BYRON ELLIS

REAL-TIME ANALYTICS

TECHNIQUES TO ANALYZE AND VISUALIZE STREAMING DATA

WILEY

Table of Contents

Guide

List of Illustrations

List of Tables

Chapter 1
Introduction to Streaming Data

It seems like the world moves at a faster pace every day. People and places become more connected, and people and organizations try to react at an ever-increasing pace. Reaching the limits of a human's ability to respond, tools are built to process the vast amounts of data available to decision makers, analyze it, present it, and, in some cases, respond to events as they happen.

The collection and processing of this data has a number of application areas, some of which are discussed in the next section. These applications, which are discussed later in this chapter, require an infrastructure and method of analysis specific to streaming data. Fortunately, like batch processing before it, the state of the art of streaming infrastructure is focused on using commodity hardware and software to build its systems rather than the specialized systems required for real-time analysis prior to the Internet era. This, combined with flexible cloud-based environment, puts the implementation of a real-time system within the reach of nearly any organization. These commodity systems allow organizations to analyze their data in real time and scale that infrastructure to meet future needs as the organization grows and changes over time.

The goal of this book is to allow a fairly broad range of potential users and implementers in an organization to gain comfort with the complete stack of applications. When real-time projects reach a certain point, they should be agile and adaptable systems that can be easily modified, which requires that the users have a fair understanding of the stack as a whole in addition to their own areas of focus. “Real time” applies as much to the development of new analyses as it does to the data itself. Any number of well-meaning projects have failed because they took so long to implement that the people who requested the project have either moved on to other things or simply forgotten why they wanted the data in the first place. By making the projects agile and incremental, this can be avoided as much as possible.

This chapter is divided into sections that cover three topics. The first section, “Sources of Streaming Data,” is some of the common sources and applications of streaming data. They are arranged more or less chronologically and provide some background on the origin of streaming data infrastructures. Although this is historically interesting, many of the tools and frameworks presented were developed to solve problems in these spaces, and their design reflects some of the challenges unique to the space in which they were born. Kafka, a data motion tool covered in Chapter 4, “Flow Management for Streaming Analysis,” for example, was developed as a web applications tool, whereas Storm, a processing framework covered in Chapter 5, “Processing Streaming Data,” was developed primarily at Twitter for handling social media data.

The second section, “Why Streaming Data is Different,” covers three of the important aspects of streaming data: continuous data delivery, loosely structured data, and high-cardinality datasets. The first, of course, defines a system to be a real-time streaming data environment in the first place. The other two, though not entirely unique, present a unique challenge to the designer of a streaming data application. All three combine to form the essential streaming data environment.

The third section, “Infrastructures and Algorithms,” briefly touches on the significance of how infrastructures and algorithms are used with streaming data.

Sources of Streaming Data

There are a variety of sources of streaming data. This section introduces some of the major categories of data. Although there are always more and more data sources being made available, as well as many proprietary data sources, the categories discussed in this section are some of the application areas that have made streaming data interesting. The ordering of the application areas is primarily chronological, and much of the software discussed in this book derives from solving problems in each of these specific application areas.

The data motion systems presented in this book got their start handling data for website analytics and online advertising at places like LinkedIn, Yahoo!, and Facebook. The processing systems were designed to meet the challenges of processing social media data from Twitter and social networks like LinkedIn.

Google, whose business is largely related to online advertising, makes heavy use of the advanced algorithmic approaches similar to those presented in Chapter 11. Google seems to be especially interested in a technique called deep learning, which makes use of very large-scale neural networks to learn complicated patterns.

These systems are even enabling entirely new areas of data collection and analysis by making the Internet of Things and other highly distributed data collection efforts economically feasible. It is hoped that outlining some of the previous application areas provides some inspiration for as-of-yet-unforeseen applications of these technologies.

Operational Monitoring

Operational monitoring of physical systems was the original application of streaming data. Originally, this would have been implemented using specialized hardware and software (or even analog and mechanical systems in the pre-computer era). The most common use case today of operational monitoring is tracking the performance of the physical systems that power the Internet.

These datacenters house thousands—possibly even tens of thousands—of discrete computer systems. All of these systems continuously record data about their physical state from the temperature of the processor, to the speed of the fan and the voltage draw of their power supplies. They also record information about the state of their disk drives and fundamental metrics of their operation, such as processor load, network activity, and storage access times.

To make the monitoring of all of these systems possible and to identify problems, this data is collected and aggregated in real time through a variety of mechanisms. The first systems tended to be specialized ad hoc mechanisms, but when these sorts of techniques started applying to other areas, they started using the same collection systems as other data collection mechanisms.

Web Analytics

The introduction of the commercial web, through e-commerce and online advertising, led to the need to track activity on a website. Like the circulation numbers of a newspaper, the number of unique visitors who see a website in a day is important information. For e-commerce sites, the data is less about the number of visitors as it is the various products they browse and the correlations between them.

To analyze this data, a number of specialized log-processing tools were introduced and marketed.

With the rise of Big Data and tools like Hadoop, much of the web analytics infrastructure shifted to these large batch-based systems. They were used to implement recommendation systems and other analysis. It also became clear that it was possible to conduct experiments on the structure of websites to see how they affected various metrics of interest. This is called A/B testing because—in the same way an optometrist tries to determine the best prescription—two choices are pitted against each other to determine which is best. These tests were mostly conducted sequentially, but this has a number of problems, not the least of which is the amount of time needed to conduct the study.

As more and more organizations mined their website data, the need to reduce the time in the feedback loop and to collect data on a more continual basis became more important. Using the tools of the system-monitoring community, it became possible to also collect this data in real time and perform things like A/B tests in parallel rather than in sequence. As the number of dimensions being measured and the need for appropriate auditing (due to the metrics being used for billing) increased, the analytics community developed much of the streaming infrastructure found in this book to safely move data from their web servers spread around the world to processing and billing systems.

This sort of data still accounts for a vast source of information from a variety of sources, although it is usually contained within an organization rather than made publicly available. Applications range from simple aggregation for billing to the real-time optimization of product recommendations based on current browsing history (or viewing history, in the case of a company like Netflix).

Online Advertising

A major user and generator of real-time data is online advertising. The original forms of online advertising were similar to their print counterparts with "buys" set up months in advance. With the rise of the advertising exchange and real-time bidding infrastructure, the advertising market has become much more fluid for a large and growing portion of traffic.

For these applications, the money being spent in different environments and on different sites is being managed on a per-minute basis in much the same way as the stock market. In addition, these buys are often being managed to some sort of metric, such as the number of purchases (called a conversion) or even the simpler metric of the number of clicks on an ad. When a visitor arrives at a website via a modern advertising exchange, a call is made to a number of bidding agencies (perhaps 30 or 40 at a time), who place bids on the page view in real time. An auction is run, and the advertisement from the winning party is displayed. This usually happens while the rest of the page is loading; the elapsed time is less than about 100 milliseconds. If the page contains several advertisements, as many of them do, an auction is often being run for all of them, sometimes on several different exchanges.

All the parties in this process—the exchange, the bidding agent, the advertiser, and the publisher—are collecting this data in real time for various purposes. For the exchange, this data is a key part of the billing process as well as important for real-time feedback mechanisms that are used for a variety of purposes. Examples include monitoring the exchange for fraudulent traffic and other risk management activities, such as throttling the access to impressions to various parties.

Advertisers, publishers, and bidding agents on both sides of the exchange are also collecting the data in real time. Their goal is the management and optimization of the campaigns they are currently running. From selecting the bid (in the case of the advertiser) or the "reserve" price (in the case of the publisher), to deciding which exchange offers the best prices for a particular type of traffic, the data is all being managed on a moment-to-moment basis.

A good-sized advertising campaign or a large website could easily see page views in the tens or hundreds of millions. Including other events such as clicks and conversions could easily double that. A bidding agent is usually acting on the part of many different advertisers or publishers at once and will often be collecting on the order of hundreds of millions to billions of events per day. Even a medium-sized exchange, sitting as a central party, can have billions of events per day. All of this data is being collected, moved, analyzed, and stored as it happens.

Social Media

Another newer source of massive collections of data are social media sources, especially public ones such as Twitter. As of the middle of 2013, Twitter reported that it collected around 500 million tweets per day with spikes up to around 150,000 tweets per second. That number has surely grown since then.

This data is collected and disseminated in real time, making it an important source of information for news outlets and other consumers around the world. In 2011, Twitter users in New York City received information about an earthquake outside of Washington, D.C. about 30 seconds before the tremors struck New York itself.

Combined with other sources like Facebook, Foursquare, and upcoming communications platforms, this data is extremely large and varied. The data from applications like web analytics and online advertising, although highly dimensional, are usually fairly well structured. The dimensions, such as money spent or physical location, are fairly well understood quantitative values.

In social media, however, the data is usually highly unstructured, at least as data analysts understand the term. It is usually some form of "natural language" data that must be parsed, processed, and somehow understood by automated systems. This makes social media data incredibly rich, but incredibly challenging for the real-time data sources to process.

Mobile Data and the Internet of Things

One of the most exciting new sources of data was introduced to the world in 2007 in the form of Apple's iPhone. Cellular data-enabled computers had been available for at least a decade, and devices like Blackberries had put data in the hands of business users, but these devices were still essentially specialist tools and were managed as such.

The iPhone, Android phones, and other smartphones that followed made cellular data a consumer technology with the accompanying economies of scale that goes hand in hand with a massive increase in user base. It also put a general-purpose computer in the pocket of a large population. Smartphones have the ability not only to report back to an online service, but also to communicate with other nearby objects using technologies like Bluetooth LE.

Technologies like so-called "wearables," which make it possible to measure the physical world the same way the virtual world has been measured for the last two decades, have taken advantage of this new infrastructure. The applications range from the silly to the useful to the creepy. For example, a wristband that measures sleep activity could trigger an automated coffee maker when the user gets a poor night's sleep and needs to be alert the next day. The smell of coffee brewing could even be used as an alarm. The communication between these systems no longer needs to be direct or specialized, as envisioned in the various "smart home" demonstration projects during the past 50 years. These tools are possible today using tools like If This Then That (IFTTT) and other publicly available

systems built on infrastructure similar to those in this book.

On a more serious note, important biometric data can be measured in real time by remote facilities in a way that has previously been available only when using highly specialized and expensive equipment, which has limited its application to high-stress environments like space exploration. Now this data can be collected for an individual over a long period of time (this is known in statistics as longitudinal data) and pooled with other users' data to provide a more complete picture of human biometric norms. Instead of taking a blood pressure test once a year in a cold room wearing a paper dress, a person's blood pressure might be tracked over time with the goal of "heading off problems at the pass."

Outside of health, there has long been the idea of "smart dust"—large collections of inexpensive sensors that can be distributed into an area of the world and remotely queried to collect interesting data. The limitation of these devices has largely been the expense required to manufacture relatively specialized pieces of equipment. This has been solved by the commodification of data collection hardware and software (such as the smartphone) and is now known as the Internet of Things. Not only will people continually monitor themselves, objects will continually monitor themselves as well. This has a variety of potential applications, such as traffic management within cities to making agriculture more efficient through better monitoring of soil conditions.

The important piece is that this information can be streaming through commodity systems rather than hardware and software specialized for collection. These commodity systems already exist, and the software required to analyze the data is already available. All that remains to be developed are the novel applications for collecting the data.

Why Streaming Data Is Different

There are a number of aspects to streaming data that set it apart from other kinds of data. The three most important, covered in this section, are the "always-on" nature of the data, the loose and changing data structure, and the challenges presented by high-cardinality dimensions. All three play a major role in decisions made in the design and implementation of the various streaming frameworks presented in this book. These features of streaming data particularly influence the data processing frameworks presented in Chapter 5. They are also reflected in the design decisions of the data motion tools, which consciously choose not to impose a data format on information passing through their system to allow maximum flexibility. The remainder of this section covers each of these in more depth to provide some context before diving into Chapter 2, which covers the components and requirements of a streaming architecture.

Always On, Always Flowing

This first is somewhat obvious: streaming data streams. The data is always available and new data is always being generated. This has a few effects on the design of any collection and analysis system. First, the collection itself needs to be very robust. Downtime for the primary collection system means that data is permanently lost. This is an important thing to remember when designing an edge collector, and it is discussed in more detail in Chapter 2.

Second, the fact that the data is always flowing means that the system needs to be able to keep up with the data. If 2 minutes are required to process 1 minute of data, the system will not be real time for very long. Eventually, the problem will be so bad that some data will have to be dropped to allow the system to catch up. In practice it is not enough to have a system that can merely "keep up" with data in real time. It needs to be able to process data far more quickly than real time. For reasons that are either intentional, such as a planned downtime, or due to catastrophic failures, such as network outages, the system either whole or in part will go down.

Failing to plan for this inevitability and having a system that can only process at the same speed as events happen means that the system is now delayed by the amount of data stored at the collectors while the system was down. A system that can process 1 hour of data in 1 minute, on the other hand, can catch up fairly quickly with little need for intervention. A mature environment that has good horizontal scalability—a concept also discussed in Chapter 2—can even implement auto-scaling. In this setting, as the delay increases, more processing power is temporarily added to bring the delay back into acceptable limits.

On the algorithmic side, this always-flowing feature of streaming data is a bit of a double-edged sword. On the positive side, there is rarely a situation where there is not enough data. If more data is required for an analysis, simply wait for enough data to become available. It may require a long wait, but other analyses can be conducted in the meantime that can provide early indicators of how the later analysis might proceed.

On the downside, much of the statistical tooling that has been developed over the last 80 or so years is focused on the discrete experiment. Many of the standard approaches to analysis are not necessarily well suited to the data when it is streaming. For example, the concept of "statistical significance" becomes an odd sort of concept when used in a streaming context. Many see it as some sort of "stopping rule" for collecting data, but it does not actually work like that. The p-value statistic used to make the significance call is itself a random value and may dip below the critical value

(usually 0.05) even though, when the next value is observed, it would result in a p-value above 0.05.

This does not mean that statistical techniques cannot and should not be used—quite the opposite. They still represent the best tools available for the analysis of noisy data. It is simply that care should be taken when performing the analysis as the prevailing dogma is mostly focused on discrete experiments.

Loosely Structured

Streaming data is often loosely structured compared to many other datasets. There are several reasons this happens, and although this loose structure is not unique to streaming data, it seems to be more common in the streaming settings than in other situations.

Part of the reason seems to be the type of data that is interesting in the streaming setting. Streaming data comes from a variety of sources. Although some of these sources are rigidly structured, many of them are carrying an arbitrary data payload. Social media streams, in particular, will be carrying data about everything from world events to the best slice of pizza to be found in Brooklyn on a Sunday night. To make things more difficult, the data is encoded as human language.

Another reason is that there is a "kitchen sink" mentality to streaming data projects. Most of the projects are fairly young and exploring unknown territory, so it makes sense to toss as many different dimensions into the data as possible. This is likely to change over time, so the decision is also made to use a format that can be easily modified, such as JavaScript Object Notation (JSON). The general paradigm is to collect as much data as possible in the event that it is actually interesting.

Finally, the real-time nature of the data collection also means that various dimensions may or may not be available at any given time. For example, a service that converts IP addresses to a geographical location may be temporarily unavailable. For a batch system this does not present a problem; the analysis can always be redone later when the service is once more available. The streaming system, on the other hand, must be able to deal with changes in the available dimensions and do the best it can.

High-Cardinality Storage

Cardinality refers to the number of unique values a piece of data can take on. Formally, cardinality refers to the size of a set and can be applied to the various dimensions of a dataset as well as the entire dataset itself. This high cardinality often manifests itself in a "long tail" feature of the data. For a given dimension (or combination of dimensions) there is a small set of different states that are quite common, usually accounting for the majority of the observed data, and then a "long tail" of other data states that comprise a fairly small fraction.

This feature is common to both streaming and batch systems, but it is much harder to deal with high cardinality in the streaming setting. In the batch setting it is usually possible to perform multiple passes over the dataset. A first pass over the data is often used to identify dimensions with high cardinality and compute the states that make up most of the data. These common states can be treated individually, and the remaining state is combined into a single "other" state that can usually be ignored.

In the streaming setting, the data can usually be processed a single time. If the common cases are known ahead of time, this can be included in the processing step. The long tail can also be combined into the "other" state, and the analysis can proceed as it does in the batch case. If a batch study has

already been performed on an earlier dataset, it can be used to inform the streaming analysis. However, it is often not known if the common states for the current data will be the common states for future data. In fact, changes in the mix of states might actually be the metric of interest. More commonly, there is no previous data to perform an analysis upon. In this case, the streaming system must attempt to deal with the data at its natural cardinality.

This is difficult both in terms of processing and in terms of storage. Doing anything with a large set necessarily takes time to process anything that involves a large number of different states. It also requires a linear amount of space to store information about each different state and, unlike batch processing, storage space is much more restricted than in the batch setting because it must generally use very fast main memory storage instead of the much slower tertiary storage of hard drives. This has been relaxed somewhat with the introduction of high-performance Solid State Drives (SSDs), but they are still orders of magnitude slower than memory access.

As a result, a major topic of research in streaming data is how to deal with high-cardinality data. This book discusses some of the approaches to dealing with the problem. As an active area of research, more solutions are being developed and improved every day.

Infrastructures and Algorithms

The intent of this book is to provide the reader with the ability to implement a streaming data project from start to finish. An algorithm without an infrastructure is, perhaps, an interesting research paper, but not a finished system. An infrastructure without an application is mostly just a waste of resources.

The approach of “build it and they will come” really isn't going to work if you focus solely on an algorithm or an infrastructure. Instead, a tangible system must be built implementing both the algorithm and the infrastructure required to support it. With an example in place, other people will be able to see how the pieces fit together and add their own areas of interest to the infrastructure. One important thing to remember when building the infrastructure (and it bears repeating) is that the goal is to make the infrastructure and algorithms accessible to a variety of users in an organization (or the world). A successful project is one that people use, enhance, and extend.

Conclusion

Ultimately, the rise of web-scale data collection has been about connecting “sensor platforms” for a real-time processing framework. Initially, these sensor platforms were entirely virtual things such as system monitoring agents or the connections between websites and a processing framework for the purposes of advertising. With the rise of ubiquitous Internet connectivity, this has transferred to the physical world to allow collection of data across a wide range of industries at massive scale.

Once data becomes available in real time, it is inevitable that processing should be undertaken in real time as well. Otherwise, what would be the point of real-time collection when bulk collection is probably still less expensive on a per-observation basis? This brings with it a number of unique challenges. Using the mix of infrastructural decisions and computational approaches covered in this book, these challenges can be largely overcome to allow for the real-time processing of today's data as well as the ever-expanding data collection of future systems.

Part I
Streaming A Analytics Architecture

In This Part

Chapter 2
Designing Real-Time Streaming Architectures

By their nature, real-time architectures are layered systems that rely on several loosely coupled systems to achieve their goals. There are a variety of reasons for this structure, from maintaining the high availability of the system in an unreliable world to service requirements and managing the cost structure of the architecture itself.

The remainder of the first section of this book introduces software, frameworks and methods for dealing with the various elements of these architectures. This chapter serves as the blueprint and foundation for the architecture. First, the various components of the architecture are introduced. These usually, but not always, correspond to separate machine instances (physical or virtual). After these components have been introduced, they can be discussed in the context of the primary features of a real-time architecture: high availability, low latency, and horizontal scalability.

This chapter also spends some time discussing the languages used to build real-time architectures. It is assumed that the reader will have some familiarity with the languages used for the examples in this book: Java and JavaScript. Many of the software packages in this book are implemented using the Java Virtual Machine, though not necessarily Java itself. JavaScript, of course, is the lingua franca of the web and is used extensively in the later sections of this book to implement the interfaces to the data.

Finally, this chapter offers some advice to someone planning a streaming architecture pilot project. Many of the software packages in this book can be considered interchangeable in the sense that they can all be made to function in any environment, but they all have their strengths and weaknesses. Based on real-world experience, this naturally means that some packages are easier to use in some environments than others. The final section of this chapter provides some insight into solutions that have worked in the past and can serve as a starting point for the future.

Real-Time Architecture Components

This section describes the components often found in a modern real-time architecture. Not every system will have every component, especially at first. The goal should be to build a system appropriate to an initial application, with the ability to expand to incorporate any missing components in the future.

After the first application is established, other applications of the infrastructure will inevitably follow with their own requirements. In fact, it is quite likely that some components will have multiple implementations. It happens most often in the storage component, where there is a wide range of performance characteristics among the different products available.

This section covers each of the components and some aspects of their development and future path. In general, the specifics of each component will be covered in more depth in their relevant chapters in the remainder of Part I.

Collection

The most common environment for real-time applications discussed in this book is handled over TCP/IP-based networks. As a result, the collection of data happens through connections over a TCP/IP network, probably using a common protocol such as HTTP. The most obvious example of this form of collection is a large latency-sensitive website that geo-distributes its servers (known as edge servers) to improve end-user latency.

