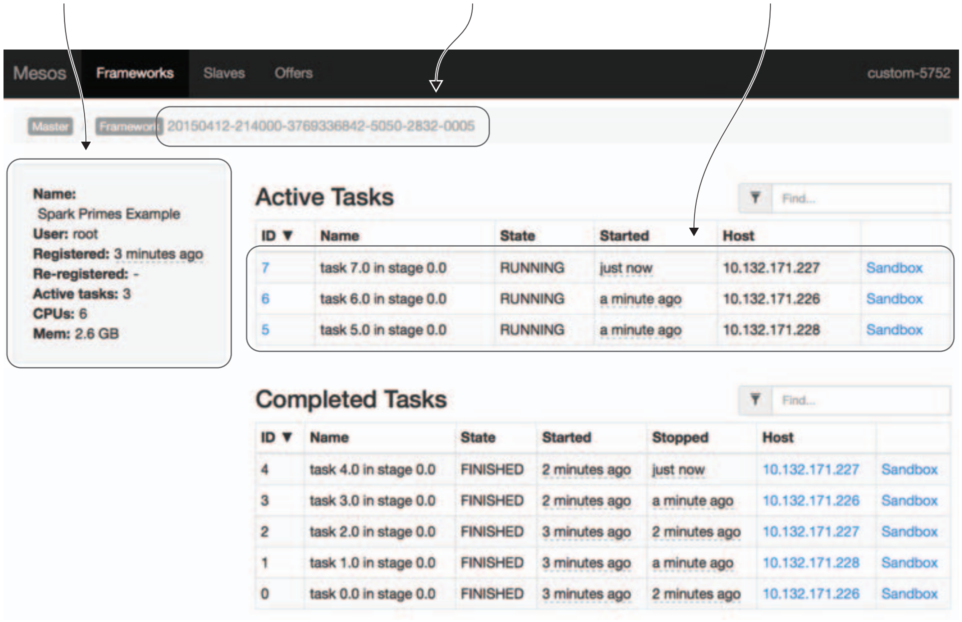
2.3.2 Spark UI

In addition to the web interface provided by Mesos, Spark launches its own web interface for monitoring the progress of your Spark jobs. You saw evidence of this in the Spark job's output in listing 2.1. Although accessing this interface isn't necessary for the proper functioning of the Mesos cluster, it does provide a nicer, cluster manager–agnostic view of the progress of a particular Spark job.

Information about the framework, including active tasks and resources consumed

Unique framework ID generated by the Mesos master

Active tasks for the framework, and their location in the cluster



20150412-214000-3769336842-5050-2832-0005

Name:
Spark Primes Example
User: root
Registered: 3 minutes ago
Re-registered: -
Active tasks: 3
CPUs: 6
Mem: 2.6 GB

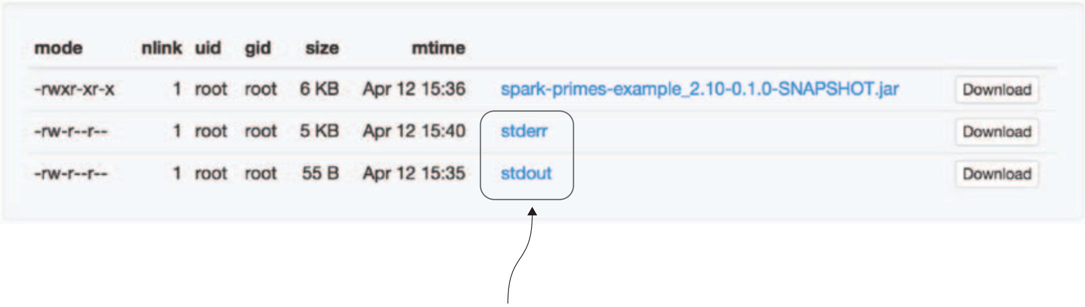
Active Tasks

ID ▼	Name	State	Started	Host	
7	task 7.0 in stage 0.0	RUNNING	just now	10.132.171.227	Sandbox
6	task 6.0 in stage 0.0	RUNNING	a minute ago	10.132.171.226	Sandbox
5	task 5.0 in stage 0.0	RUNNING	a minute ago	10.132.171.228	Sandbox

Completed Tasks

ID ▼	Name	State	Started	Stopped	Host	
4	task 4.0 in stage 0.0	FINISHED	2 minutes ago	just now	10.132.171.227	Sandbox
3	task 3.0 in stage 0.0	FINISHED	2 minutes ago	a minute ago	10.132.171.226	Sandbox
2	task 2.0 in stage 0.0	FINISHED	3 minutes ago	2 minutes ago	10.132.171.227	Sandbox
1	task 1.0 in stage 0.0	FINISHED	3 minutes ago	a minute ago	10.132.171.228	Sandbox
0	task 0.0 in stage 0.0	FINISHED	3 minutes ago	2 minutes ago	10.132.171.226	Sandbox