With websites being one of the original use cases for large-scale data collection, it is not surprising that the log formats used by web servers dominated the formatting of log files. Unfortunately, most web servers adhered to the National Center for Supercomputing Applications (NCSA) Common Log Format and the later World Wide Web Consortium (W3C) Extended Log File Format, which was never intended to support the rich data payloads (and does so poorly).

Newer systems now log in a variety of formats, with JavaScript Object Notation (JSON) being one of the most popular. It has gained popularity by being able to represent rich data structures in a way that enables it to be easily extended. It is also relatively easy to parse, and libraries for it are widely available. It is used to transfer data between client-side web applications and the server side.

JSON's flexibility also leads to a significant downside: processing-time validation. The lack of defined schemas and the tendency of different packages to produce structurally different, but semantically identical structures means that processing applications must often include large amounts of validation code to ensure that their inputs are reasonable. The fact that JSON essentially sends its schema along with every message also leads to data streams that require relatively large bandwidth. Compression of the data stream helps, often achieving compression rates well more than 80 percent, but this suggests that something could be done to improve things further.

If the data is well structured and the problem fairly well understood, one of the structured wire formats is a possibility. The two most popular are Thrift and Protocol Buffers (usually called Protobuf). Both formats, the former developed at Facebook and the latter at Google, are very similar in their design (not surprising given that they also share developers). They use an Interface Definition Language (IDL) to describe a data structure that is translated into code that is usable in a variety of output languages. This code is used to encode and decode messages coming over the wire. Both formats provide a mechanism to extend the original message so that new information can be added.

Another, less popular option is the Apache Avro format. In concept it is quite similar to Protocol Buffers or Thrift. Schemas are defined using an IDL, which happens to be JSON, much like Thrift or Protobuf. Rather than using code generation, Avro tends to use dynamic encoders and decoders, but the binary format is quite similar to Thrift. The big difference is that, in addition to the binary format, Avro can also read and write to a JSON representation of its schema. This allows for a transition path between an existing JSON representation, whose informal schema can often be stated as an explicit Avro schema, and the more compact and well-defined binary representation.

For the bulk of applications, the collection process is directly integrated into the edge servers themselves. For new servers, this integrated collection mechanism likely communicates directly with the data-flow mechanisms described in the next section. Older servers may or may not integrate directly with the data-flow mechanism, with options available for both. These servers are usually application specific, so this book does not spend much time on this part of the environment except to describe the mechanisms for writing directly to the data-flow component.

Data Flow

Collection, analysis, and reporting systems, with few exceptions, scale and grow at different rates within an organization. For example, if incoming traffic remains stable, but depth of analysis grows, then the analysis infrastructure needs more resources despite the fact that the amount of data collected stays the same. To allow for this, the infrastructure is separated into tiers of collection, processing, and so on. Many times, the communication between these tiers is conducted on an ad hoc basis, with each application in the environment using its own communication method to integrate with its other tiers.

One of the aims of a real-time architecture is to unify the environment, at least to some extent, to allow for the more modular construction of applications and their analysis. A key part of this is the data-flow system (also called a data motion system in this book).

These systems replace the ad hoc, application-specific, communication framework with a single, unified software system. The replacement software systems are usually distributed systems, allowing them to expand and handle complicated situations such as multi-datacenter deployment, but they expose a common interface to both producers and consumers of the data.

The systems discussed in this book are primarily what might be considered third-generation systems. The “zero-th generation” systems are the closely coupled ad hoc communication systems used to separate applications into application-specific tiers.

The first generation systems break this coupling, usually using some sort of log-file system to collect application-specific data into files. These files are then generically collected to a central processing location. Custom processors then consume these files to implement the other tiers. This has been, by far, the most popular system because it can be made reliable by implementing “at least once” delivery semantics and because it's fast enough for batch processing applications. The original Hadoop environments were essentially optimized for this use case.

The primary drawback of the log-based systems is that they are fairly slow. Data must be collected in a batch form when a log file is “rolled” and processed en masse. Second-generation data-flow systems recognized that reliable transport was not always a priority and began to implement remote procedure call (RPC) systems for moving data between systems. Although they may have some buffering to improve reliability, the second-generation systems, such as Scribe and Avro, generally

accept that speed is acquired at the expense of reliability. For many applications this tradeoff is wholly acceptable and has been made for decades in systems-monitoring software such as Syslog, which uses a similar model.

Third-generation systems combine the reliability of the first-generation log models with the speed of the second-generation RPC models. In these systems, there is a real-time interface to the data layer for both producers and consumers of the data that delivers data as discrete messages, rather than the bulk delivery found in first-generation log systems. In practice, this is usually accomplished as a "mini batch" on the order of a thousand messages to improve performance.

However, these environments also implement an intermediate storage layer that allows them to make the same "at least once" delivery guarantees of log-based delivery systems. To maintain the requisite performance, this storage layer is horizontally scaled across several different physical systems with coordination handled by the client software on both the producer and consumer sides of the system.

The first of these systems were queuing systems designed to handle large data loads; ActiveMQ is an example. By providing a queuing paradigm, the queuing systems allow for the development of message "busses" that loosely coupled different components of the architecture and free developers from the communication task. The drawback of queuing systems has been the desire to maintain queue semantics where the order of delivery to consumers is matched to the order of submission. Generally, this behavior is unnecessary in distributed systems and, if needed, usually better handled by the client software.

Recognition of the fact that queuing semantics are mostly unneeded has led the latest entrants in the third-generation of data-flow systems, Kafka and Flume, to largely abandon the ordering semantics while still maintaining the distributed system and reliability guarantees. This has allowed them to boost performance for nearly all applications. Kafka is also notable in that it was explicitly designed to handle data flow in a large multi-datacenter installation.

Processing

Map-Reduce processing (Hadoop being the most popular open source version) became popular not because it was a game-changing paradigm. After all, the Hollerith tabulation machines used for the 1890 census used a tabulating mechanism very similar to the map-reduce procedure implemented by Hadoop and others.

NOTE

Hollerith later founded the corporations that would eventually become IBM.

Map-reduce processing also did not introduce the idea of distributed computation and the importance of the computation being local to the data for performance. The high-performance and large-scale computing communities have long known that it is often more efficient to move software close to the data, and there is no shortage of software available from that community for distributed computing.

What the map-reduce paper did do was to use this paradigm to efficiently arrange the computation in a way that did not require work on the part of the application developer, provided the developer could state the problem in this peculiar way. Large amounts of data had long been tabulated, but it was up to the specific application to decide how to divide that data among processing units and move it around. The first generation log collection systems made it possible to collect large amounts of data in a generic way. Map-reduce, Hadoop in particular, made it easy to distribute that processing so that it did not require application-specific code or expensive hardware and software solutions.

The same thing happened in the real-time space with the introduction of the second generation of data-flow frameworks. Data that had formerly been locked in log files became available in a low-latency setting. Like the original batch systems, the first systems tended to have task-specific implementations. For example, Twitter's Rainbird project implements a hierarchical streaming counting system using a database called Cassandra.

The current second-generation systems have moved beyond task-specific implementations into providing a general service for arranging streaming computation. These systems, such as Storm (also a Twitter project, but originally developed by a company called BackType) typically provide a directed acyclic graph (DAG) mechanism for arranging the computation and moving messages between different parts of the computation.

These frameworks, which are relatively young as software projects, are still often under heavy development. They also lack the guidance of well-understood computational paradigms, which means that they have to learn what works well and what does not as they go along. As their development progresses, there will probably be an erosion in the difference between batch-based systems and streaming systems. The map-reduce model is already gaining the ability to implement more complicated computation to support the demand for interactive querying as parts of projects like Cloudera's Impala and the Stinger project being led by Hortonworks.

Storage

Storage options for real-time processing and analysis are plentiful—almost to the point of being overwhelming. Whereas traditional relational databases can and are used in streaming systems, the preference is generally for so-called “NoSQL” databases. This preference has a number of different drivers, but the largest has generally been the need to prioritize performance over the ACID (Atomicity, Consistency, Isolation, Durability) requirements met by the traditional relational database. Although these requirements are often met to some degree, a common feature of these databases is that they rarely support all the requirements.

The other thing that these databases usually lack when compared to a relational database is a richness of schema and query language. The NoSQL moniker is a reference to the fact that this class of database usually has a much more restricted query language than that allowed by the SQL (Standard

Query Language) standard. This simplified and/or restricted query language is usually coupled with a restricted schema, if any formal schema mechanism exists at all.

The most common styles of NoSQL databases are the various forms of persistent key-value stores. They range from single-machine master-slave data stores, such as Redis, to fully distributed, eventually consistent, stores, such as Cassandra. Their fundamental data structure is the key with an arbitrary byte array value, but most have built some level of abstraction on the core entity. Some, such as Cassandra, even extend the abstraction to offering a SQL-like language that includes schemas and familiar-looking statements, although they don't support many features of the language.

The NoSQL database world also includes a variety of hybrid data stores, such as MongoDB. Rather than being a key-value store, MongoDB is a form of indexed document store. In many ways it is closer to a search engine like Google than it is to a relational database. Like the key-value stores, it has a very limited query language. It does not have a schema; instead it uses an optimized JSON representation for documents that allow for rich structure. Unlike most of the key-value stores, it also offers abstractions for reference between documents.

Along with the simplification of the query languages and maintenance of schemas comes a relaxation of the aforementioned ACID requirements. Atomicity and consistency, in particular, are often sacrificed in these data stores. By relaxing the consistency constraint to one of "eventual consistency," these stores gain some performance through reduced bookkeeping and a much simpler model for distributing themselves across a large number of machines that may be quite distant physically. In practice, for streaming applications, it is generally not necessary that each client have the same view of the data at the same time. The principal problem is when two physically separate copies of the database attempt to modify the same piece of state. Resolving this problem is tricky, but it is possible and is discussed in detail in Chapter 3, "Service Configuration and Coordination."

Relaxing the atomicity requirement also usually results in a performance gain for the data store, and maximum performance is the ultimate goal of all of these data stores. Most of them maintain atomicity in some lightweight way, usually the special case of counter types. Maintaining atomic counters, as it happens, is not too difficult and also happens to be a common use case, leading most data stores to implement some form of counter.

Each of the stores discussed in this book have different strengths and weaknesses. There is even a place in real-time applications for traditional relational databases. Different applications will perform better with one data store or another depending on the goal of the implementation, and the notion that a single data store is sufficient for all purposes is unrealistic. The aim of this book is to allow for an infrastructure that uses the real-time store as a good approximation of the system of record, which is likely to be based on a batch system and a relational database. Data stores are then chosen based on their performance characteristics for their problems with the expectation that the infrastructure can ensure that they all hold essentially the same data optimized for their use cases.

The acceptance that practical realities require that data be stored in different stores extends to integration with batch processing. Streaming architectures are well suited to tasks that can accept some error in their results. Depending on the calculations required, this error might even be intentional as discussed in Chapter 9, "Statistical Approximation of Streaming Data," and Chapter 10, "Approximating Streaming Data with Sketching." However, it is often the case that the data calculated must eventually be resolved and audited, which is a task much better suited to batch computation. Ideally, the streaming architecture makes the data available quickly and then is replaced by data from

the batch system as it becomes available. Nathan Marz, the primary creator of Storm, coined the term for this: Lambda Architecture. In the Lambda Architecture system, the user interface or another piece of middleware is responsible for retrieving the appropriate data from the appropriate store. As the convergence of the batch and real-time systems continues, through infrastructure improvements like the Apache YARN project, this should become the common case.

Delivery

The delivery mechanism for most real-time applications is some sort of web-based interface. They have a lot of advantages. They are relatively easy to implement and are lightweight to deploy in situ. There are dozens of frameworks available to aid the process.

Originally, these dashboards were websites with a refresh `META` tag that would reload a static representation of the process every few minutes. Many of the systems-monitoring tools, such as Graphite, still work that way. This works well enough for applications whose data is usually fairly slow to change. It also works well when a variety of environments need to be supported, as the technology is nearly as old as the web itself.

Web Communication Methods

Modern applications have a number of other options for delivering data to a web browser for display. A slightly faster version of the refreshing web page is to use the `XMLHttpRequest` (XHR) feature of most web browsers to load the data into the browser rather than an image of the data. This collection of technologies is also known by the name AJAX, which stands for "Asynchronous JavaScript and XML." It offers some improvements over the old model by separating the data layer and the presentational layer. This allows the application to tailor rendering of the data to the environment as well as usually reducing the size of the data transferred.

Transferring data via `XMLHttpRequest`, like the page refresh, is inherently built around polling. Data can only update as quickly as the polling frequency, which limits how quickly new information can be delivered to the front end. If events are relatively rare, such as with a notification system, a large number of calls are also wasted to ensure that "no news is good news." As the need for more real-time communication has grown, a number of "hacks" have been developed to make interactions between the browser and the server seem more immediate, by using techniques such as long polling. Collectively known as Comet, these methods now include true streaming to the browser, allowing for the implementation of truly real-time interfaces.

There are two standards currently in use: Server Sent Events (SSE) and Web Sockets. SSE implements an HTTP-compatible protocol for unidirectional streaming data from the server to the client. This is ideal for applications, such as a dashboard, which mostly take in data. Communication in the other direction is handled by normal AJAX calls. Web Sockets, on the other hand, implements a more complex protocol designed by bidirectional interaction between the client and the server. This protocol is not HTTP compatible, but it does have support from the three major desktop browsers (Internet Explorer, Chrome, and Firefox) as well as most of the mobile browsers. SSE are not available in Internet Explorer, but its HTTP compatibility means that it can fall back to polling for browsers that do not support the interface natively without extra server-side code.

Web-Rendering Methods

By sending the data to the client side, applications can also take advantage of the rendering

components built into modern web browsers: Scalable Vector Graphics (SVG) and Canvas drawing. Other drawing technologies—such as Microsoft's Vector Markup Language (VML), which was only supported by Internet Explorer, and the WebGL standard supported by Google's Chrome browser—have also been available over the years. These technologies attempt to bring a standard for 3D rendering to the web browser.

Introduced in the late 1990s, but not well supported until the current generation of web browsers, the SVG format offers web pages the ability to render resolution-independent graphics using a model very similar to the PostScript and Portable Display Format (PDF) drawing models. Like HTML, graphical elements are specified using a markup language that can display rectangles, circles, and arbitrary paths. These can be styled using Cascading Style Sheets (CSS) and animated using Document Object Model (DOM) manipulation. Originally only supported through a clunky plug-in interface, native SVG was added to Firefox 2.0 and early versions of the WebKit rendering engine used in both the Safari and Chrome browsers. Today it is supported by the current version of all major browsers and most mobile browser implementations. This broad support and resolution independence makes it an ideal choice for graphical rendering across varying platforms.

Originally introduced as part of Apple's Safari browser and subsequently included in the WHAT-WG proposal that serves as the basis for the HTML5 standard, the Canvas element also enjoys widespread support across browsers. The Canvas element provides a drawing surface rather than a Document Object Model for rendering an image. The drawing model itself is very similar to SVG because it also draws its inspiration from PostScript and PDF. However, it only maintains the actual pixels drawn rather than the objects used to render the pixels. This allows the canvas element to act more like the traditional HTML image element as well as conferring performance benefits relative to SVG in certain situations. In graphics circles, Canvas would be known as an “immediate mode” implementation while SVG is a “retained mode” implementation.

Features of a Real-Time Architecture

All of the components of real-time systems discussed in the last section share three key features that allow them to operate in a real-time streaming environment: high availability, low latency, and horizontal scalability. Without these features, the real-time system encounters fundamental limitations that prevent it from scaling properly. This section discusses what each of these features means in the context of real-time applications.

High Availability

The high-availability requirement for the entire system is probably the key difference between the real-time streaming application and the more common batch-processing or business-intelligence application. If these systems become unavailable for minutes or even hours, it is unlikely to affect operations. (Often, users of the system do not even notice.) Real-time systems, on the other hand, are sensitive to these sorts of outages and may even be sensitive to scheduled maintenance windows.

This may not extend to all parts of the stack, such as the delivery mechanism, but it is usually important for the collection, flow, and processing systems. To ensure high availability, most of the systems in this stack resort to two things: distribution and replication. *Distribution* means the use of multiple physical servers to distribute the load to multiple end points. If one of the machines in, say, the collection system, is lost, then the others can pick up the slack until it can be restored or replaced.

Of course, high availability does not refer to only the availability of the service; it also refers to the availability of the data being processed. The collection system generally does not maintain much local state, so it can be replaced immediately with no attempt to recover. Failures in the data-flow system, on the other hand, cause some subset of the data itself to become unavailable. To overcome this problem, most systems employ some form of replication.

The basic idea behind *replication* is that, rather than writing a piece of data to single-machine, a system writes to several machines in the hopes that at least one of them survives. Relational databases, for example, often implement replication that allows edits made to a master machine to be replicated to a number of slave machines with various guarantees about how many slaves, if any, must have the data before it is made available to clients reading from the database. If the master machine becomes unavailable for some reason, some databases can fail over to one of the slaves automatically, allowing it to become the master. This failover is usually permanent, promoting the new master on a permanent basis because the previous master will be missing interim edits when and if it is restored.

This same approach is also used in some of the software stacks presented in this book. The Kafka data motion system uses a master-slave style of replication to ensure that data written to its queues remains available even in the event of a failure. MongoDB, a data store, also uses a similar system for its replication implementation. In both cases, automatic failover is offered, although a client may need to disconnect and reconnect to successfully identify the new master. It is also the client's responsibility to handle any in-flight edits that had not yet been acknowledged by the master.

The other approach, often found in NoSQL data stores, is to attempt a masterless form high availability. Like the master-slave configuration, any edits are written to multiple servers in a distributed pool of machines. Typically, a machine is chosen as the “primary” write machine according to some feature of the data, such as the value of the primary key being written. The primary

then writes the same value to a number of other machines in a manner that can be determined from the key value. In this way, the primary is always tried first and, if unavailable, the other machines are tried in order for both writing and reading. This basic procedure is implemented by the Cassandra data store discussed in this book. It is also common to have the client software implement this multiple writing mechanism, a technique commonly used in distributed hash tables to ensure high availability. The drawback of this approach is that, unlike the master-slave architecture, recovery of a server that has been out of service can be complicated.

Low Latency

For most developers, "low latency" refers to the time it takes to service a given connection. Low latency is generally desirable because there are only so many seconds in a day (86,400 as it happens) and the less time it takes to handle a single request, the more requests a single-machine can service.

For the real-time streaming application, low latency means something a bit different. Rather than referring to the return time for a single request, it refers to the amount of time between an event occurring somewhere at the "edge" of the system and it being made available to the processing and delivery frameworks. Although not explicitly stated, it also implies that the variation between the latency of various events is fairly stable.

For example, in a batch system operating on the order of minutes the latency of some events is very low, specifically, those events that entered the batch to be processed right before the processing started. Events that entered the batch just after the start of a processing cycle will have a very high latency because they need to wait for the next batch to be processed. For practical reasons, many streaming systems also work with what are effectively batches, but the batches are so small that the time difference between the first and last element of the mini-batch is small, usually on the order of milliseconds, relative to the time scale of the phenomena being monitored or processed.

The collection components of the system are usually most concerned with the first definition of low latency. The more connections that can be handled by a single server, the fewer servers are required to handle the load. More than one is usually required to meet the high-availability requirements of the previous section, but being able to reduce the number of edge servers is usually a good source of cost savings when it comes to real-time systems.

The data flow and processing components are more concerned with the second definition of latency. Here the problem is minimizing the amount of time between an event arriving at the process and it being made available to a consumer. The key here is to avoid as much synchronization as possible, though it is sometimes unavoidable to maintain high availability. For example, Kafka added replication between version 0.7, the first publicly available version, and version 0.8, which is the version discussed in this book. It also added a response to the Producer API that informs a client that a message has indeed reached the host. Prior to this, it was simply assumed. Unsurprisingly, Kafka's latency increased somewhat between the two releases as data must now be considered part of an "in sync replica" before it can be made available to consumers of the data. In most practical situations, the addition of a second or two of latency is not critical, but the tradeoff between speed and safety is one often made in real-time streaming applications. If data can safely be lost, latency can be made very small. If not, there is an unavoidable lower limit on latency, barring advances in physics.

Horizontal Scalability

Horizontal scaling refers to the practice of adding more physically separate servers to a cluster of

servers handling a specific problem to improve performance. The opposite is *vertical scaling*, which is adding more resources to a single server to improve performance. For many years, vertical scaling was the only real option without requiring the developer to resort to exotic hardware solutions, but the arrival of low-cost high-performance networks as well as software advances have allowed processes to scale almost linearly. Performance can then be doubled by essentially doubling the amount of hardware.

The key to successfully horizontally scaling a system is essentially an exercise in limiting the amount of coordination required between systems. This limited coordination applies to both systems communication between different instances as well as those potentially on the same instances. One of the chief mechanisms used to accomplish this feat is the use of partitioning techniques to divide the work between machines. This ensures that a particular class of work goes to a well-known set of machines without the need for anyone to ask a machine what it is doing.

When coordination is required, the problem can become fairly complicated. There have been a number of algorithms developed over the years that are designed to allow loosely coupled machines to coordinate and handle events such as network outages. In practice, they have proven devilishly difficult to implement. Fortunately, these algorithms are now offered "as a service" through coordination servers. The most popular is the ZooKeeper system developed by Yahoo! and discussed in great detail in Chapter 3. Used by many of the other frameworks and software packages in this book, ZooKeeper greatly simplifies the task of efficiently coordinating a group of machines when required.

Languages for Real-Time Programming

There are a number of programming languages in use today, but only a few of them are widely used when implementing real-time streaming applications. Part of this is due to the fact that the core packages are implemented in a specific language, and the focus for the client libraries for that package is, of course, on its native language. In other cases, the speed or ease of use of the language makes it well suited for implementing real-time applications. This section briefly discusses some of the languages that are often encountered in real-time applications.

Java

The Java language was made public in the mid-1990s with the promise of "Write Once Run Anywhere" software. Initially, this was used to embed Java software into web pages in the form of applets. The plan was to let the web server act as the distribution mechanism for "fat" client-server applications. For a variety of reasons, this never really caught on, and Java is now disabled by default in many web browsers. What functionality it originally sought to provide was mostly taken over by Adobe's Flash product and, more recently, web pages themselves as the JavaScript engine built into every web browser has become sufficiently powerful to implement rich applications.

Although it lost on the client side, Java did find a niche on the server side of web applications because it's easier to deploy than its primary competitor, C++. It was particularly easy to deploy when database connections were involved, having introduced the Java Database Connectivity (JDBC) standard fairly early in its life. Most web applications are, essentially, attractive connections to a database, and Java became a common choice of server implementation language as a result.

The development of a staggering number of software libraries and the slow but steady improvements in performance made Java a natural choice when many of the first Big Data applications, especially Hadoop, were being developed. (Google's internal Map-Reduce application, however, was written in C++.) It made sense to develop the rest of the stack in Java because it was on the front end and on the back end in the form of Hadoop.

Because most of the packages in this book were written in Java, or at least written to run on the Java Virtual Machine (JVM), Java is used as the example language for most of this book. The reasons for this decision are largely the same as those used to implement the packages in the first place. The Java language has a variety of software libraries that are fairly easy to obtain thanks to its similarly wide array of build management systems.

In particular, this book uses a build system called Maven to handle the management of packages and their dependencies. Maven is nice because it provides the ability for distributors of software packages to make them available either through Maven's own central repository or through repositories that library authors maintain on their own. These repositories maintain the software itself and also describe the dependencies of that software package. Maven can then obtain all the appropriate software required to build the package automatically.

Scala and Clojure

In the last section it was mentioned that most of the software was built in Java or to "run on the Java Virtual Machine." This is because some of the software was written in languages that compile to run on the JVM and can interact with Java software packages but are not actually Java.

Although there are a number of non-Java JVM languages, the two most popular ones used in real-time application development are Scala and Clojure. In this book, Scala was used to implement the Kafka package used for data motion and the Samza framework used in conjunction with Kafka to process data. Scala has spent most of its life as an academic language, and it is still largely developed at universities, but it has a rich standard library that has made it appealing to developers of high-performance server applications. Like Java, Scala is a strongly typed object-oriented language, but it also includes many features from functional programming languages that are not included in the standard Java language. Interestingly, Java 8 seems to incorporate several of the more useful features of Scala and other functional languages.

Whereas Scala does not have a direct analog in the world of programming languages, Clojure is a JVM language based on the venerable Lisp language. There have been a number of Lisp interpreters written for Java over the years, but Clojure is compiled and can make direct use of Java libraries within the language itself. Clojure was used to implement the Storm streaming data processing framework discussed in this book. Lisp-style languages are ideally suited to the implementation of domain-specific languages, which are small languages used to implement specific tasks. This appears to have influenced the development of Storm's Trident Domain-Specific Language, which is used to define the flow of data through various steps of processing in Storm.

JavaScript

JavaScript, of course, is the lingua franca of the web. It is built into every web browser and, thanks to an arms race between browser developers, is now even fast enough to serve as a language for implementing web applications themselves.

The only relationship between JavaScript and Java, aside from the name—which was essentially a marketing gimmick—is a superficial similarity in their syntaxes. Both languages inherit much of their syntaxes from the C/C++ language, but JavaScript is much more closely related to Clojure and Scala.

Like Scala and Clojure, JavaScript is a functional language. In Java, an object is an entity that contains some data and some functions, called *methods*. These two entities are separate and treated quite differently by the Java compiler. When a method is compiled by Java, it is converted to instructions called byte code and then largely disappears from the Java environment, except when it is called by other methods. In a functional language, functions are treated the same way as data. They can be stored in objects the same way as integers or strings, returned from functions, and passed to other functions. This feature is heavily used in many different client-side JavaScript frameworks, including the D3 framework used in this book. The D3 framework, which stands for Data Driven Documents, uses functions to represent equations that it uses to manipulate web pages. For example, a function can be passed to the method of an object that tells a rectangle where to appear on the page. This allows the framework to respond to data coming from outside sources without a lot of overhead.

JavaScript is also increasingly found in the implementation of the delivery server itself, primarily in the form of the node.js project found at http://nodejs.org. This project combines the JavaScript engine from the Chrome web browser, known as V8, with an event loop library. This allows the inherently single-threaded server to implement event-driven network services that can handle large numbers of simultaneous connections. Although not nearly as efficient as the modern Java frameworks, it is generally sufficient for the implementation of the server that handles delivery of data to the front end. It also has an advantage over a non-JavaScript-based server in that the fallback support for browsers that do not support modern features, such as SVG and Canvas, can be implemented on the server side

using the same codebase as the client side. In particular, node.js has packages that allow for the creation of a DOM and a rendering service that allows for the rendering of SVG visualizations on the server side rather than the client side.

The Go Language

Although it's not used in this book, the Go language is an interesting new entry in the realm of high-performance application development, including real-time applications. Developed at Google by a fairly small team, the Go language combines the simplicity of the C language with a framework for developing highly concurrent applications. It shares core team members with the original UNIX team and the developers of the C language, which makes the similarity to C unsurprising.

The Go language is still very much under development, and its compiler is not as good as many C compilers (or even necessarily as good as Java's compilers at this point). That said, it has been shown to perform well on real-world benchmarks for web server development and already has clients for several of the packages discussed in this book, particularly Kafka and Cassandra. This makes it a potential candidate for the development of edge collection servers if not for the processing or delivery components of the real-time application.

A Real-Time Architecture Checklist

The remainder of the first half of this book discusses the frameworks and software packages used to build real-time streaming systems. Every piece of software has been used in some real-world application or another and they should not be considered to be in competition with one another. Each has its purpose and, many times, attempting to compare them in some general way is missing the point. Rather than comparing various software packages, this section offers a simple opinion about which packages, frameworks, or languages work best in a given setting.

Collection

In many cases, there will not be a choice. The edge server will be a pre-existing system, and the goal will be to retrofit or integrate that server with a real-time framework.

In the happy event that the edge server is being custom built to collect data from the outside world, a Java-based server is probably the best choice. Java is in no way a "sexy" language. In fact, in many cases it can be an unpleasant language to use. But it has been around for nearly 20 years, its shortcomings are well understood, and people have been trying to squeeze performance out of its frameworks for a long time. For the same amount of tuning effort in a real-world application, Java will almost always be faster.

Other compiled JVM languages, such as Scala or Clojure, yield performance similar to Java itself, particularly, if none of the advanced features of the language, such as Scala's collection library, are used. That said, the goals of a dedicated edge server are to be as simple as possible and to deliver the data to the data-flow framework quickly and safely. The same cannot be said of the various complicated Java frameworks. These more complicated frameworks are designed for building websites, not high-performance data collection systems. Instead, you should use a lightweight framework rather than a heavyweight solution designed from the implementation of relatively low volume interactive websites.

As mentioned earlier in the section on programming languages, the Go language often performs well at this task. Although it does not yet have the outright performance of the well-tuned Java web servers, it still manages to perform better than nearly everything else in practical benchmarks. Additionally, it seems to generally have a lighter memory footprint, and there are some indications that it should perform better than a Java application when there are a truly enormous number of connections being made to the server. Anecdotal evidence suggests that being able to handle in excess of 50k concurrent connections is not outside the realm of "normal" for Go. The downside is that almost nobody knows how to program in Go, so it can be difficult to locate resources.

Data Flow

If there is a pre-existing system to be integrated, choose Flume. Its built-in suite of interfaces to other environments makes it an ideal choice for adding a real-time processing system to a legacy environment. It is fairly easy to configure and maintain and will get the project up and running with minimal friction.

If retrofitting a system or building something new from scratch, there is no reason not to use Kafka. The only possible reason to not use it would be because it is necessary to use a language that does not have an appropriate Kafka client. In that case, the community would certainly welcome even a partial client (probably the Producer portion) for that language. Even with the new safety features, its

performance is still very good (to the point of being a counterexample to the argument against Scala in the previous section), and it essentially does exactly what is necessary.

Processing

Although Samza shows great promise, it is unfortunately still too immature for most first attempts at a real-time processing system. This is primarily due to its reliance on the Apache YARN framework for its distributed processing. The claim is that YARN can support many different types of computation, but the reality of the situation is that nearly all the documentation only considers the map-reduce case. This makes it difficult for an inexperienced user to get much traction with getting set up.

A user already familiar with the maintenance of a YARN cluster will have much more success with Samza and should consider it for implementing a real-time application. In many ways it is a cleaner design than Storm, and its native support of Kafka makes it easy to integrate into a Kafka-based environment.

For first-time users, Storm is the more successful framework. Although it too can be hosted in a YARN cluster, that is not the typical deployment. The non-YARN deployment discussed in this book is much easier for new users to understand, and it's relatively easy to manage. The only disadvantage is that it does not include native support for either Kafka or Flume. Chapter 5 addresses the integration, but the development cycle of the plug-ins is quite different from the mainline Storm code, which can cause incompatibilities around the time of release.

Storage

For relatively small build-outs, such as a proof of concept or a fairly low-volume application, Redis is the obvious choice for data storage. It has a rich set of abstractions beyond the simple key-value store that allows for sophisticated storage. It is easy to configure, install, and maintain, and it requires almost no real maintenance (it is even available as a turnkey option from various cloud-based providers). It also has a wide array of available clients.

The two drawbacks of Redis are that it has not really addressed the problem of horizontal scalability for writes and it is limited to available random access memory (RAM) of the master server. Like many of the tools in this book, it offers a master-slave style of replication with the ability to failover to one of the replicas using Redis Sentinel. However, this still requires that all writes go to a single server. There are several client-side projects, such as Twitter's Twemproxy, that attempt to work around this limitation, but there is no native Redis solution as of yet. There is an ongoing clustering project, but there is no timeline for its stable release.

If very large amounts of infrequently accessed data are going to be needed, Cassandra is an excellent choice. Early versions of Cassandra suffered from a “query language” that was essentially just an RPC layer on the internal data structures of the Cassandra implementation. This issue combined with a somewhat unorthodox data model (for the time) makes it very hard to get started with Cassandra. With the newest versions of Cassandra and version three of the Cassandra Query Language (CQL), these problems have mostly been corrected. Combined with other changes around the distribution of data in the cluster, Cassandra has become a much more pleasant environment to use and maintain. It is also quite fast, especially when coupled with Solid State Drive (SSD)–equipped servers. It is not quite as fast as Redis because it may have to retrieve data from disk, but it is not that much slower.

MongoDB is really a first choice only when the data to be stored has a rich structure—for example,

when streaming geographical data. MongoDB offers utilities and indexing to support geographical information system (GIS) data more efficient than either Redis or Cassandra. Other options in this space tend to be based on traditional relational stores with added geographical indexing, so MongoDB can provide a higher-performance alternative. Foursquare, which deals almost exclusively in geographical data, is an early and well-known user of MongoDB for this reason.

Delivery

Delivery of almost every application is going to be some sort of web application, at least for its first iteration. A native application, especially for mobile devices, certainly delivers a much more rich and satisfying experience for the user. It will also require a skillset that is not necessarily immediately available to most organizations. If mobile developers are available, by all means, use them. A good tablet or phone interface to a streaming application is much more compelling than a web interface, thanks to the native experience.

That said, the native mobile experience is still going to need a server that can deliver the data. The best way to do this is to implement application programming interfaces (APIs) that support SSE that can be used by both the web applications and native applications. The reason to choose SSE over Web Sockets, despite the fact that Web Sockets has slightly better support across different browsers, is its HTTP basis compared to Web Socket's non-HTTP interface. SSE is simply an HTTP connection, which allows for the use of “polyfills” that can simulate the SSE on older browsers. This is much more difficult, and sometimes impossible, with Web Sockets. By the same token, it is also easier to integrate SSE into a native experience than it is to integrate Web Sockets' more complicated protocol. Because Web Sockets is intended for use within web browsers, it is not necessarily available as a library in a native platform, whereas the standard HTTP library can usually be easily modified to support SSE by changing the keep alive and timeout behavior.

Conclusion

This chapter has provided a broad overview of the development of real-time applications. The various components, also known as tiers, of a real-time system have been described, and the names of some potential frameworks used in each tier have been introduced. In addition, the low-level tools of the real-time application—the programming languages—have also been briefly introduced. As mentioned at the beginning of this chapter, it is assumed that there is some knowledge of the languages discussed in this chapter to get the most of out of the rest of this book.

The rest of this book discusses each component described in this chapter in more depth. Chapter 3 begins by discussing the coordination server ZooKeeper, which was mentioned in the section on horizontal scalability. This is a critical piece of software used by several other packages in this book, making it deserving of its own chapter. These coordination servers are used to manage the data flow from the edge servers to the processing environment. Two packages used to manage data flows, Kafka and Flume, are covered in Chapter 4, “Flow Management for Streaming Analysis,” and should be used, as mentioned earlier, according to need. Next, the data is processed using either Storm or Samza, both of which are covered in Chapter 5, “Processing Streaming Data.” The results of this processing need to be stored, and there are a variety of options available. Some of the more popular options are laid out in Chapter 6, “Storing Streaming Data,” so that they can be accessed by the delivery mechanisms discussed in Chapter 7, “Delivering Streaming Metrics.”

Chapters 8 through 11 are more focused on building applications on top of this foundation. They introduce aggregation methods used in the processing section as well as statistical techniques applied to process data to accomplish some goal. Chapter 11, “Beyond Aggregation,” in particular, is reserved for more “advanced topics,” such as machine learning from streaming data.

Chapter 3
Service Configuration and Coordination

The early development of high-performance computing was primarily focused on scale-up architectures. These systems had multiple processors with a shared memory bus. In fact, modern multicore server hardware is architecturally quite similar to the supercomputer hardware of the 1980s (although modern hardware usually foregoes the Cray's bench seating).

As system interlinks and network hardware improved, these systems began to become more distributed, eventually evolving into the Network of Workstations (NOW) computing environments used in most present-day computing environments. However, these interlinks usually do not have the bandwidth required to simulate the shared resource environments of a scale-up architecture nor do they have the reliability of interlinks used in a scale-up environment. Distributed systems also introduce new failure modes that are not generally found in a scale-up environment.

This chapter discusses systems and techniques used to overcome these problems in a distributed environment, in particular, the management of shared state in distributed applications. Configuration management and coordination is first discussed in general. The most popular system, used by most of the other software in this book, is ZooKeeper. This system was developed by Yahoo! to help manage its Hadoop infrastructure and is discussed extensively because it is used by most of the other distributed software systems in this book.

Motivation for Configuration and Coordination Systems

Most distributed systems need to share some system metadata as well as some state concerning the distributed system itself. The metadata is typically some sort of configuration information. For example, it might need to store the location of various database shards in a distributed server. System state is usually data used to coordinate the application. For example, a master-slave system needs to keep track of the server currently acting as the master. Furthermore, this happens in a relatively unreliable environment, so a server must also somehow track the correctness of the current configuration or the validity of its coordination efforts.

Managing these requirements in a distributed environment is a notoriously difficult-to-solve problem, often leading to incorrect server behavior. Alternatively, if the problems are not addressed, they can lead to single points of failure in the distributed system. For “offline” processing systems, this is only a minor concern as the single point of failure can be re-established manually. In real-time systems, this is more of a problem as recovery introduces potentially unacceptable delays in processing or, more commonly, missed processing entirely.

This leads directly to the motivation behind configuration and coordination systems: providing a system-wide service that correctly and reliably implements distributed configuration and coordination primitives.

These primitives, similar to the coordination primitives provided for multithreaded development, are then used to implement distributed versions of high-level algorithms.

Maintaining Distributed State

Writing concurrent code that shares state within a single application is hard. Even with operating systems and development environments providing support for concurrency primitives, the interaction between threads and processes is still a common source of errors. The problems are further compounded when the concurrency spans multiple machines. Now, in addition to the usual problems of concurrency—deadlocks, race conditions, and so on—there are a host of new problems to address.

Unreliable Network Connections

Even in the most well-controlled datacenter, networks are unreliable relative to a single-machine. Latency can vary widely from moment to moment, bandwidth can change over time, and connections can be lost. In a wide area network, a "Backhoe Event" can sever connections between previously unified networks. For concurrent applications, this last event (which can happen within a single datacenter) is the worst problem.

In concurrent programming, the loss of connectivity between two groups of systems is known as the "Split Brain Problem." When this happens, a distributed system must decide how to handle the fact that some amount of state is suddenly inaccessible. There are a variety of strategies in practice, though the most common is to enter a degraded state. For example, when connectivity loss is detected, many systems disallow changes to the distributed state until connectivity has been restored.

Another strategy is to allow one of the partitions to remain fully functional while degrading the capabilities of the other partition. This is usually accomplished using a quorum algorithm, which requires that a certain number of servers are present in the partition. For example, requiring that a fully functional partition contain an odd number of processes is a common strategy. If an odd number of servers are split into two groups, one group always contains an odd number of servers whereas the other contains an even number of servers. The group with the odd number of servers remains functional, and the one with an even number of servers becomes read-only or has otherwise degraded functionality.

Clock Synchronization

It may seem like a simple thing, but distributed systems often require some sort of time synchronization. Depending on the application, this synchronization may need to be fairly precise. Unfortunately, the hardware clocks found in servers are not perfect and tend to drift over time. If they drift far enough, one server can experience an event that happened after the current time, resulting in some interesting processing events. For example, an analysis system that was interested in the difference between the timestamps of two types of events might start to experience negative duration.

In most circumstances, servers are synchronized using the Network Time Protocol (NTP). Although this still allows sometimes-significant drift between machines, it is usually "close enough." Problems can arise, however, when it is not possible to synchronize machines to the same set of NTP servers, which sometimes happens in environments that have a secure internal domain that communicates via a very limited gateway to an external-facing domain. In that case, an internal NTP server can drift away from NTP servers used by external users. In some situations, such as application programming interfaces (APIs) that use time to manage authorization, the drift can cause profound failures of the service.

Unfortunately, other than "don't do that" there is no easy solution to that particular problem.

Consensus in an Unreliable World

In addition to unreliable network connections and poorly synchronized clocks, there is the possibility that the processes themselves might contain faults that cause them to generate incorrect results at times. In some academic work on the subject, these processes may even be actively malicious, though many real-world systems do not consider this scenario.

In the face of all the things that can go wrong, the distributed state maintained across machines should be consistent. The most famous (though not the first) algorithm for doing this, known as Paxos, was first published in the late 1980s.

In the Paxos algorithm, a process that wants to mutate the shared state first submits a proposal to the cluster. This proposal is identified by a monotonically increasing integer, which must be larger than any identifier previously used by the Proposer.

The proposal is then transmitted to a quorum of other processes, called the Acceptors. If a given Acceptor receives a message identifier smaller than the largest it has seen from the proposer, it simply discards the proposal as being stale (alternatively, it can explicitly deny the request to the Proposer).

If the message seen by the Acceptor is the largest it has ever seen from the Proposer, it updates the largest identifier seen for the Proposer. At the same time, it returns the previous identifier and the associated value it had received from that Proposer. This is called the promise phase because the Acceptor has now promised not to deny all requests with an identifier smaller than the new value.

After the Proposer has received enough positive responses from the promise phase, it may consider its request to mutate the state to be accepted. The Proposer then sends the actual change to the Acceptors, who then propagate the information to other nodes in the cluster, which are called Learners. This process also identifies the Proposer whose proposal has been accepted as the Leader of the cluster. In practice, this is often used to implement Leader Election for a cluster of servers (for example, a clustered database application).

This basic form of Paxos is concerned with updating, essentially, the entire state. It can be used to update individual elements of a distributed state representation, but this tends to introduce a lot of overhead leading to the development of Multi-Paxos.

The Multi-Paxos is often the algorithm that is actually implemented in the real world. It depends on the fact that processes are relatively stable and that a change in the Leader of a cluster is going to be rare. This allows the algorithm to dispense with the Prepare-Promise phase of the communication protocol after a process has been declared a Leader. When a new Leader is elected it must repeat the Prepare-Promise cycle again, so the worst-case scenario is the basic Paxos algorithm.