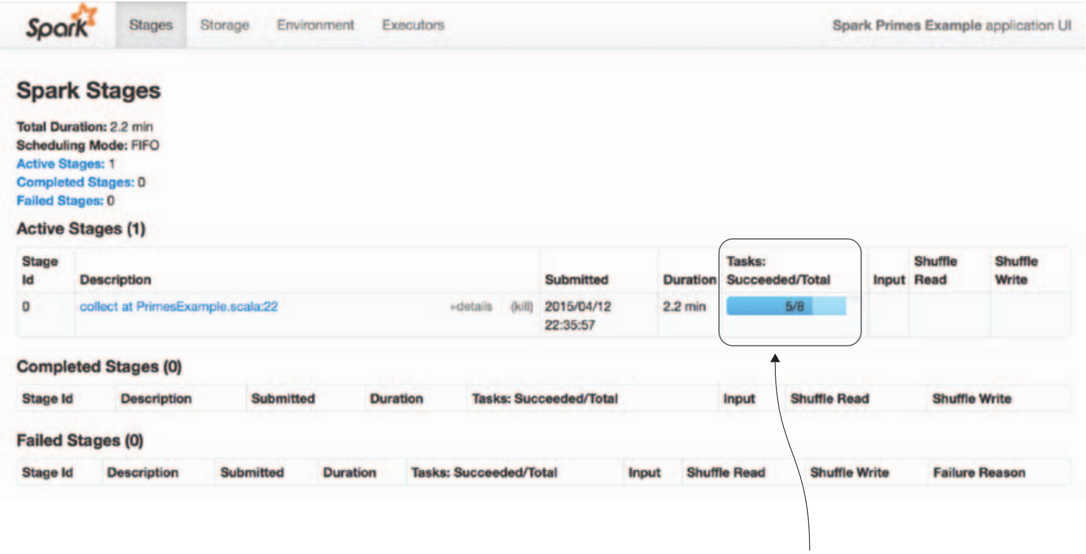
Figure 2.5 The Spark Primes Example framework is consuming cluster resources and running tasks.



mode	nlink	uid	gid	size	mtime		
-rwxr-xr-x	1	root	root	6 KB	Apr 12 15:36	spark-primes-example_2.10-0.1.0-SNAPSHOT.jar	Download
-rw-r--r--	1	root	root	5 KB	Apr 12 15:40	stderr	Download
-rw-r--r--	1	root	root	55 B	Apr 12 15:35	stdout	Download

Console output—stdout and stderr—are captured in text files in a task's sandbox

Figure 2.6 Files within a Mesos sandbox for a single Spark task



Spark Stages

Total Duration: 2.2 min
Scheduling Mode: FIFO
Active Stages: 1
Completed Stages: 0
Failed Stages: 0

Active Stages (1)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write
0	collect at PrimesExample.scala:22 +details (kill)	2015/04/12 22:35:57	2.2 min	5/8			

Completed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write

Failed Stages (0)

Stage Id	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Shuffle Read	Shuffle Write	Failure Reason

Spark splits its workload into individual tasks. The user can monitor the progress of Spark's tasks here, without diving into the Mesos UI.

Figure 2.7 The Spark web interface shows the progress of the Spark Primes Example job

In figure 2.7, you can see that the job's only stage, "collect at PrimesExample.scala:22," is currently running on the cluster and has completed 5/8 of its tasks.

Regardless of whether this job was running on Mesos or on another cluster manager, the tasks would be distributed in a similar fashion. The difference, however, is that Mesos allows you to run multiple frameworks—and their tasks—alongside each other in an isolated manner.

2.4 Summary

Although this example was about how Mesos schedules and distributes work on a cluster in the context of a Spark job, hopefully it was beneficial for you to see an example Mesos workload—and Mesos in action—all in a single chapter. By doing so, you now know what you can expect after deploying Mesos. To recap, you learned the following from this chapter:

- Distributed frameworks such as Apache Spark and Jenkins CI can use Mesos as their cluster manager, simultaneously.
- Mesos' fine-grained resource sharing can lead to higher resource utilization across the datacenter.

- A Spark job is composed of a number of individual tasks, or units of work, that are distributed on the Mesos cluster.
- The Mesos web interface provides a glimpse into the current state of the cluster.

Modern datacenters are complex environments, and when you throw Docker and other container-based systems into the mix, there's a great need to simplify. Mesos is an open source cluster management platform that transforms the whole datacenter into a single pool of compute, memory, and storage resources that you can allocate, automate, and scale as if you're working with a single supercomputer.

Mesos in Action introduces readers to the Apache Mesos cluster manager and the concept of application-centric infrastructure. Filled with helpful figures and hands-on instructions, this book guides you from your first steps creating a highly-available Mesos cluster through deploying applications in production and writing native Mesos frameworks.

What's inside

- Spinning up your first Mesos cluster
- Scheduling, resource administration, and logging
- Deploying containerized applications with Marathon, Chronos, and Aurora
- Writing Mesos frameworks using Python

Readers need to be familiar with the core ideas of datacenter administration and need a basic knowledge of Python or a similar programming language.

Index