Although these algorithms appear to be relatively straightforward, they have proven notoriously difficult to implement properly. Thus, it is recommended to use systems like those described in this chapter rather than attempting an implementation. This chapter introduces two of these systems: ZooKeeper and etcd. The ZooKeeper project is a distributed configuration and coordination tool used to coordinate many of the other distributed systems found in this book. Etcd is a new system that provides many of the same features of ZooKeeper through an HTTP-based interface and using a new consensus algorithm called Raft, which was designed explicitly to deal with the complexities of Paxos.

Apache ZooKeeper

The ZooKeeper project was developed at Yahoo! with the mission of providing exactly the primitives described in the last section. Originally developed to tame the menagerie of services being implemented in Yahoo!'s internal Hadoop environments, it has been open-sourced and contributed to the Apache Foundation.

It has since proven itself to be a valuable infrastructural component for the development of distributed systems. In fact, it is used to coordinate many of the applications used later in this book. The Kafka data motion system uses it to manage metadata for both its servers and clients. It is also a key component of the Storm data processing framework's server management features and plays a similar role for the Samza data processing framework.

Rather than imposing a particular set of coordination or configuration features, ZooKeeper provides a low-level API for implementing coordination and configuration using a system loosely based on a hierarchical file system. These features, often called recipes in ZooKeeper jargon, are left as an exercise for the application author. This has the advantage of simplifying the ZooKeeper codebase, but it does force the application developer to deal with some potentially difficult implementations.

Fortunately, there are client libraries, such as the Curator library discussed later in this chapter, that have carefully implemented the more complicated recipes. This section covers the usage of the Curator library, but first it covers the basics of ZooKeeper itself.

The znode

The core of ZooKeeper's data structures is the `znode`. These nodes can be arranged hierarchically into tree structures as well as hold data. Because ZooKeeper is not intended as a bulk storage facility, the amount of data a single `znode` can hold is limited to about 1 MB of data. It also means that ZooKeeper does not support partial reads, writes, or appends. When reading and writing data, the entire byte array is transmitted in a single call.

The `znode` API supports six fundamental operations:

- The **create** operation, which takes a path and optional data element. This, naturally, creates a new `znode` at the specified path with data if it does not exist.
- The **delete** operation, which removes a `znode` from the hierarchy.
- The **exists** operation, which takes a path and allows applications to check for the presence of a `znode` without reading its data.
- The **setData** operation, which takes the same parameters as the **create** operation. It overwrites data in an existing `znode`, but it doesn't create a new one if it does not already exist.
- The **getData** operation retrieves the data block for a `znode`.
- The **getChildren** operation retrieves a list of children of the `znode` at the specified path. This operation is a key part of many of the coordination recipes developed for ZooKeeper.

All of these operations are subject to ZooKeeper's access control policies. These operate much like file system permissions and dictate which clients may write to which part of a hierarchy. This allows for the implementation of multi-tenant Zookeeper clusters.

Ephemeral Nodes

A `znode` may be either persistent or ephemeral. With a persistent `znode`, the `znode` is only

destroyed when the **delete** operation is used to explicitly remove the `znode` from ZooKeeper.

Ephemeral nodes, on the other hand, are also destroyed when the client session that created the node loses contact with the ZooKeeper cluster or otherwise ends its session. In the former case, the time required for the loss of contact to cause the destruction of an ephemeral node is controlled by a heartbeat timeout.

Because they may be destroyed at any time, ephemeral nodes may not contain children. This may change in future releases of ZooKeeper, but as of the 3.4 series of releases, ephemeral nodes may be files, but they may not be directories.

Sequential Nodes

A `znode` may also be declared as sequential. When a sequential `znode` is created, it is assigned a monotonically increasing integer that is appended to the node's path.

This serves two useful purposes in many algorithms. First, it provides a mechanism for creating unique nodes. The counter ensures that a node name will never be repeated. Secondly, it provides a sorting mechanism when requesting the children of a parent node. This is used to implement, among other things, leader elections.

Versions

All `znodes`, when they are created, are given a version number. This version number is incremented whenever the data associated with a node changes. This version number can be optionally passed to the **delete** and **setData** operations, which allows clients to ensure that they do not accidentally overwrite changes to a node.

Watches and Notifications

The primary use case for ZooKeeper involves waiting for events, which are changes to data associated with a `znode` or with the children of the `znodes`. Rather than using polling, ZooKeeper uses a notification system to inform clients of changes.

This notification, called a `watch`, is established when the application registers itself with ZooKeeper to be informed of changes on a particular `znode` path. This is a one-time operation, so when the notification is fired, the client must reregister the `watch` to receive future notifications about changes.

When attempting to continuously monitor a path, it is possible to lose a notification. This can occur when a change is made to a path in the time after the client receives the notification of a change but before setting a watch for the next notification. Fortunately, ZooKeeper's designers anticipated this scenario, and setting a watch also reads data from the `znode`. This effectively allows clients to coalesce notifications.

Maintaining Consistency

ZooKeeper does not employ the Paxos' consensus algorithm directly. Instead it uses an alternative algorithm called Zab, which was intended to offer performance improvements over the Paxos algorithm.

The Zab algorithm operates on the order of changes to state as opposed to the Paxos algorithm that updates the entire state when changes are made. That said, it shares many of the same features as

Paxos: A Leader goes through a Proposal process, accepting acknowledgment from a quorum of recipients before committing the proposal. The algorithm also uses a counter system, called an epoch number, similar to the one employed by Paxos.

Creating a ZooKeeper Cluster

ZooKeeper is envisioned as a shared service, acting as a common resource for a number of different applications. It uses a quorum of machines to maintain high availability. One of these machines will be declared the Leader using the Zab algorithm described in the previous section and all changes will be replicated to the other servers in the quorum. This section describes installing the ZooKeeper server for use in either a standalone development environment or in a multiserver cluster.

Installing the ZooKeeper Server

ZooKeeper can often be found in the installation packages of Hadoop distributions. These distributions often have repositories compatible with various server flavors.

It is also possible to install ZooKeeper from the binary archives available on the Apache ZooKeeper website (http://zookeeper.apache.org). The most current version of the server at the time of writing is version 3.4.5. Generally speaking, this latest version works with most applications, but it introduces some differences relative to the 3.3 series of servers. For software that requires the 3.3 series of ZooKeeper, 3.3.6 is the latest stable version.

After unpacking the ZooKeeper tarball, the directory structure should look something like this:

```
$ cd zookeeper-3.4.5/
$ ls
CHANGES.txt               docs
LICENSE.txt               ivy.xml
NOTICE.txt                ivysettings.xml
README.txt                lib
README_packaging.txt      recipes
Bin                       src
build.xml                 zookeeper-3.4.5.jar
conf                      zookeeper-3.4.5.jar.asc
contrib                   zookeeper-3.4.5.jar.md5
dist-maven                zookeeper-3.4.5.jar.sha1
```

Before the server can be started, it must be configured. There is a sample configuration in the `conf` directory that provides a good starting point for configuring a ZooKeeper server. Start by copying that file to `zoo.cfg`, which is what the server startup script expects the configuration file to be called. The first few options in this file usually do not require modification, as they have to do with the time between heartbeat packets:

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5
```

The next parameter is the location of the ZooKeeper data directory. It defaults to `/tmp/zookeeper`, but you should change it immediately as most systems will discard data in the `/tmp` directory after some time. In this example it has been set to `/data/zookeeper`. In all

cases, the directory and all of the files within it should be owned and writable by the user that will be running the ZooKeeper server itself.

```
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/data/zookeeper
```

Despite the fact that ZooKeeper can only have a database as large as available RAM, make sure that there is plenty of space available in the data directory. ZooKeeper takes snapshots of the database along with logs of changes between snapshots that are stored in this directory.

Finally, the port used by clients and other servers to communicate with the server is specified:

```
# the port at which the clients will connect
clientPort=2181
```

There is no reason to change this port unless it happens to conflict with another pre-existing service.

When the configuration is finished, create the data directory in the appropriate location (such as `/data/zookeeper`). In that directory, create a file called `myid` that contains an integer that is unique to each ZooKeeper server. This file serves to identify the server to the rest of the cluster and ZooKeeper will not function without it. It should be assigned to a number between 1 and 255 on each node in the quorum:

```
echo 1 > /data/zookeeper/myid
```

When all this is done, the ZooKeeper daemon can be started with using the `zkServer.sh` script, found in ZooKeeper's `bin` subdirectory. The `zkServer.sh` script is set up like an `init.d` script and provides the usual stop, start, and restart commands. It also has a start-foreground option that is useful for development. This has the daemon log to the foreground rather than a log file:

```
$ bin/zkServer.sh start-foreground
JMX enabled by default
Using config: /Users/bellis/Projects/zookeeper-3.4.5/bin/../conf/zoo.cfg
[myid:] - INFO  [main:QuorumPeerConfig@101] - Reading configuration
  from: /Users/bellis/Projects/zookeeper-3.4.5/bin/../conf/zoo.cfg
[myid:] - INFO  [main:DatadirCleanupManager@78] -
  autopurge.snapRetainCount
[myid:] - INFO  [main:DatadirCleanupManager@79] -
  set to 0
[myid:] - INFO  [main:DatadirCleanupManager@101] - Purge task is not
  autopurge.purgeInterval scheduled.
[myid:] - WARN  [main:QuorumPeerMain@113] - Either no config or no
  quorum defined in config, running  in standalone mode
[myid:] - INFO  [main:QuorumPeerConfig@101] - Reading configuration
  from: /Users/bellis/Projects/zookeeper-3.4.5/bin/../conf/zoo.cfg
[myid:] - INFO  [main:ZooKeeperServerMain@95] - Starting server
[myid:] - INFO  [main:Environment@100] - Server
  environment:zookeeper.version=3.4.5-1392090,
  built on 09/30/2012 17:52 GMT
[myid:] - INFO  [main:Environment@100] - Server
  environment:host.name=byrons-air
[myid:] - INFO  [main:Environment@100] - Server
  environment:java.version=1.7.0_45
[myid:] - INFO  [main:Environment@100] - Server
  environment:java.vendor=Oracle Corporation
[myid:] - INFO  [main:Environment@100] - Server
  environment:java.home=/Library/Java/JavaVirtualMachines/
```

```
jdk1.7.0_45.jdk/Contents/Home/jre
```

Choosing a Quorum Size

During development, a single ZooKeeper server is usually sufficient. However, for production systems, using a single ZooKeeper server would introduce a single point of failure. To overcome this, a cluster of servers, called a quorum, is deployed to provide fault tolerance against the loss of a single-machine.

An important, often overlooked, aspect of ZooKeeper is that the size of the cluster can have a large effect on performance. Due to the need to maintain consensus, as the cluster increases so does the time required for ZooKeeper to make changes to its state. As a result, there is an inverse relationship between fault tolerance, in the form of more servers, and performance.

Generally speaking, there is no real reason to have a ZooKeeper cluster larger than five nodes and, because an even number of servers does not increase fault tolerance, no reason to have fewer than three nodes. Choosing either five or three nodes for a cluster is mostly a matter of load on the cluster. If the cluster is only being lightly utilized for applications such as leader elections, or it's mostly used for reading configuration, then five nodes provide more fault tolerance. If the cluster will be heavily utilized, as can happen when it's used for applications like Kafka (which is discussed in Chapter 4, “Flow Management for Streaming Analysis”), then the added performance of only having three nodes is more appropriate.

Monitoring the Servers

ZooKeeper provides two mechanisms for monitoring server components of the quorum. The first is a simple set of monitoring keywords that can be accessed over ZooKeeper's assigned ports. The other mechanism is through the Java Management Extensions (JMX) interface. Although these two mechanisms overlap in some places, they have some complementary data that makes both of them useful in a single environment.

The native ZooKeeper monitoring keywords—colloquially known as the “four-letter words”—are a series of commands that return state information about the ZooKeeper cluster. As the nickname suggests, all of these commands consist of four letters: `dump`, `envi`, `reqs`, `ruok`, `srst`, and `stat`. The commands themselves are transmitted in the clear to ZooKeeper on its client port, returning plaintext output. For example, to send the `ruok` command from a shell command, simply use `echo` and the `netcat` utility:

```
echo ruok | nc 127.0.0.1 2181
```

If the ZooKeeper server is operating normally, the server returns an `imok` response. If the server is in a bad state it does not send a response at all. By sending this command periodically, monitoring software can ensure that all nodes are up and running.

The `dump` command returns data about currently connected sessions and ephemeral nodes being held by those sessions. It must be run against the current Leader of the ZooKeeper cluster. In this particular example, the distributed queue producer example found later in this chapter is running and the command returns the following information:

```
$ echo dump | nc 127.0.0.1 2181
SessionTracker dump:
Session Sets (2):
0 expire at Wed Jan 15 22:13:03 PST 2014:
```

```
1 expire at Wed Jan 15 22:13:06 PST 2014:
0x143999dfb5c0004
ephemeral nodes dump:
Sessions with Ephemerals (0):
```

In this case, there are no ephemeral nodes being used, and two sessions have been opened.

The `envi` command dumps information from the server's environment variables. For example, this ZooKeeper instance is included with Kafka 2.8.0 (covered in Chapter 4) and is running under Oracle Java 7 on OS X:

```
$ echo envi | nc 127.0.0.1 2181
Environment:
zookeeper.version=3.3.3-1203054, built on 11/17/2011 05:47 GMT
host.name=byrons-air
java.version=1.7.0_45
java.vendor=Oracle Corporation
java.home=/Library/Java/JavaVirtualMachines/jdk1.7.0_45.jdk/Contents/
  Home/jre
java.class.path=:
    ../core/target/scala-2.8.0/*.jar:
    ../perf/target/scala-2.8.0/kafka*.jar:
    ../libs/jopt-simple-3.2.jar:
    ../libs/log4j-1.2.15.jar:
    ../libs/metrics-annotation-2.2.0.jar:
    ../libs/metrics-core-2.2.0.jar:
    ../libs/scala-compiler.jar:
    ../libs/scala-library.jar:
    ../libs/slf4j-api-1.7.2.jar:
    ../libs/slf4j-simple-1.6.4.jar:
    ../libs/snappy-java-1.0.4.1.jar:
    ../libs/zkclient-0.3.jar:
    ../libs/zookeeper-3.3.4.jar:
    ../kafka_2.8.0-0.8.0.jar
java.library.path=/usr/local/mysql/lib::
  /Users/bellis/Library/Java/Extensions:
  /Library/Java/Extensions:
  /Network/Library/Java/Extensions:
  /System/Library/Java/Extensions:/usr/lib/java:.
java.io.tmpdir=/var/folders/5x/0h_qw77n4q73ncv4l16697l80000gn/T/
java.compiler=<NA>
os.name=Mac OS X
os.arch=x86_64
os.version=10.9.1
user.name=bellis
user.home=/Users/bellis
user.dir=/Users/bellis/Projects/kafka_2.8.0-0.8.0
```

The `stat` command returns information about the performance of the ZooKeeper server:

```
$ echo stat | nc 127.0.0.1 2181
Zookeeper version: 3.3.3-1203054, built on 11/17/2011 05:47 GMT
Clients:
 /127.0.0.1:58503[1](queued=0,recved=731,sent=731)
 /127.0.0.1:58583[0](queued=0,recved=1,sent=0)
Latency min/avg/max: 0/2/285
Received: 762
Sent: 761
Outstanding: 0
Zxid: 0x2f9
Mode: standalone
```

```
Node count: 752
```

As shown in the preceding code, the `stat` command returns information about the ZooKeeper server's response latency, the number of commands received, and the number of responses sent. It also reports the mode of the server, in this case a `standalone` server used along with Kafka for development purposes. Sending the `srst` command resets the latency and command statistics. In this example, the `srst` command was sent after the previous `stat` command while a queue example was running. This resets the count statistics, but the queue had been adding more elements to the queue, increasing the node count:

```
$ echo srst | nc 127.0.0.1 2181
Server stats reset.
$ echo stat | nc 127.0.0.1 2181
Zookeeper version: 3.3.3-1203054, built on 11/17/2011 05:47 GMT
Clients:
 /127.0.0.1:58725[0](queued=0,recved=1,sent=0)
 /127.0.0.1:58503[1](queued=0,recved=1933,sent=1933)
Latency min/avg/max: 2/2/3
Received: 15
Sent: 15
Outstanding: 0
Zxid: 0x7ab
Mode: standalone
Node count: 1954
```

Finally, the `reqs` command returns currently outstanding requests, useful for determining whether or not clients are blocked.

ZooKeeper's Native Java Client

The ZooKeeper library ships with a Java client that you can use in projects. This section describes setting up a Maven project to use the ZooKeeper library. It also covers the basic usage of the client.

Adding ZooKeeper to a Maven Project

The basic ZooKeeper client library, version 3.4.5 at the time of writing, is hosted on Maven Central and can be included in a project by adding the following dependencies to the project's `pom.xml`:

```
<dependency>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  <version>3.4.5</version>
  <exclusions>
    <exclusion>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
    </exclusion>
  </exclusions>
</dependency>
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.16</version>
</dependency>
```

Note that the `log4j` artifact has been excluded from the ZooKeeper artifact and then manually included. This addresses a problem specific to the version of Log4J included by ZooKeeper 1.2.15, which references libraries that are no longer accessible. If a newer version of ZooKeeper has been

released, this may have been addressed, which eliminates the need for this workaround.

Connecting to ZooKeeper

A ZooKeeper session is established by creating a `ZooKeeper` object using its standard constructors. The most complicated version of this constructor has the following signature:

```
public ZooKeeper(
  String connectString,
  int sessionTimeout,
  Watcher watcher,
  long sessionId,
  byte[] sessionPasswd,
  boolean canBeReadOnly
) throws IOException
```

The shortest version of the constructor only needs the `connectionString` and `sessionTimeout` arguments.

The `connectionString` is a comma-separated list of hostname and port pairs. In version 3.4 and later, an optional path can be appended to the hostname. This acts as a sort of `chroot` when connecting the server, instructing the client to treat the path as the root of the server.

When the client connects to ZooKeeper, it first shuffles the hosts listed in the connection string into a random order. It then tries the first host in the shuffled list. If connecting to that host fails, it tries the second host and so on through the list.

For example, to connect to three servers named `zookeeper1`, `zookeeper2`, and `zookeeper3` but with subdirectory `other`, the ZooKeeper client would be initialized as follows:

```
new ZooKeeper(
  "zookeeper1,zookeeper2,zookeeper3/other",
  3000,
  new Watcher() {
    public void process(WatchedEvent event) {
    }
  }
);
```

The Watcher interface implemented in the preceding code is an empty implementation for this example, but it is usually used to respond to all notifications sent by ZooKeeper. Sometimes the type field of the `event` object is empty, in which case the notification pertains to the connection state. At other times the Watcher is used to receive watch events for requests that set a watch without specifying a specific `Watcher` object.

The commands used by the ZooKeeper client are the same as the commands described in the previous section: `create`, `delete`, `exists`, `getData`, `setData`, and `getChildren`. Each of these commands returns results immediately. The commands optionally set a watch on the specified path, either returning the event to the client-level watch specified earlier or to another Watcher object that is passed into the command. The easiest way to see this in action is through an example.

Leader Election Using ZooKeeper

One of the most common uses for ZooKeeper is to manage a quorum of servers in much the same way it manages itself. In these quorums, one of the machines is considered to be the Leader, while other machines are operating as either replicas or as hot-standby servers, depending on the application.

This example uses the simplest possible algorithm for implementing leader election:

- Create an ephemeral sequential node in the target path.
- Check the list of children. The smallest node in the list of children is the leader.
- Otherwise, watch the list of children and try again when it changes.

In this case, the `LeaderElection` class implements a latching approach to leader election. First, the class is initialized with a path and a ZooKeeper client:

```
public class LeaderElection implements Watcher, ChildrenCallback {
  String    path;
  ZooKeeper client;
  String    node;
  public Integer   lock      = new Integer(0);
  public boolean   leader    = false;
  public boolean   connected = false;
  public LeaderElection(String connect,String path)
    throws IOException {
    this.path = path;
    client = new ZooKeeper(connect, 3000, this);
  }
```

The class registers itself as the Watcher for events on this client connection. When the client connects to ZooKeeper, it should request a position in the leadership queue. If the connection fails, it should relinquish leadership (if it had it) and inform the server that the connection has been lost. This is handled by the process method:

```
public void process(WatchedEvent event) {
  if(event.getType() == Watcher.Event.EventType.None) {
  switch(event.getState()) {
  case SyncConnected:
    connected = true;
    try {
      requestLeadership();
    } catch(Exception e) {
      //Quit
      synchronized(lock) {
        leader = false;
        lock.notify();
      }
    }
    break;
  case Expired:
  case Disconnected:
    synchronized(lock) {
      connected = false;
      leader    = false;
      lock.notify();
    }
    break;
  default:
```

```
      break;
    }
    } else {
      if(path.equals(event.getPath()))
        client.getChildren(path, true, this, null);
    }
  }
```

In the Watcher method, first the event is checked to see if it is a ZooKeeper status event (`EventType.Node`). If it is a connection event, then a leadership position is requested. If it is not a connection event, the path is checked. If it is the same path as the one registered when creating the object, a list of children is obtained. In this case, the callback version of the event is used so there only needs to be a single location where leadership status is checked. The watch is also reset when `getChildren` is called.

When leadership is requested, first the main path is created if it does not exist. This call can produce errors because each client tries to create the path. If the path already exists by the time the client calls the `create` method, the error can be safely ignored. Next, an ephemeral and sequential node is created in this path. This node is the server's node and is stored for all future transactions. Before the node is created, the object registers itself for changes to the main path so that the act of creating a new node registers a notification in the event that only one server is started. This is all implemented in `requestLeadership`, shown here:

```
  public void requestLeadership() throws Exception {
    Stat stat = client.exists(path, false);
    if(stat == null) {
      try {
        client.create(path, new byte[0],
          ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
      } catch(KeeperException.NodeExistsException e) { }
    }
    client.getChildren(path, true, this, null);
    node = client.create(path+"/n_", new byte[0],
      ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL_SEQUENTIAL);
    System.out.println("My node is: "+node);
  }
```

When the `getChildren` calls return their results, they use a callback method, `processResult`, defined in the `ChildrenCallback` interface. In this example, the method calls `isLeader` with the array of child nodes to check to see if it is first in the list. If that is the case, it updates its leader status and notifies any threads that might be waiting for leadership to be obtained that the leader status might have changed:

```
  public void processResult(int rc,
      String path, Object ctx,
      List<String> children) {
    synchronized(lock) {
      leader = isLeader(children);
      System.out.println(node+" leader? "+leader);
      lock.notify();
    }
  }
```

When ZooKeeper returns a list of children, it does not guarantee any sort of ordering for the elements. To find out if the node is the leader, which is defined as having the smallest node sequence, the list

first has to be sorted. Then the first element can be checked against the node name:

```
protected boolean isLeader(List<String> children) {
  if(children.size() == 0) return false;
  Collections.sort(children);
  System.out.println(path+"/"+children.get(0)
    +" should become leader. I am "+node);
  return (node.equals(path+"/"+children.get(0)));
}
```

Notice that the `getChildren` response does not include the full path. That has to be added during the comparison because the `create` method does return the full path.

A server waiting to become leader calls the `isLeader` method without any arguments in a loop. If the node is currently the leader, the method quickly returns **`true`** and the server can continue processing. If it is not the leader, the calling thread waits on the lock object until it has been notified by `processResult` or because the server has lost its connection to ZooKeeper:

```
public boolean isLeader() {
  if(leader && connected) {
    System.out.println(node+" is the leader.");
    return leader;
  }
  System.out.println("Waiting for a change in the lock.");
  synchronized(lock) {
    try {
      lock.wait();
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
  return leader && connected;
}
```

To see the example in action, a server is needed. This server does not do anything except announce that it is "processing" something in a loop while it has leadership. However, it is slightly unreliable, and 30 percent of the time it crashes, relinquishing its leadership position. If it loses leadership without crashing, it goes back to waiting for leadership. If it has lost connection, it exits entirely:

```
public class UnreliableServer implements Runnable {
  String name;
  LeaderElection election;
  public UnreliableServer(String name,String connect,String path)
    throws IOException {
    this.name = name;
    election = new LeaderElection(connect,path);
  }
  public void run() {
    System.out.println("Starting "+name);
    Random  rng  = new Random();
    do {
      if(election.isLeader()) {
        System.out.println(name+": I am becoming the leader.");
        do {
          if(rng.nextDouble() < 0.30) {
            System.out.println(name+": I'm about to fail!");
            try {
              election.close();
```

```
            } catch (InterruptedException e) { }
            return;
          }
          System.out.println(name
            +": I'm leader and handling a work element");
          try {
            Thread.sleep(rng.nextInt(1000));
          } catch (InterruptedException e) { }
      } while(election.isLeader());
      //If we lose leadership but are still connected, try again.
        if(election.connected) {
          try {
            election.requestLeadership();
          } catch (Exception e) {
            return;
          }
        }
      }
    } while(election.connected);
  }
  }
```

To see it in action, build a little test harness with three different servers that all take leadership until they crash:

```
public class LeaderElectionTest {
  @Test
  public void test() throws Exception {
    Thread one   = new Thread(new UnreliableServer("huey","localhost"
      ,"/ducks"));
    Thread two   = new Thread(new UnreliableServer("dewey","localhost"
      ,"/ducks"));
    Thread three = new Thread(new UnreliableServer("louis","localhost"
      ,"/ducks"));
    one.start();
    two.start();
    three.start();
    //Wait for all the servers to finally die
    one.join();
    two.join();
    three.join();
  }
}
```

This is an example output from this process. Each time the child list changes, the servers all check to see if they are leader. The one that is leader starts processing until it "crashes." This continues until all three have crashed:

```
Starting huey
Starting dewey
Waiting for a change in the lock.
Waiting for a change in the lock.
Starting louis
Waiting for a change in the lock.
My node is: /ducks/n_0000000009
My node is: /ducks/n_0000000010
My node is: /ducks/n_0000000008
/ducks/n_0000000009 leader? false
/ducks/n_0000000010 leader? false
```

```
Waiting for a change in the lock.
Waiting for a change in the lock.
/ducks/n_0000000008 leader? false
Waiting for a change in the lock.
/ducks/n_0000000008 should become leader. I am /ducks/n_0000000008
/ducks/n_0000000008 leader? true
/ducks/n_0000000008 should become leader. I am /ducks/n_0000000009
louis: I am becoming the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 should become leader. I am /ducks/n_0000000010
/ducks/n_0000000010 leader? false
/ducks/n_0000000009 leader? false
Waiting for a change in the lock.
Waiting for a change in the lock.
/ducks/n_0000000008 is the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 is the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 is the leader.
louis: I'm leader and handling a work element
/ducks/n_0000000008 is the leader.
louis: I'm about to fail!
/ducks/n_0000000009 should become leader. I am /ducks/n_0000000010
/ducks/n_0000000010 leader? false
/ducks/n_0000000009 should become leader. I am /ducks/n_0000000009
/ducks/n_0000000009 leader? true
Waiting for a change in the lock.
dewey: I am becoming the leader.
dewey: I'm leader and handling a work element
/ducks/n_0000000009 is the leader.
dewey: I'm leader and handling a work element
/ducks/n_0000000009 is the leader.
dewey: I'm about to fail!
/ducks/n_0000000010 should become leader. I am /ducks/n_0000000010
/ducks/n_0000000010 leader? true
huey: I am becoming the leader.
huey: I'm leader and handling a work element
/ducks/n_0000000010 is the leader.
huey: I'm leader and handling a work element
/ducks/n_0000000010 is the leader.
huey: I'm about to fail!
```

There are optimizations of this process that result in fewer messages being sent. In particular, the server awaiting leadership only really needs to watch the next smallest node in the last, not the entire set. When that node is deleted, the check to find the smallest node is repeated.

The Curator Client

The popular Curator client is a wrapper around the native ZooKeeper Java API. It is commonly used in lieu of directly using the ZooKeeper API as it handles some of the more complicated corner cases associated with ZooKeeper. In particular, it addresses the complicated issues around managing connections in the ZooKeeper client.

The Curator client was originally developed by Netflix, but has since been contributed to the Apache Foundation where it entered the incubation process in March 2013 (http://curator.apache.org).

Adding Curator to a Maven Project

The Curator library has been broken into several different components, all of which are available

from Maven Central. Adding the curator-recipes artifact to the project includes all of the relevant components:

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>2.3.0</version>
</dependency>
```

Most users will only ever need to use this artifact because it also includes in-depth implementations of recipes provided in the last section. If the recipes should not be included, the framework can be imported by itself via the curator-framework artifact:

```
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-framework</artifactId>
  <version>2.3.0</version>
</dependency>
```

Connecting to ZooKeeper

The Curator ZooKeeper client class is `CuratorFramework`, which is created by calling the `newClient` static method of the `CuratorFrameworkFactory` class. For example, connecting to a local ZooKeeper instance would look like this:

```
CuratorFramework client = CuratorFrameworkFactory.newClient(
  "localhost:2181",
  new ExponentialBackoffRetry(1000,3)
);
```

The first argument is the usual ZooKeeper connection string. As usual, it can contain a comma-separated list of available hosts, one of which is selected for connection. The second argument is a retry policy, which is unique to Curator. In this case the `ExponentialBackoffRetry` policy is used with a timeout of 1000 milliseconds and a maximum of three tries. This policy retries connection attempts with increasingly longer waits between attempts.

Curator has several other retry policies available for use. The `RetryNTimes` policy is like the `ExponentialBackoff` policy, except that it waits the same amount of time between attempts instead of increasing the wait. The `RetryUntilElapsed` waits a fixed amount of time between retries, like the `RetryNTimes` policy. The difference is that it continues trying until a certain amount of time has elapsed instead of trying a fixed number of times.

The `CuratorFrameworkFactory` class can also return a connection `Builder` class instead of immediately creating the client. This is useful when using some of Curator's advanced features. The previously described connection looks like this with the `Builder` method:

```
CuratorFrameworkFactory.Builder builder =
  CuratorFrameworkFactory.builder()
    .connectString("localhost:2181")
    .retryPolicy(new ExponentialBackoffRetry(1000,3));
```

In addition to the basic connection string and retry policy, the connection can be allowed to enter a read-only state in the case of a network partition with the `canBeReadOnly` method:

```
builder.canBeReadOnly(true);
```

A namespace, which is prepended to all paths, can be set using the namespace method:

```
builder.namespace("subspace");
```

Finally, to create the `CuratorFramework` object, call the build method:

```
CuratorFramework client = builder.build();
```

After the `CuratorFramework` object has been created, the connection must be started using the `start` method. Most of the `CuratorFramework` methods will not work if the connection has not been started.

The `start` method is used to give applications the chance to bind Listener methods, to the client before events are sent. This is particularly important when listening for connection state events. For example, the following code snippet, found in `wiley.streaming.curator.ConnectTest`, causes state changes to be printed to the console:

```
CuratorFramework client =
  CuratorFrameworkFactory.builder()
    .connectString("localhost:2181")
    .retryPolicy(new ExponentialBackoffRetry(1000,3))
    .namespace("subspace")
    .build();
client.getConnectionStateListenable().addListener(
  new ConnectionStateListener() {
    public void stateChanged(
      CuratorFramework arg0,
      ConnectionState arg1
    ) {
      System.out.println("State Changed: "+arg1);
      }
});
client.start();
Thread.sleep(1000);
```

In this example, the client is first built using the `Builder` interface with a namespace associated with it. Then a `Listener` is implemented to print out the changes in state as they happen. Finally, the client is started using `start`. Running this code should output a connected command if a short `Thread.sleep` command is added after the `start` to give the client some time to connect:

```
log4j:WARN No appenders could be found for logger
  (org.apache.curator.framework.imps.CuratorFrameworkImpl).
log4j:WARN Please initialize the log4j system properly.
State Changed: CONNECTED
```

Working with znodes

The Curator framework uses a "fluent" style for most of its commands. In this style of framework, most methods return an object that allows calls to be chained together. The `Builder` object in the last section was an example of this interface, and it is continued in all of Curator's `znode` manipulation routines.

Using the fluent style, Curator directly supports all of the basic `znode` operations as a series of chained method calls terminated by a call to `forPath`. The `forPath` method takes a complete path and an optional `byte` array containing any payload that should be associated with this `znode`. This example creates a new path in a synchronous fashion, creating parent `znodes` as needed:

```
client.create()
  .creatingParentsIfNeeded()
  .forPath("/a/test/path");
```

These methods can also be executed asynchronously by adding the `inBackground` call to the method chain:

```
client.create()
.creatingParentsIfNeeded()
.inBackground(new BackgroundCallback() {
public void processResult(CuratorFramework arg0, CuratorEvent arg1)
  throws Exception {
    System.out.println("Path Created: "+arg1);
  }
})
.forPath("/a/test/path");
```

Note that the `inBackground` method returns a `PathAndBytesable<T>` object rather than the original builder. This object only supports the `forPath` method, so it must be last in the chain of calls. When executed, as in the example provided by `wiley.streaming.curator.PathTest`, this command should have an output something like this:

```
Path Created: CuratorEventImpl{
  type=CREATE,
  resultCode=0,
  path='/a/test/path',
  name='/a/test/path',
  children=null,
  context=null,
  stat=null,
  data=null,
  watchedEvent=null,
  aclList=null
}
```

To create paths with different modes, use the `withMode` method in the `create` command. For example, a queuing system might want to create `znodes` that are both persistent and sequential:

```
client.create()
  .withMode(CreateMode.PERSISTENT_SEQUENTIAL)
  .forPath("/queue/job")
;
```

Valid modes are `PERSISTENT`, `EPHEMERAL`, `PERSISTENT_SEQUENTIAL`, and `EPHEMERAL_SEQUENTIAL`. The mode can also be determined using the `CreateMode .fromFlag` method, which converts the flags used by the native ZooKeeper client to the appropriate `CreateMode` option as demonstrated in the `wiley.streaming .curator.ModeTest` example.

The `checkExists` command has two uses. First, whether or not the command returns `null` allows for the existence check implied by its name. If the path given exists, a `Stat` object is returned with a variety of useful information about the path. This includes the number of children the path has, the version number, and the creation and modification times.

The data element or children of a `znode` are retrieved using the `getData` and `getChildren` commands, respectively. Like the `create` command, they can be executed in the background and always end with a `forPath` method. Unlike `create`, they do not take an optional byte array in their path.

When used immediately, the `getData` command returns a byte array, which can be deserialized into

whatever format the application desires. For example, if the data were a simple text string, it can be set and then retrieved as follows (assuming the `znode` already exists):

```
client.setData()
  .forPath("test/string", "A Test String".getBytes());
System.out.println(
  new String(
    client.getData()
      .forPath("test/string")
  )
);
```

Like the create command, `getData` can be executed in the background. In the background case, the `forPath` method returns `null` and the data is returned in the background callback. For example, the same code as the previous example implemented as a background callback would look like this:

```
client.getData().inBackground(new BackgroundCallback() {
public void processResult(CuratorFramework arg0, CuratorEvent arg1)
  throws Exception {
    System.out.println(new String(arg1.getData()));
  }
}).forPath("test/string");
```

The previous two examples are implemented in `wiley.streaming.curator .DataTest`.

The `getChildren` command works essentially the same way as the `getData` command. Instead of returning a byte array, this command returns a list of `String` objects giving the names of the child `znodes` of the parent object. This example shows this command in action by first adding several jobs to an imaginary queue `znode` and then calling `getChildren` on the node to retrieve the list of children, as demonstrated in `wiley.streaming.curator.QueueTest`:

```
if(client.checkExists().forPath("/queue") == null)
  client.create().forPath("/queue");
for(int i=0;i<10;i++) {
  client.create()
    .withMode(CreateMode.PERSISTENT_SEQUENTIAL)
    .forPath("/queue/job-");
}
for(String job : client.getChildren().forPath("/queue")) {
  System.out.println("Job: "+job);
}
```

When this is run, this piece of code returns output like this:

```
Job: job-0000000003
Job: job-0000000002
Job: job-0000000001
Job: job-0000000000
Job: job-0000000007
Job: job-0000000006
Job: job-0000000005
Job: job-0000000004
Job: job-0000000009
Job: job-0000000008
```

Note that Curator does not do anything special with the `getChildren` call, so it has the same property of returning child nodes in an arbitrary order as the native ZooKeeper client. To implement a FIFO (first in, first out) queue, the `List` returned must be sorted before processing.

The `Stat` object can also be used in conjunction with other commands that read data from ZooKeeper, such as the `getData` and `getChildren` command using the `storeStatIn` method. This method takes a `Stat` object whose data will be populated by the command when it runs. For example, when retrieving the list of children from the `/queue` path as above:

```
Stat stat = new Stat();
List<String> jobs = client.getChildren()
  .storingStatIn(stat)
  .forPath("/queue");
  for(String job : jobs)
    System.out.println("Job: "+job);
System.out.println("Number of children: "
  +stat.getNumChildren());
```

To remove a path, use the `delete` command. Like the native ZooKeeper client, this command can be set to act on a specific version with the `withVersion` command. This example checks to see if the path exists and then deletes the specific version if it does. It will also delete any children the `znode` may have:

```
Stat stat;
if((stat = client.checkExists().forPath("/a/test/path")) != null)
  client.delete()
    .deletingChildrenIfNeeded()
    .withVersion(stat.getVersion())
    .forPath("/a/test/path");
```

Using Watches with Curator

The `checkExists`, `getData`, and `getChildren` commands support having watches set on them via the `usingWatch` method. This method takes, as an argument, an object implementing either the native ZooKeeper `Watcher` interface or the Curator-specific `CuratorWatcher` interface. The `Watcher` interface has been described previously, and the `CuratorWatcher` is essentially identical. For example, the following code uses a `CuratorWatcher` to observe changes to the existence of a specific path:

```
client.checkExists().usingWatcher(new CuratorWatcher() {
  public void process(WatchedEvent arg0) throws Exception {
    if (arg0.getType() == Watcher.Event.EventType.None) {
      //State change
    } else {
      System.out.println(arg0.getType());
      System.out.println(arg0.getPath());
    }
  }
}).forPath("/a/test/path");
```

Curator Recipes

In addition to implementing a robust and easy-to-use ZooKeeper client, Curator goes the extra mile by providing implementations of many of the recipes found on the ZooKeeper site.

The recipes as given in this section are essentially skeletons, primarily provided to give clear examples of working with ZooKeeper in common situations. The Curator recipes take these basic skeletons and make them robust, and often more feature-rich, for use in production environments. They provide an excellent starting point for implementing high-level logic based on these common algorithmic structures.

This section covers using some of the various recipes, which includes everything from the recipes section of the ZooKeeper website except the Multi-Version Concurrency Control implementation. In some cases, such as the Distributed Queue implementation, extensions to the basic algorithm not found on the ZooKeeper site are added.

Distributed Queues

The Curator website has a document called "Technical Note 4" that states that ZooKeeper is a terrible system for implementing queues. The reasons given include ZooKeeper's relatively small node payload, the need to keep the entire dataset in memory, and performance issues when a `znode` has a very large number of children.

In one sense, this is entirely correct. ZooKeeper would be a very poor choice for handling queues in an environment with a high message volume. A dedicated queue management system such as ActiveMQ or RabbitMQ would be more appropriate in that situation. In a very high-volume environment, a data motion system like those discussed in the next chapter is even more appropriate. However, for something like a job queue where relatively few items are parceled out to worker nodes, ZooKeeper can be a good and fairly lightweight solution.

In this latter situation, Curator provides several different distributed queue options, accessible via the `QueueBuilder` object. As an example, consider a simple distributed queue that needs to handle `WorkUnit` objects. This simple class serves an example of an object that contains the information needed to perform some task. In this case, it only contains an identifier and a payload string:

```
public class WorkUnit implements Serializable {
  private static final long serialVersionUID = -248144165425681310 1L;
  long   workId;
  String payload;
  public WorkUnit() { }
  public WorkUnit(long workId,String payload) {
    this.workId = workId;
    this.payload = payload;
  }
  public String toString() {
    return "WorkUnit<"+workId+","+payload+">";
  }
}
```

To define a producer of `WorkUnit` objects, begin by defining a `DistributedQueue` object and creating the Curator client. In this case, the client is defined using a connection string and a single shot retry policy. The client is then started to begin processing ZooKeeper messages:

```
public class DistributedQueueProducer implements Runnable {
  DistributedQueue<WorkUnit> queue;
  public DistributedQueueProducer(String connectString)
    throws Exception {
    CuratorFramework client = CuratorFrameworkFactory.newClient(
      connectString,
      new RetryOneTime(10)
    );
    client.start();
```

Next, the queue itself is created. The `QueueBuilder`'s builder method always takes four arguments: the Curator client, a `QueueConsumer<T>` object, a `QueueSerializer<T>` object, and a path. In this case, the thread acts only as a Producer, so the `QueueConsumer` object can be

left as `null`. Then, a `DistributedQueue` object is built and started. Like the client itself, the queue must be started before it can be used:

```
queue = QueueBuilder.builder(client,
  null,
  new SimpleSerializer<WorkUnit>(),
  "/queues/work-unit")
.buildQueue();
queue.start();
```

Curator does not ship with any built-in `QueueSerializer` implementations. It is left entirely to the application to implement an appropriate serializer for each queue. In this case, the `WorkUnit` implements the `Serializable` interface to simple Java. Serialization is used to convert the `WorkUnit` object to and from a byte array. The `SimpleSerializer` implements a serializer that can be used for any class that implements `Serializable`:

```
public class SimpleSerializer<T extends Serializable> implements
             QueueSerializer<T> {
public byte[] serialize(T item) {
  ByteArrayOutputStream bos = new ByteArrayOutputStream();
  byte[] value = null;
  try {
    ObjectOutput out = new ObjectOutputStream(bos);
    out.writeObject(item);
    value = bos.toByteArray();
    out.close();
  } catch(IOException e) { }
  try {
    bos.close();
  } catch(IOException e) { }
    return value;
  }
  @SuppressWarnings("unchecked")
  public T deserialize(byte[] bytes) {
    ByteArrayInputStream bin = new ByteArrayInputStream(bytes);
    Object object = null;
    try {
      ObjectInput in  = new ObjectInputStream(bin);
      object = in.readObject();
      in.close();
    } catch(IOException e) {
    } catch (ClassNotFoundException e) { }
    try {
      bin.close();
    } catch(IOException e2) { }
      return (T)object;
  }
}
```

Now, all that remains is to start adding items to the queue. This simple example implements Runnable and a simple run loop that adds items to the queue after a random wait of up to 2 seconds:

```
Random rng = new Random();
public void run() {
  System.out.println("Starting Producer");
  boolean cont = true;
  long id = 0;
  while(cont) {
```

```
      try {
        queue.put(new WorkUnit(id++,"Next Work Unit "+id));
        System.out.println("Added item");
      } catch (Exception e1) {
        e1.printStackTrace();
      }
      try {
        Thread.sleep(rng.nextInt(1000));
      } catch (InterruptedException e) {
        cont = false;
      }
    }
  }
```

Next is the `QueueConsumer` implementation. In this case, to maintain some symmetry with the producer, the `QueueConsumer` implementation also handles the creation of the `DistributedQueue` implementation. It starts very much like the producer implementation:

```
  public class DistributedQueueConsumer
    implements QueueConsumer<WorkUnit> {
    DistributedQueue<WorkUnit> queue;
    String name;
    public DistributedQueueConsumer(
      String connectString,
      String name) throws Exception {
      this.name = name;
      CuratorFramework client = CuratorFrameworkFactory.newClient(
        connectString,
        new RetryOneTime(10)
      );
      client.start();
      queue = QueueBuilder.builder(
        client,
        this,
        new SimpleSerializer<WorkUnit>(),
        "/queues/work-unit"
      )
      .buildQueue();
      queue.start();
    }
```

The only difference is that, rather than passing `null` as the `QueueConsumer`, the consumer passes itself. It also implements the `QueueConsumer` interface, which requires the implementation of two methods. In this case, the "processing" of a queue element simply emits the information found in the `WorkUnit` object:

```
  public void stateChanged(CuratorFramework arg0, ConnectionState arg1) {
    System.out.println(arg1);
  }
  public void consumeMessage(WorkUnit message) throws Exception {
    System.out.println(name
      +" consumed "+message.workId+"/"+message.payload);
  }
```

Finally, here is a simple process that can be created that starts two queue consumers to handle queue items and a queue producer to create new items:

```
  DistributedQueueConsumer q1 = new DistributedQueueConsumer(
    "localhost","queue 1"
```

```
);
DistributedQueueConsumer q2 = new DistributedQueueConsumer(
  "localhost","queue 2"
);
new Thread(new DistributedQueueProducer("localhost")).run();
```

When this process starts running, it first consumes any items left over in the queue from previous executions. It then starts producing and consuming items. Note that the items are assigned to arbitrary queue consumers depending on the watch that happens to get triggered by the ZooKeeper server:

```
Starting Producer
Added item
queue 2 consumed 0/Next Work Unit 1
Added item
queue 2 consumed 1/Next Work Unit 2
Added item
queue 2 consumed 2/Next Work Unit 3
Added item
queue 1 consumed 3/Next Work Unit 4
Added item
queue 2 consumed 4/Next Work Unit 5
Added item
queue 2 consumed 5/Next Work Unit 6
Added item
queue 1 consumed 6/Next Work Unit 7
Added item
queue 1 consumed 7/Next Work Unit 8
Added item
queue 2 consumed 8/Next Work Unit 9
Added item
queue 2 consumed 9/Next Work Unit 10
Added item
queue 2 consumed 10/Next Work Unit 11
Added item
queue 1 consumed 11/Next Work Unit 12
```

Leader Elections

Curator provides two recipes for implementing Leader Elections: a `LeaderSelector` class and a `LeaderLatch` class. The `LeaderSelector` class uses a callback mechanism to implement its functionality, whereas the `LeaderLatch` class uses a process similar to the earlier Leader Election example using the native Java API.

To use the `LeaderSelector`, the server trying to obtain leadership of the process must implement the `LeaderSelectorListener` interface. This interface specifies the `takeLeadership` method, which serves as the main method for the server. Returning from this method relinquishes the leadership position. For example, a dummy unreliable server might look like this:

```
public class UnreliableLeader implements LeaderSelectorListener {
  String  name   = "";
  boolean leader = false;
  String  connect;
  public UnreliableLeader(String connect,String name) {
    this.connect = connect;
    this.name = name;
  }
  public void stateChanged(CuratorFramework arg0,
    ConnectionState arg1) {
    if(arg1 != ConnectionState.CONNECTED) leader = false;
```

```
    }
    public void takeLeadership(CuratorFramework client) throws Exception {
      leader = true;
      Random rng = new Random();
      while(leader) {
        if(rng.nextDouble() < 0.30) {
          System.out.println(name+": crashing");
          return;
        }
        System.out.println(name+": processing event");
        Thread.sleep(rng.nextInt(1000));
      }
    }
  }
```

The `LeaderLatch` client takes a different approach, offering a `hasLeadership` method to check to see if the server still has leadership and an `await` method to wait for leadership. Using the `LeaderLatch` looks something like this:

```
LeaderLatch latch = new LeaderLatch(client,"/leader_queue");
latch.start();
latch.await();
while(latch.hasLeadership()) {
 //Do work here
}
```

The `wiley.streaming.curator.LeaderSelectionTest` uses the `UnreliableLeader` class to start three different leaders that then take over processing events from each other when one crashes:

```
Thread[] thread = new Thread[3];
for(int i=0;i<thread.length;i++) {
  UnreliableLeader l = new UnreliableLeader(
    "localhost","server"+i);
  thread[i] = new Thread(l);
  thread[i].run();
}
for(int i=0;i<thread.length;i++) {
  thread[i].join();
}
```

The output from this example looks something like this:

```
server0 is about to start waiting for leadership
server0: processing event
server0: processing event
server0: processing event
server0: processing event
server0: crashing
server1 is about to start waiting for leadership
server1: crashing
server2 is about to start waiting for leadership
server2: processing event
server2: processing event
server2: processing event
server2: crashing
```

Conclusion

This chapter has introduced the concepts behind maintaining shared state and coordinating between distributed processes. These tasks have been handled in an ad hoc manner by many distributed systems over the years, most typically by using a relational database. The relational database option generally works, but it introduces the potential for error when trying to implement a do-it-yourself distributed lock manager or other system.

To combat that problem, Yahoo! and other companies began to develop coordination tools that act as a service rather than as a component of a piece of software. This resulted in the development of ZooKeeper and its eventual contribution to the Apache project. The success of this approach is evident: Most of the distributed software used in this book uses ZooKeeper for configuration and coordination. Other systems, such as etcd, are now being developed, but ZooKeeper remains the most popular by far.

The next few chapters of this book put ZooKeeper to work by discussing distributed systems. The heaviest user is the Kafka project, a data motion system discussed in Chapter 4. It is also heavily used to coordinate stream-processing applications such as Storm and Samza, both of which use the project.

ZooKeeper is also useful in end-user applications. It can help to manage a distributed state such as data schemas. Additionally, it can provide controlled access to limited shared resources as needed. Its versatility and usefulness is why it was presented here in such detail.

Chapter 4
Data-Flow Management in Streaming Analysis

Chapter 3, “Service Configuration and Coordination,” introduces the concept and difficulties of maintaining a distributed state. One of the most common reasons to require this distributed state is the collection and processing of data in a scalable way.

Distributed data flows, which include processing and collection, have been around a long time. Generally, the systems designed to handle this task have been bespoke applications developed either in-house or through consulting agreements. More recently, the technologies used to implement these data flow systems has reached the point of common infrastructure. Data flow systems can be split into a separate service in much the same way that coordination and configuration can. They are now general enough in their interfaces and their assumptions that they can be used outside of their originally intended applications.

The earliest of these systems were arguably the queuing systems, such as ActiveMQ, which started to come onto the scene in the early 2000s. However, they were not really designed for high-throughput volumes (although many of them can now achieve fairly good performance) and tended to be very Java centric.

The next systems on the scene were those open-sourced by the large Internet companies such as Facebook. One of the most well-known systems of this generation was a tool called Scribe, which was released in 2008. It used an RPC-like mechanism to concentrate data from edge servers into a processing framework like Hadoop. Scribe has many of the same features of the current generation, including the ability to spool data to disk, but it can only account for intermittent connectivity failures.

Flume, developed by Cloudera, and Kafka are the current generation of distributed data collection systems, and they represent two entirely separate philosophies. This chapter discusses the care and feeding of both of these data motion systems. In addition, there is some discussion of their underlying philosophies to help make the decision about which system to use in which situation. However, these two data motion systems should not be considered to be mutually exclusive. There is no reason that the two cannot both be used in a single environment according to need.

Distributed Data Flows

A distributed data flow system has two fundamental properties that should be addressed. The first is an “at least once” delivery semantic. The second is solving the “n+1” delivery problem. Without these, a distributed data flow will have difficulty successfully scaling. This section covers these two components and why they are so important to a distributed data flow.

At Least Once Delivery

There are three options for data delivery and processing in any sort of data collection framework:

- At most once delivery
- At least once delivery
- Exactly once delivery

Many processing frameworks, particularly those used for system monitoring, provide “at most once” delivery and processing semantics. Largely, this is because the situations they were designed to handle do not require all the data be transmitted, but they do require maximum performance to alert administrators to problems. In fact, many of these systems down-sample the data to further improve performance. As long as the rate of data loss is approximately known, the monitoring software can recover a usable value during processing.

In other systems—for instance financials systems or advertising systems where logs are used to determine fees—every lost data record means lost revenue. Furthermore, audit requirements often mean that this data loss cannot be estimated through techniques used in the monitoring space. In this case, most implementations turn to “exactly once” delivery through queuing systems. Popular examples include the Apache project's ActiveMQ queuing system, as well as RabbitMQ, along with innumerable commercial solutions. These servers usually implement their queue semantics on the server side, primarily because they are usually designed to support a variety of producers and consumers in an Enterprise setting.

The “exactly once” delivery mechanism, of course, sacrifices the raw performance of the “at most once” delivery mechanism in an effort to provide this safety whether or not it is required. The “at least once” delivery system tries to balance these two extremes by providing reliable message delivery by pushing the handling semantics to the consumer. By doing this, the consumer can use the delivered data stream to construct the semantics required by the application without affecting other consumers of the data stream. For example, a consumer requiring “exactly once” processing for something such as financial transactions between two accounts can implement a de-duplication procedure to ensure that messages are not reprocessed. Another consumer who cares only about tracking something like the number of unique account transaction pairs does not need “exactly once” delivery because it uses idempotent set operations, which means it does not need to pay the management penalty. The general theory is that appropriate message handling, beyond ensuring that they are delivered at least once, is dependent on the application logic and should be handled at that level.

The “n+1” Problem

In “traditional” log processing systems, the architecture is essentially a funnel. The data from a potentially large number of edge systems is collected to a small number of central locations (usually one, but legal or physical restrictions sometime necessitate more than one processing location). These

processing locations then manipulate the data and perhaps move it along to another location, such as a data warehouse, for further analysis.

After this first funnel is in place, the inevitable happens: A second data consumer needs to be added, resulting in the construction of another funnel. After that, perhaps another front-end service is introduced that has its own data collection mechanism. Eventually, every time a new service or processing mechanism is added it must integrate with each of the other systems and, even worse, they must integrate with it. It may seem like an exaggeration, but this is actually a fairly common antipattern in the industry.

In the Enterprise space, this problem manifested itself in the form of the service-oriented architecture (SOA) and saw the development of the idea of the enterprise service bus (ESB). The idea was that the bus would handle the interaction with various services, which were implemented independently with no particular common protocol, and standardize the communication between them. In practice, they are most commonly used to move data between different "silos" within an organization, for example, between the sales organization's Salesforce implementation and the analytics team's in-house databases.

For the data flow tools discussed in this chapter, the opposite happens. In this case, the communication between the bus layer and each application is standardized, and the messaging system is primarily responsible for managing the physical flow between systems. This allows any number of producers and consumers to communicate through the common mechanism of the data flow protocol without having to worry about other producers and consumers.

Apache Kafka: High-Throughput Distributed Messaging

The Apache Kafka project was developed by LinkedIn to connect its website services and its internal data warehouse infrastructure. It was open-sourced in 2011 as a 0.6 release and accepted as an Apache Incubator project later that year. The current release of Kafka is 0.8, which added many new features, including a new producer message application programming interface (API) and internal replication. This section discusses the 0.8 series of releases, though the simpler 0.7.2 release is often still used by organizations because third-party clients are more plentiful for this version.

Design and Implementation

Kafka was purpose-built to solve specific problems for a specific company—in this case LinkedIn. These problems were a high message volume and a large number of disparate services that needed to communicate. As of late 2013, and excluding messaging between datacenters that are mirroring data, LinkedIn reportedly processed approximately 60 billion messages per day. It also had a relatively large number of services that needed to be integrated, quite possibly hundreds of different producers and consumers. LinkedIn also had essentially complete control over their environment and software, allowing for the construction of a very opinionated data motion environment.

As it happens, this sort of environment is not unique to LinkedIn. Many companies that deal primarily with "Internet data" find themselves in the same situation. Additionally, many of them are engineering focused, meaning that most of their software is developed in-house rather than licensed from a third party. This allows the companies to use the Kafka model, and it is useful enough that a similar system, called Kinesis, was recently announced by Amazon.com. This product aims to make up a core part of the integration between various Amazon.com services, such as its key-value store Dynamo, its block storage engine S3, its Hadoop infrastructure Elastic MapReduce, and its high-performance data warehouse Redshift.

This section covers the design of Kafka's internals and how they integrate to solve the problems mentioned here.

Topics, Partitions, and Brokers

The organizing element of Kafka is the "topic." In the Kafka system, this is a physical partitioning of the data such that all data contained within the topic should be somehow related. Most commonly, the messages in this topic are related in that they can be parsed by the same common mechanism and not much else.

A topic is further subdivided into a number of partitions. These partitions are, effectively, the limit on the rate that an I/O-bound consumer can retrieve data from Kafka. This is because clients often use a single consumer thread (or process) per partition. For example, with Camus, a tool for moving data from Kafka into the Hadoop Distributed File System (HDFS) using Hadoop, a Mapper can pull from multiple partitions, but multiple Mappers will not pull from the same partition.

Partitions are also used to logically organize a topic. Producer implementations usually provide a mechanism to choose the Kafka partition for a given message based on the key of that message.

Partitions themselves are distributed among brokers, which are the physical processes that make up a Kafka cluster. Typically, each broker in the cluster corresponds to a separate physical server and manages all of the writes to that server's disk. The partitions are then uniformly distributed across the different brokers and, in Kafka 0.8 and later, replicas are distributed across other brokers in the

cluster.

If the number of brokers changes, partitions and their replicas can be reassigned to other brokers in the cluster. When consuming from a topic, the consumer application, or consumer group if it is a distributed application, will assign a single thread or process to each partition. These independent threads then process each partition at their own pace, much like the Map phase of a Map-Reduce application. Kafka's high-level consumer implementation tracks the consumption of the various threads, allowing processing to be restarted if an individual process or thread is interrupted. While this model can be circumvented to improve consumption using Kafka's low-level interfaces, the preferred mechanism is to increase the number of partitions in a topic as necessary. To accomplish this task, Kafka provides tools to add partitions to existing topics in a live environment.

Log-Structured Storage

Kafka is structured around an append-only log mechanism similar to the write-ahead-log protocol found in database applications. The write-ahead-log—which was apparently first developed by Ron Obermark in 1974 while he was working on IBM's System R—essentially says that before an object can be mutated, the log of its mutation must first have been committed to a recovery log. This forms an "undo" log of message mutations that can be applied or removed from the database to reconstruct the state of the database at any particular moment in time.

To accomplish this, the recovery log is structured such that with every change it assigned a unique and increasing number (what mathematicians would call a strictly monotonically increasing sequence) and appended to the end of a theoretically infinite file. Often, this increasing number is the position of the record within the file because it is usually easy to obtain this information when appending to a file. Of course, in a practical system, no file can be infinitely large, so after a file reaches its maximum size, a new file is created and the offsets are reset. If the files are also named using a strictly monotonically increasing sequence, the semantics of the write-ahead-log is completely maintained.

This approach is not only simple, but it also maximizes the performance of the storage media most often used to maintain these logs. Back when they were first being developed, write-ahead-logs would have been written to tape storage media (each tape holding approximately 50MB of data). Tape, of course, works best with sequential writes. Even modern spinning media—hard drives—usually exhibit far superior sequential read/write performance relative to random reads and writes.

Kafka uses essentially the same system as the write-ahead-log, maintaining separate logs for each partition of each topic in the system. Of course, Kafka is not a database. Rather than mutating a record in a database, Kafka does not make a message available to consumers until after it has been committed to the log, as shown in Figure 4.1. In this way, no consumer can consume a message that could potentially be lost in the event of a broker failure.

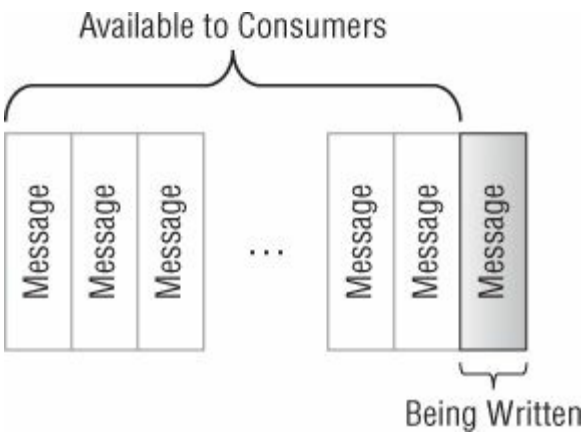


Figure 4.1

In a post to LinkedIn's developer blog, Jay Kreps, one of Kafka's developers, made an offhand comment that describes that Kafka implements "logs as a service." This could be taken quite literally, and Kafka could be used to, essentially, implement a traditional relational database management system (RDBMS) replication system. To do this, changes to the table in the database would be written to Kafka using the normal producer protocol. The database itself would be a consumer of the Kafka data (in 0.8 this can be accomplished using producer callbacks with the producer response set to "all" replicas) and applies changes made to the tables by reading from Kafka. To improve recovery times, it would occasionally snapshot its tables to disk along with the last offset to be applied from each partition. It would appear that Kafka's developers are also contemplating this style of application. The log compaction proposal presented at https://cwiki.apache.org/confluence/display/KAFKA/Log+Compaction indicates an environment where the topic would be long-lived and allows consumers to recover the most recent value for a given key. In a database application, this would cover recovery and replication use cases. The key in a database application would represent the primary keys of tables with the consumer application responsible for mutating a local copy into a format more suited to querying (such as B-Tree).

Space Management

Although disk space is inexpensive, it is neither free nor infinite. This means that logs must eventually be removed from the system. Kafka, unlike many systems, uses a time-based retention mechanism for its logs. This means that logs, usually grouped into 1GB files unless otherwise specified, are removed after a certain number of hours of retention.

This means that Kafka happily uses its entire disk and does have a failure mode in the 0.7.x series; it continues to accept messages despite being unable to successfully persist them to non-volatile storage. At first glance, this seems like a recipe for disaster. However, it is generally much easier to manage available storage than it is to manage potential slow clients or requests for older data. Space management is generally a built-in component of core system monitoring and is easily remedied via the introduction of more disk per broker or adding brokers and partitions.

Not a Queuing System

Kafka is very explicitly not a queuing system like ActiveMQ or RabbitMQ, which go through a lot of trouble to ensure that messages are processed in the order that they were received. Kafka's partitioning system does not maintain this structure. There is no defined order of writes and reads to partitions of a particular topic, so there is no guarantee that a client won't read from partitions in a

different order than they were written. Additionally, it is not uncommon to implement producers asynchronously, so a message sent to one partition may be written after a message sent to another partition despite happening first due to differences in latency or other nondeterministic events.

This is generally fine because, in many Internet applications, maintaining explicit ordering is overkill. The order of events is essentially arbitrary to begin with, so there is no real benefit to preserving one particular arbitrary order over another arbitrary order. Furthermore, the events where ordering is important usually occur far enough apart that they still appear to be ordered by any processing system.

Kafka also differs from many queuing systems in how it deals with message consumers. In many queuing systems, messages are removed from the system when they have been consumed. Kafka has no such mechanism for removing messages, instead relying on consumers to keep track of the offset of the last message consumed. Although the high-level consumer shipped with Kafka simplifies this somewhat by using ZooKeeper to manage the offset, the brokers themselves have no notion of consumers or their state.

Replication

With version 0.8, Kafka introduced true replication for a broker cluster. Previously, any notion of replication had to be handled via the cluster-to-cluster mirroring features discussed in the next section. However, this is merely a backup strategy rather than a replication strategy.

Kafka's replication is handled at the topic level. Each partition of a topic has a leader partition that is replicated by zero or more follower partitions (in Kafka 0.8, unreplicated topics are simply leaders without followers). These follower partitions are distributed among the different physical brokers with the intent to place each follower on a different broker. Implementations should ensure that they have sufficient broker capacity to support the number of partitions and replicas that will be used for a topic.

When a partition is created, all of these followers are considered to be “in-sync” and form the in-sync replica set (ISR). When a message arrives at the leader partition to be written, it is first appended to the leader's log. The message is then forwarded to each of the follower partitions currently in the ISR. After each of the partitions in the ISR acknowledges the message, the message is considered to be committed and can now be read by consumers. The leader also occasionally records a high watermark containing the offset of the most recently committed message and propagates it to the follower partitions in the ISR.

If the broker containing a particular replica fails, it is removed from the ISR and the leader does not wait for its response. This is handled through Kafka's use of ZooKeepers to maintain a set of “live” brokers. The leader then continues to process messages using the smaller pool of replicas in the ISR. When the replica recovers, it examines its last-known high watermark and truncates its log to that partition. The replica then copies data from this position to the current committed offset of the leader. After the replica has caught up, it may be added back to the ISR and everything proceeds as before.

The addition of replication to Kafka also introduced some changes to the Kafka Producer API. In versions of Kafka prior to 0.8 there was no acknowledgement in the Producer API. Applications wrote to the Kafka socket and hoped for the best. In Kafka 0.8, there are now three different levels of acknowledgements available: none, leader, and all.

The first option, none, is the same as in Kafka 0.7 and earlier and no response is returned to producer. This is the least-durable situation and allows data to be lost, but it affords maximum performance that

can be easily measured into the tens of thousands of messages per second.

The second option, leader, sends an acknowledgement after the leader has received the message but before it has received acknowledgements from the ISR. This reduces performance somewhat and can still lead to data loss, but this option offers a reasonable level of durability for most applications.

The final option, all, sends the acknowledgement only after the leader has committed the message. In this situation, the data is not lost so long as at least one partition remains in the ISR. However, the performance reduction relative to the none case is significant, though much of this can be recovered with a large number of partitions and a highly concurrent Producer implementation.

Multiple Datacenter Deployments

Many web applications are latency sensitive, requiring them to be geo- distributed around the globe. The connections between these far-flung datacenters are, unsurprisingly, less reliable than connections within a datacenter. Kafka helps to deal with potential (and depressingly common) increased latency and complete connection loss between datacenters by providing built-in mirroring tools. Using these tools, a Kafka cluster is established in each datacenter with a retention time designed to balance the need to cover an extended outage, and the available space in the remote datacenter. If enough space is available, a longer retention time can be used as a guard against disaster.

These remote clusters are then copied into the main processing cluster using a tool called MirrorMaker. This tool, which is shipped with Kafka, can read from multiple remote clusters and writes the messages there into a single output cluster. Writes to the same topic in each cluster are merged into a single topic on the output cluster.

In addition to remote datacenter mirroring, the mirror facilities are also useful for development. A remote cluster can simply be mirrored into the development cluster to allow development using production data without risking production environments.

Configuring a Kafka Environment

This section describes how to start a Kafka environment. On development systems, a single broker is usually sufficient for testing whereas a larger system should probably have a minimum of three to five brokers depending on replication requirements. On the top end, quite large installations are possible. Chapter 5, “Processing Streaming Data,” covers a tool called Samza, which uses Kafka for data processing in the same way that Hadoop Map-Reduce uses its distributed filesystem. In this instance, there could be dozens of brokers in the cluster that are all running Kafka.

This section covers everything needed to get Kafka up and running. There is also a small section on developing multi-broker applications on a single machine by using separate configurations.

Installing Kafka

Installing Kafka is simply a matter of unpacking the current version of Kafka, 0.8.0 at the time of writing, and configuring it. Kafka itself is largely written in a language called Scala, which has several versions. To avoid confusion between versions, the Kafka developers choose to include the build version of Scala in the archive name. This is mostly important for developers who are trying to use the libraries included in the archive for development. Currently, Kafka's binary distribution is built using Scala 2.8.0, so the archive is named `kafka_2.8.0-0.0.tar.gz`. You can download

it from http://kafka.apache .org and then unpack it:

```
$ tar xvfz ~/Downloads/kafka_2.8.0-0.8.0.tar.gz
x kafka_2.8.0-0.8.0/
x kafka_2.8.0-0.8.0/libs/
x kafka_2.8.0-0.8.0/libs/slf4j-api-1.7.2.jar
x kafka_2.8.0-0.8.0/libs/zkclient-0.3.jar
x kafka_2.8.0-0.8.0/libs/scala-compiler.jar
x kafka_2.8.0-0.8.0/libs/snappy-java-1.0.4.1.jar
x kafka_2.8.0-0.8.0/libs/metrics-core-2.2.0.jar
x kafka_2.8.0-0.8.0/libs/metrics-annotation-2.2.0.jar
x kafka_2.8.0-0.8.0/libs/log4j-1.2.15.jar
x kafka_2.8.0-0.8.0/libs/jopt-simple-3.2.jar
x kafka_2.8.0-0.8.0/libs/slf4j-simple-1.6.4.jar
x kafka_2.8.0-0.8.0/libs/scala-library.jar
x kafka_2.8.0-0.8.0/libs/zookeeper-3.3.4.jar
x kafka_2.8.0-0.8.0/bin/
[ More Output Omitted ]
$
```

Most operating system configurations limit the number of open files allowed by a process to a relatively small number, such as 1024. Due to a number of factors, such as the number of topics, partitions, and retention time, it is possible that Kafka will need to consume many more file handles than this under normal operation. Increasing this limit depends on the operating system, but on Linux it involves editing the `/etc/security/limits.conf` file. For example, adding the following `nofile` (it stands for "number of files" rather than "no files") line increases the allowable open files for the `kafka` user to 50,000:

```
kafka             -    nofile          50000
```

Kafka Prerequisites

Kafka's only external dependency is a ZooKeeper installation. In a production environment, refer to the installation guide in the previous chapter to get a ZooKeeper cluster running. For local development environments, Kafka ships with a preconfigured ZooKeeper server that can be used to host several brokers on a single machine.

To start this development version, Kafka includes a script `bin/zookeeper-server-start.sh` and a configuration file `config/zookeeper.properties`. You can use these as they are to start the server in a terminal window:

```
$ cd kafka_2.8.0-0.8.0/
$ ./bin/zookeeper-server-start.sh config/zookeeper.properties
INFO Reading configuration from:
  config/zookeeper.properties
  (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
WARN Either no config or no quorum defined
  in config, running  in standalone mode
  (org.apache.zookeeper.server.quorum.QuorumPeerMain)
INFO Reading configuration from:
  config/zookeeper.properties
  (org.apache.zookeeper.server.quorum.QuorumPeerConfig)
INFO Starting server
  (org.apache.zookeeper.server.ZooKeeperServerMain)
[ Other Output Omitted ]
```

After ZooKeeper has been started, all that remains is to configure and start the Kafka brokers.

Configuring the Broker

The Broker is the name of the Kafka server. Like ZooKeeper, it is configured with a simple properties file that defines the broker itself. Information about the topics and offset data is kept either with the data itself or in the ZooKeeper cluster. This section walks through this properties file and explains each of the settings and their effect on the cluster.

After some boilerplate containing the appropriate Apache licensing information, the first setting is the broker identifier. Unlike ZooKeeper, where the id is limited to 0-255, Kafka represents the broker as a unique unsigned 32-bit integer (internally, this is an encoded as a Java long value, but the value must be positive, which limits the effective range to 32 bits):

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

If you're using IPv4 addressing, an option for assigning unique broker ids is to use the IPv4 address converted to an unsigned 32-bit number. Many programming languages have a function called `inet_aton` that is used to convert a "dotted quad" IP address to an unsigned 32-bit integer, but this can also be calculated from the Linux Bash shell using the following one-liner, assuming that the `bc` utility is installed:

```
$ IP=$(ip=`hostname -I` && ip=( ${ip//\./ } ) \
>  && echo "(${ip[0]} * (2^24)) + (${ip[1]}*(2^16)) \
>  + (${ip[2]}*(2^8)) + ${ip[3]}" | bc) \
> && echo "broker.id: $IP"
```

This command uses the Linux `hostname` command to obtain the "default" IP address using the `-I` option. This is broken up into its component parts by `ip=(${ip//\./ })`. Note that the trailing space after the "/" character is important and cannot be omitted. These component parts are then bit-shifted by multiplying them appropriately using the `bc` command.

The `broker.id` value should only be set during initial configuration and the IP address is used only to ensure unique IDs during cluster setup. If the server receives a new IP address due to being restarted, it should retain its current `broker.id`. Of course, problems can arise when new servers are added and receive an IP address that had been assigned during initial cluster configuration.

The next parameter is the port the broker uses for communication. The default is 9092 and there is little reason to change it unless multiple broker instances are to be run on the same machine (see the A Multi-Broker Cluster on a Single Machine section):

```
# The port the socket server listens on
port=9092
```

The `host.name` option is used to set the host name reported by the broker. In most circumstances, this can be left commented out because `getCanonicalHostName()` does the correct thing. If the compute environment is somewhat exotic and a server has multiple hostnames, it may be necessary to set this explicitly. One example of this is Amazon's EC2 environment. Each server in EC2 has at least two fully qualified domain names—an internal and an external domain—and may have other names assigned using Amazon's Route 53 DNS service. Depending on which services need to communicate with the cluster, the default name may not be appropriate to report. If this problem cannot be overcome at the operating system level, it is possible to explicitly define the name here so that metadata calls return the correct hostname.

```
# Hostname the broker will bind to and
# advertise to producers and consumers.
# If not set, the server will bind to all
# interfaces and advertise the value returned from
# from java.net.InetAddress.getCanonicalHostName().
#host.name=localhost
```

These next two options control the number of threads that Kafka uses for processing network requests and managing the log files. In older versions of Kafka, there was only a single option to set the number of threads `kafka.num .threads`. The new setting allows for greater flexibility in tuning the Kafka cluster. The default value for `num.network.threads` is 3 and for `num.io.threads` it is 8. The recommendation for `num.network.threads` of 3 threads is probably sufficient for most use cases. For `num.io.threads`, the recommendation is that the number of threads equal the number of disks used for log storage.

```
# The number of threads handling network requests
num.network.threads=3
# The number of threads doing disk I/O
num.io.threads=8
# The number of requests that can be queued for I/O before network
# threads stop reading
#queued.max.requests=
```

The next settings control the size of the socket buffers and requests. For multi-datacenter transfers, the LinkedIn reference configuration increases these values to 2MB from the following 1MB configuration. If these settings are commented out, the default is a relatively small 100KB buffer size, which may be too small for a high-volume production environment. It is usually not necessary to modify the maximum request size, which is set to 100MB in this case:

```
# The send buffer (SO_SNDBUF) used by the socket server
socket.send.buffer.bytes=1048576
# The receive buffer (SO_RCVBUF) used by the socket server
socket.receive.buffer.bytes=1048576
# The maximum size of a request that the socket server
# will accept (protection against OOM)
socket.request.max.bytes=104857600
```

The `log.dirs` property controls Kafka's log output location. These locations contain a number of partition directories of the form `<topic name>- <partition number>` and a file called `replication-offset-checkpoint`:

```
$ ls
analytics-0  audit-1  replication-offset-checkpoint
```

This text file contains the “high watermark” data described in the earlier replication section and should not be deleted. If multiple comma-separated directories are given to Kafka, it distributes partitions evenly across each of the directories. This is a tempting option for servers where JBOD (Just a Bunch of Disks) configurations are the default, such as Amazon's EC2. Unfortunately, Kafka simply distributes partitions randomly to the various disks rather than trying to, say, distribute partitions of the same topic across multiple disks. It is often the case that topics have very different data sizes, and by luck it can happen that different disks have very different usage characteristics. If possible, it is usually better to use RAID0 to combine the disks into a single large disk and use a single log directory.

```
# A comma seperated list of directories under which to store log files
log.dirs=/tmp/kafka-logs
```

These next settings control the creation and management of topics.

```
# Disable auto creation of topics
#auto.create.topics.enable=false
# The number of logical partitions per topic per server.
# More partitions allow greater parallelism
# for consumption, but also mean more files.
num.partitions=2
```

The default replication factor in a Kafka cluster is 1, meaning that only the leader has the data. You can manually set the replication factor on a per-topic basis by explicitly creating the topic, but this replication factor will be used for all auto-created topics (if auto-creation is enabled).

```
# Set the default replication factor for a topic
#default.replication.factor=3
```

If a replica is being used, the next few parameters control the maintenance of ISR sets. In general, there is no real reason to change most of these defaults. (You should use care if you do change them.) In particular, setting the lag time or messages to be too small can make it impossible for replicas to remain in the ISR. The one setting that can sometimes improve performance is increasing `num.replica.fetchers` to improve replication throughput.

```
# The maximum time a leader will wait for a fetch request before
# evicting a follower from the ISR
#replica.lag.time.max.ms=10000
# The maximum number of messages a replica can lag the leader before it
# is evicted from the ISR
#replica.lab.max.messages=4000
# The socket timeout for replica requests to the leader
#replica.socket.timeout.ms=30000
# The replica socket receive buffer bytes
#replica.socket.receive.buffer.bytes=65536
# The maximum number of bytes to receive in each replica fetch request
#replica.fetch.max.bytes=1048576
# The maximum amount of time to wait for a response from the leader
# for a fetch request
#replica.fetch.wait.max.ms=500
# The minimum number of bytes to fetch from the leader during a request
#replica.fetch.min.bytes=1
# The number of threads used to process replica requests
#num.replica.fetchers=1
# The frequency with with the high watermark is flushed to disk
#replica.high.watermark.checkpoint.interval.ms=5000
```

The next group of settings controls the behavior of Kafka's disk input/output (I/O). These settings are used to balance the durability of the data when it's not using replication with Kafka's throughput for producers and consumers.

The two parameters, `log.flush.interval.messages` and `log.flush .interval.ms` control the minimum and maximum times between flush events. The `log.flush.interval.ms` setting is the longest Kafka waits until flushing to disk, thereby defining an upper bound for flush events. The `log.flush .interval.messages` parameter causes Kafka to flush messages to disk after a number of messages have been received. If the partition collects messages faster than the interval time, this defines the lower bound on the flush rate.

Without replication, the time between flush events is the amount of data that can be lost in the event of a broker failure. The default is 10,000 messages or 1,000 milliseconds (1 second), which is typically

sufficient for most use cases. Before setting these values to be larger or smaller, there are two things to consider: throughput and latency. The process of flushing to disk is the most expensive operation performed by Kafka, and increasing this rate reduces the throughput of the system as a whole. Conversely, if the flush is very large, it takes a relatively long time. Increasing the time between flushes can lead to latency spikes in responses to producers and consumers as a large bolus of data is flushed to disk.

You can control both these parameters on a per-topic basis using the `log.flush.intervals.messages.per.topic` and `log.flush.intervals.ms.per.topic` parameters respectively. These parameters take a comma-separated list of `<topic>:<value>` tuples.

```
# The number of messages to accept before forcing
# a flush of data to disk
log.flush.interval.messages=10000
# The maximum amount of time a message can sit in a
# log before we force a flush
log.flush.interval.ms=1000
# Per-topic overrides for log.flush.interval.ms
#log.flush.intervals.ms.per.topic=topic1:1000, topic2:3000
```

The next set of parameters control the retention time for topics as well as the size of log segments. Logs are always removed on deleting an entire segment, so the size of the segment file combined with the retention time defines how quickly expired data will be removed.

The basic parameter that controls the retention is the `log.retention.hours` setting. When a log segment exceeds this age it is deleted. This happens on a per-partition basis. There is also a `log.retention.bytes` setting, but it has little utility in practice. First, as discussed earlier, deleting log segments on space constraints is somewhat more difficult to manage than time. Second, `log .retention.bytes` uses a 32-bit integer rather than Java's 64-bit long. This limits topics to at most 2GB per-partition per-topic. Outside of development and testing, this is rarely sufficiently large in a production environment.

```
# The minimum age of a log file to be eligible for deletion
log.retention.hours=168
# A size-based retention policy for logs. Segments are pruned from the
# log as long as the remaining segments don't drop
# below log.retention.bytes.
#log.retention.bytes=1073741824
```

The size of each segment is controlled by the `log.segment.bytes` and the `log.index.size.max.bytes` settings. The first parameter controls the size of the log segment, which is the file that contains the actual messages as they were sent to Kafka. The on-disk format is identical to the wire format so that the messages can be quickly flushed to this disk. The default size of this file is 1GB, and in the following example it has been set to 512MB for development purposes.

The `log.index.size.max.bytes` parameter controls the size of the index file that accompanies each log segment. This file is used to store index information about the offsets for each message stored in the file. It is allocated 10MB of space using a sparse file method by default, but if it fills up before the segment file itself is filled, it causes a new log segment to roll over.

```
# The maximum size of a log segment file. When this size is
# reached a new log segment will be created.
log.segment.bytes=536870912
```

The rate at which segments are checked for expiration is controlled by `log.cleanup.interval.mins`. This defaults to checking every minute and this is usually sufficient.

```
# The interval at which log segments are checked to see if they can
# be deleted according to the retention policies
log.cleanup.interval.mins=1
```

As mentioned in the "Kafka Prerequisites" section, Kafka uses ZooKeeper to manage the status of the brokers in a given cluster and the metadata associated with them. The `zookeeper.connect` parameter defines the ZooKeeper cluster that the broker uses to expose its metadata. This takes a standard ZooKeeper connection string, allowing for comma-separated hosts. It also allows the brokers to take advantage of ZooKeeper's `chroot`-like behavior by specifying a default path in the connection string. This can be useful when hosting multiple independent Kafka clusters in a single ZooKeeper cluster.

```
# Zookeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify
# the root directory for all kafka znodes.
zookeeper.connect=localhost:2181
```

Like other ZooKeeper clients, Kafka allows for the control of the connection and session timeout values. These are controlled by the `zookeeper .connection.timeout.ms` and `zookeeper.session.timeout.ms` settings, respectively, and both are set to 6,000 milliseconds (6 seconds) by default.

```
# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=1000000
# Timeout in ms for the zookeeper session heartbeat.
#zookeeper.session.timeout.ms=6000
```

With the introduction of replication, Kafka 0.8 also introduces a controlled shutdown behavior for the broker. The setting `controlled.shutdown.enabled`, which is active by default, controls this behavior. When active, when a leader receives a shutdown command it attempts to reassign control of each of the topic partitions it leads to another broker in the ISR. It attempts to do this `controlled.shutdown.max.retries` times, backing off by `controlled .shutdown.retry.backoff.ms` milliseconds between each attempt. After the last retry it shuts down, possibly uncleanly.

This feature allows for maintenance to be more easily performed on the cluster. Assuming that consumers and producers are aware of changes to the metadata, moving the leadership position to another broker can allow the entire cluster to be upgraded without any perceptible downtime. Unfortunately, most clients do not have good recovery facilities, so this is presently not possible.

```
#controlled.shutdown.enabled=true
#controlled.shutdown.max.retries=3
#controlled.shutdown.retry.backoff.ms=5000
```

You can find most of the settings shown in this section in the `config/server .properties` file of the Kafka installation. This properties file is mostly intended for development purposes. In production, most of the defaults can be used, except for perhaps the socket buffering options. The `broker.id`, `log.dirs` and `zookeeper.connect` properties must be set in all cases.

Starting a Kafka Broker Cluster

After a proper Kafka configuration file has been created, you can start it using the `kafka-server-start.sh` script. This script takes the properties file containing the configuration:

```
$ ./bin/kafka-server-start.sh config/server.properties
```

The server normally starts in the foreground because it doesn't have a built-in daemon mode. A common pattern for starting the Kafka server in the background, such as in an `init.d` script, is to write the output to a log file and background the server:

```
$ ./bin/kafka-server-start.sh \
> config/server.propertiers > kafka.log 2>&1 &
```

A Multi-Broker Cluster on a Single Machine

Sometimes during the course of development it is desirable to have more than one Kafka broker available to ensure that applications can appropriately deal with partitions on multiple servers and the management of reading from the ISR and so on. For example, developing a client for a new language should always be done against a multi-broker cluster.

Starting multiple brokers on a single machine is a fairly straightforward task. Each broker must be assigned a separate broker id, port and log directory. For example, the following three files could be used:

```
config/server-1.properties:
broker.id=1
port=6001
log.dirs=/tmp/kafka-logs-1
zookeeper.connect=localhost:2181
config/server-2.properties:
broker.id=2
port=6002
log.dirs=/tmp/kafka-logs-2
zookeeper.connect=localhost:2181
config/server-3.properties:
broker.id=3
port=6003
log.dirs=/tmp/kafka-logs-3
zookeeper.connect=localhost:2181
```

Starting each of the brokers is very similar to starting a single broker on a server. The only difference is that different Java Management Extension (JMX) ports need to be specified. Otherwise, the second and third Kafka brokers will fail to start due to the fact that the first broker has bound the JMX port.

Interacting with Kafka Brokers

Kafka is designed to be accessible from a variety of platforms, but it ships with Java libraries because it natively runs on the Java Virtual Machine (JVM). In its native clients, it provides one option for applications that write to Kafka and two options for applications that read from Kafka.

To use Kafka from Java, the Kafka library and the Scala run time must be included in the project. Unfortunately, due to the way Scala is developed, libraries are generally tied to the Scala library they are built against. For pure Java projects, this does not matter, but it can be annoying for Scala projects.

These examples use only Java. The version of Kafka built against Scala 2.8.0 is used:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka_2.8.1</artifactId>
  <version>0.8.1-SNAPSHOT</version>
</dependency>
<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-library</artifactId>
  <version>2.8.1</version>
</dependency>
```

Producers

To use the Kafka Producer, a `Properties` object is either constructed or loaded from a `.properties` file. In production systems, the properties are usually loaded from a file to allow them to be changed without having to recompile the project. In this simple example, the `Properties` object is constructed. These `Properties` are then passed to the constructor of the `ProducerConfig` object:

```
public static void main(String[] args) throws Exception {
  Properties props = new Properties();
  props.put("metadata.broker.list",
    "localhost:6001,localhost:6002,localhost:6003");
  props.put("serializer.class", "kafka.serializer.StringEncoder");
  props.put("request.required.acks", "1");
  ProducerConfig config = new ProducerConfig(props);
```

This configuration object is then passed on to the Producer object, which takes a type for both its `Key` and its `Message` type. Although the Key is generally optional, it is important that the `serializer.class` defined in the configuration properties can successfully encode the objects used for the `Key` and the `Message`. In this case, Strings are being used for both `Key` and `Message`:

```
String topic = args[0];
Producer<String,String> producer = new Producer<String,String>(config);
```

This simple example reads lines from the console and sends them to Kafka. The Producer itself takes `KeyedMessage` objects either singly or as part of a List collection. These messages are constructed with a topic, a key, and a message. If the key is not required, only the topic and the message can be used as in this example:

```
BufferedReader reader = new BufferedReader(
  new InputStreamReader(System.in));
while(reader.ready()) {
  String toSend = reader.readLine();
  producer.send(new KeyedMessage<String,String>(topic, toSend));
}
```

Consumers

Applications have two options for Kafka Consumer implementations. The first is a high-level implementation that uses ZooKeeper to manage and coordinate a consumer group. The other is a low-level implementation for applications that need to manage their own offset information or need access to features not found in the Simple Consumer, such as the ability to reset their offsets to the beginning of the stream.

This example uses the Simple Consumer API to print messages produced by the Producer on the console. Like the Producer example, the example begins with the construction of the

`ConsumerConfig` object from a `Properties` object:

```
public static void main(String[] args) {
  Properties props = new Properties();
  props.put("zookeeper.connect", "localhost:2181");
  props.put("group.id", "console");
  props.put("zookeeper.session.timeout.ms", "500");
  props.put("zookeeper.sync.timeout.ms", "500");
  props.put("auto.commit.interval.ms", "500");
  ConsumerConfig config = new ConsumerConfig(props);
```

Most Consumer applications are multithreaded, and many read from multiple topics. To account for this, the high-level Consumer implementation takes a map of topics with the number of message streams that should be created for each topic. These streams are usually distributed to a thread pool for processing, but in this simple example only a single message stream will be created for the topic:

```
String topic = args[0];
HashMap<String,Integer> topicMap = new HashMap<String,Integer>();
topicMap.put(topic,1);
ConsumerConnector consumer=Consumer.createJavaConsumerConnector(config);
Map<String,List<KafkaStream<byte[],byte[]>>> consumerMap =
  consumer.createMessageStreams(topicMap);
KafkaStream<byte[],byte[]> stream = consumerMap.get(topic).get(0);
```

The `KafkaStream` object defines an iterator with a class of `ConsumerIterator`, which is used to retrieve data from the stream. Calling the `next` method on the `Iterator` returns a `Message` object that can be used to extract the Key and the Message:

```
ConsumerIterator<byte[],byte[]> iter = stream.iterator();
while(iter.hasNext()) {
  String message = new String(iter.next().message());
  System.out.println(message);
}
```

Apache Flume: Distributed Log Collection

Kafka was designed and built within a specific production environment. In this case, its designers essentially had complete control over the available software stack. In addition, they were replacing an existing component with similar producer/consumer semantics (though different delivery semantics), making Kafka relatively easy to integrate from an engineering perspective.

In many production environments, developers are not so lucky. There often are a number of pre-existing legacy systems and data collection or logging paths. In addition, for a variety of reasons, it is not possible to modify these systems' logging behavior. For example, the software could be produced by an outside vendor with their own goals and development roadmap.

To integrate into these environments, there is the Apache Flume project. Rather than providing a single opinionated mechanism for log collection, like Kafka, Flume is a framework for integrating the data ingress and egress of a variety of data services. Originally, this was primarily the collection of logs from things like web servers with transport to HDFS in a Hadoop environment, but Flume has since expanded to cover a much larger set of use-cases, sometimes blurring the line between the data motion discussed in this chapter and the processing systems discussed in Chapter 5.

This section introduces the Flume framework. The framework itself ships with a number of out-of-the-box components that can be used to construct immediately useful infrastructure. These basic components and their usage are covered in fair detail. It is also possible to develop custom components, which is usually mostly done to handle business-specific data formats or processing needs. Flume, like Kafka, is developed on top of the JVM, so extending the framework is fairly straightforward.

The Flume Agent

Flume uses a network-flow model as its primary organizing feature, which is encapsulated into a single machine known as an agent. These Agent machines are roughly analogous to Kafka's broker and minimally consist of three components: a source, a channel, and a sink. An agent can have any number of these basic flows, organized as shown in Figure 4.2.

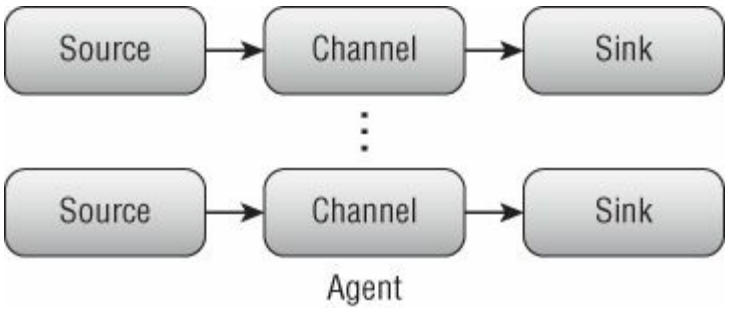


Figure 4.2

As shown in Figure 4.2, the source is the start of any flow on an agent. Sources are components that either retrieve or receive data from an outside source. In most cases, this is data from an application. For example, a web server may produce filesystem logs. To retrieve them, a source that can tail log files is used as part of an agent installed on the machine running the web server.

After the source has received data, it transfers this data to one or more channels. Channels define the persistence and transaction semantics. Depending on the channel used, the agent can store its data to

disk or a database, or it can keep it in memory. This allows different performance and reliability characteristics to be selected for each agent.

A source may also choose to write its data to multiple channels, as shown in Figure 4.3. This is typically used to multiplex data for better load balancing and processing parallelism. It can also be used to replicate data or provide multiple data paths for the entire stream. To define how this happens, the source definition uses a selector. The default selector is a replication selector that writes the same data to all channels. The other built-in option is the multiplexing selector. This selector uses information from the data element to determine the destination channel of each piece of data arriving at the source.

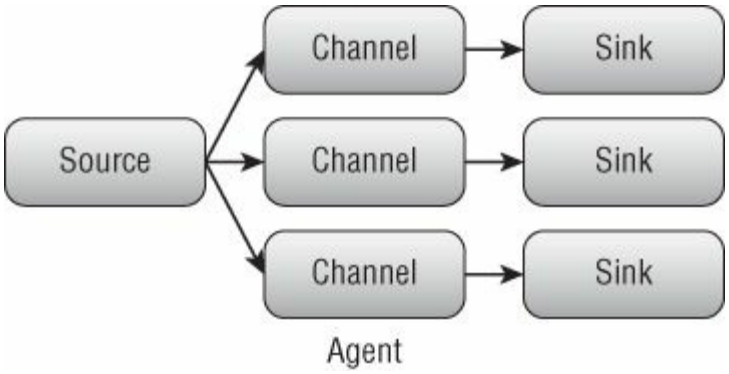


Figure 4.3

Sinks remove data from the channels and relay them to either a final output, such as HDFS, or on to another agent. Unlike sources, sinks are each attached to a single channel. Sending the output from multiple sinks to a single agent is used to perform de-multiplexing.

Sinks can also be coordinated to some extent using sink processors. There are three built-in sink processors: the default processor, a load-balancing processor, and a failover processor. These are described in more detail later in this chapter.

Configuring the Agent

Agents are configured, much like Kafka, using a properties file. It defines both the arrangement of sources, channels, and sinks as well as the configuration of the individual elements. The arrangement is the same for all combinations, whereas the individual configurations depend on the type of each of the elements. The configuration begins with the definition of sources and sinks for each agent running on the machine, with the following basic form:

```
<agent name>.sources= <source1> ... <sourceN>
<agent name>.channels= <channel1> ... <channelN>
<agent name>.sinks= <sink1> ... <sinkN>
```

The `<agent name>` and `<source1>` entries are replaced with unique names that are used to identify these elements elsewhere in the properties file. For example, defining a properties file with two agents might look something like this:

```
agent_1.sources= source-1
agent_1.channels= channel-1 channel-2
agent_1.sinks= sink-1 sink-2
agent_2.sources= source-1
agent_2.channels= channel-1
agent_2.sinks= sink-1
```

In this example, the two agents are configured to each have a single source. The first agent, `agent_1`, uses two channels and two sinks to either replicate or multiplex its input. The second agent, `agent_2`, is more simply defined with a single channel and sink.

Note that the source, channel, and sink names do not need to be unique across different agents. The configuration is always with respect to an agent definition.

The next section binds the sources and the sinks for each agent to their appropriate channels. The source is bound with properties of the form `<agent name>.sources.<source>.channels`, and the sink is bound to each channel with `<agent name>.sinks.<sink>.channel`. In this example, the source is bound to all channel(s) and the sink(s) are bound to the channel with the same name:

```
agent_1.sources.source-1.channels= channel-1 channel-2
agent_1.sinks.sink-1.channel= channel-1
agent_1.sinks.sink-2.channel= channel-2
agent_2.sources.source-1.channels= channel-1
agent_2.sinks.sink-1.channel= channel-1
```

The Flume Data Model

Before continuing with the configuration of a Flume agent, some discussion of the Flume internal data model is required. This data model is defined by the `Event` data structure. The interface to this structure is defined by Flume as follows:

```
package org.apache.flume;
import java.util.Map;
/**
 * Basic representation of a data object in Flume.
 * Provides access to data as it flows through the system.
 */
public interface Event {
  public Map<String, String> getHeaders();
  public void setHeaders(Map<String, String> headers);
  public byte[] getBody();
  public void setBody(byte[] body);
}
```

Like the Kafka `Message`, the `Event` payload is an opaque byte array called the `Body`. The `Event` also allows for the introduction of headers, much like an HTTP call. These headers provide a place for the introduction of metadata that is not part of the message itself.

Headers are usually introduced by the `Source` to add some metadata associated with the opaque event. Examples might include the host name of the machine that generated the event or a digest signature of the event data. Many `Interceptors` modify the header metadata as well. The metadata components are used by Flume to help direct events to their appropriate destination. For example, the multiplexing selector uses the header structure to determine the destination channel for an event. How each component modifies or uses the metadata details is discussed in detail in the sections devoted to each selector.

Channel Selectors

If multiple channels are to be used by a source, as is the case with the `agent_1` source in the previous example, some policy must be used to distribute data across the different channels. Flume's default policy is to use a replicating selector to control the distribution, but it also has a built-in

multiplexing selector that is used to load balance or partition inputs to multiple sinks. It is also possible to implement custom channel selection behavior.

Replicating Selector

The replicating selector simply takes all inputs and sends them to all channels for further processing. Because it is the default, no configuration is necessary, but it may be explicitly defined for a source by setting the `selector.type` property to `replicating`:

```
agent_1.source.source-1.selector.type= replicating
```

The only additional option for the replicating selector is the ability to ignore failures to write to specific channels. To do this, the `selector.optional` property is set with a space-separated list of channels to ignore. For example, this setting makes `agent_1`'s second channel optional:

```
agent_1.source.source-1.selector.optional= channel-2
```

Multiplexing Selector

The other built-in channel selector is the multiplexing selector. This selector is used to distribute data across different channels depending on the `Event`'s header metadata. It is activated by changing the source's `selector.type` to be multiplexing:

```
agent_1.source.source-1.selector.type= multiplexing
```

By default, the channel is determined from the `flume.selector.header` header in the metadata, but it can be changed with the `selector.header` property. For example, this changes the selector to use the shorter `timezone` header:

```
agent_1.source.source-1.selector.header= timezone
```

The multiplexing selector also needs to know how to distribute the values from this header to the various channels assigned to the source. The target channel(s) is specified with the `selector.mapping` property. The values must be explicitly mapped for each possible value with a default channel specified with the `selector.default` property. For example, to map Pacific Daylight Time (PDT) and Eastern Daylight Time (EDT) to `channel-2` and all other events to `channel-1`, the configuration would look like this:

```
agent_1.source.source-1.selector.mapping.PDT= channel-2
agent_1.source.source-1.selector.mapping.EDT= channel-2
agent_1.source.source-1.selector.default= channel-1
```

It is arguable that the need to explicitly specify the mapping of a header to a channel significantly reduces the utility of the multiplexing selector. In fact, this makes it mostly useful as a routing tool rather than a mechanism for improving processing parallelism. In that case, a better choice would be a load-balancing sink processor, which is described later in this chapter.

Implementing Custom Selectors

Flume also supports the implementation of custom selectors. To use this in a configuration, simply specify the fully qualified class name (sometimes abbreviated as FQCN) as the `selector.type`. For example, this tells Flume to use the `RandomSelector` for `agent_1`'s Source:

```
agent_1.source.source-1.selector.type=
  wiley.streaming.flume.RandomSelector
```

Depending on the selector implementation, other configuration parameters might be needed to operate properly. The selector itself is implemented by extending the `AbstractChannelSelector` class:

```
import org.apache.flume.Channel;
import org.apache.flume.Context;
import org.apache.flume.Event;
import org.apache.flume.channel.AbstractChannelSelector;
public class RandomChannelSelector extends AbstractChannelSelector {
  List<List<Channel>> outputs = new ArrayList<List<Channel>>();
  List<Channel>       empty   = new ArrayList<Channel>();
  Random              rng     = new Random();
```

This class requires the implementation of three different methods. The first is the configuration method, `configure`, that should extract information about the channels assigned to this source as well as any other properties that might have been set in the configuration:

```
  public void configure(Context context) {
```

Interacting with the context object is quite simple. There are a number of "getters" defined on the context class that can be used to extract configuration parameters. These properties have already been specialized to the appropriate selector. For example, if the class needed to access the `"optional"` property, like the replicating selector, then the following code would get the string associated with the property:

```
  String optionalList = context.getString("optional");
```

However, `RandomSelector` only needs the list of channels associated with this source. This method, `getAllChannels`, is actually part of the `AbstractChannelSelector` class, so it is called directly. Because the `RandomSelector` sends each event to one channel, list elements are constructed for later use:

```
  for(Channel c : getAllChannels()) {
    List<Channel> x = new ArrayList<Channel>();
    x.add(c);
    outputs.add(x);
  }
```

The other two methods that must be implemented by a custom channel selector are `getRequiredChannels` and `getOptionalChannels`. Each method gets a single argument in the form of an `Event`, whose metadata can be checked. The `RandomSelector` does not check metadata; it simply returns a random array containing a single channel as constructed in the configuration method:

```
  public List<Channel> getRequiredChannels(Event event) {
    return outputs.get(rng.nextInt(outputs.size()));
  }
  public List<Channel> getOptionalChannels(Event event) {
    return empty;
  }
```

Flume Sources

One of the chief draws for Flume is its ability to integrate with a variety of systems. To support this, Flume ships with a number of built-in source components. It also supports the ability to implement custom sources in much the same manner as the channel selector discussed in the previous section.

Avro Source

Avro is a structured data format developed by Yahoo!. Philosophically similar to other data formats such as protocol buffers from Google and Thrift from Facebook, it is the native data interchange format for Flume. All three of these systems use an Interface Definition Language (IDL) to define structures that are similar to C `structs` or simple Java classes. The formats can use this description to encode structures in their native language to a high-performance binary wire protocol that is easily decoded by another service with access to the schema defined by the IDL. In the case of Avro, this IDL is JSON, making it easy to define data structures that can be implemented by a variety of languages.

Like Thrift, Avro also defines a remote procedure call (RPC) interface. This allows "methods" defined by Avro to be executed via a command sent over a TCP/IP connection. It is this RPC interface that Flume implements to pass events into the agent.

The Avro source in Flume is assigned to a source definition by specifying the type as `avro`. It also takes a `bind` and `port` property that defines the network interface and port the agent should listen on for Avro RPC calls. For example, the following options bind the Avro listener to all network interfaces defined by the system on port 4141:

```
agent_1.source.source-1.type=avro
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=4141
```

The special IP address 0.0.0.0 should be used with care. If the agent is running on a server that has an external IP address, it binds to that IP address. In most cases, this is harmless because the server almost certainly sits behind a firewall that protects it from most mischief. However, it may also open the port to internal services that should not have access. For example, a production server could give inadvertent access to a staging server. If that staging server were to be accidentally misconfigured, it could potentially inject inappropriate data into the system.

Although the `type`, `bind`, and `port` properties are required, the Avro source also allows for some optional properties. The optional properties allow for compression of the Avro data stream by setting the `compression.type` to `deflate`. The stream may also be encrypted via SSL (Secure Socket Layer) by setting the `ssl` property to `true` and pointing the `keystore` property to the appropriate Java KeyStore file. The `keystore-password` property specifies the password for the KeyStore file itself. Finally, the type of KeyStore is specified by `keystore-type`, defaulting to JKS.

Thrift Source

Prior to the release of Flume and Kafka, Thrift and Scribe were (and still are) popular choices for collecting logs in new infrastructures. The Thrift source allows Flume to integrate with existing Thrift/Scribe pipelines. After you set the `type` to `thrift`, you must set the address binding and the port as well. Like the Avro source, take care when setting the address binding to 0.0.0.0:

```
agent_1.source.source-1.type=thrift
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=4141
```

Netcat Source

Even simpler than the Avro and Thrift sources is the Netcat Source. For those unfamiliar with `netcat`, it is a utility found in most UNIX-like operating systems that can write or read to a raw

TCP socket in much the same way that the `cat` command reads and writes to file descriptors.

The Netcat source, like the Thrift and Avro sources, binds to a specific address and port after you have set the type to `netcat`:

```
agent_1.source.source-1.type=netcat
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=4141
```

The Netcat source reads in newline-delimited text lines with, by default, a maximum length of 512 bytes (not characters, which are of variable length). To increase this limit, modify the `max-line-length` property:

```
agent_1.source.source-1.max-line-length=1024
```

To provide backpressure support, the Netcat source also positively acknowledges that every event is received with an OK response. If needed, you can disable this `acknowledgement` by setting the `ack-every-event` property to `false`:

```
agent_1.source.source-1.ack-every-event=false
```

Syslog Source

Syslog is a standard for systems logging that was originally developed as part of the Sendmail package in the 1980s. It has since grown into a standard that is governed—originally by RFC 3164 "The BSD Syslog Protocol" and now by RFC 5424 "The Syslog Protocol."

Flume supports RFC 3164 and much of RFC 5424 through three flavors of Syslog source. The first is a single port TCP source called `syslogtcp`. Like other TCP-based sources, it takes the usual bind address and port:

```
agent_1.source.source-1.type=syslogtcp
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=5140
```

The maximum size of the events is controlled with the `eventSize` property, which defaults to 2,500 bytes:

```
agent_1.source.source-1.eventSize=2500
```

Flume also supports a more modern multiport implementation of the TCP Syslog source. To use it, change the `type` to `multiport_syslogtcp` and use the `ports` property instead of the `port` property when configuring the source:

```
agent_1.source.source-1.type=multiport_syslogtcp
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=10001 10002 10003
```

The Syslog source then monitors all of the ports for syslog events in an efficient way. In addition to the `eventSize` property supported by the single-port version, the multiport implementation adds the ability to control the character set used to interpret the events on a per-port basis. The default character set, which is normally UTF-8, is controlled by the `charset.default` property and the others set by `charset.port.<port>`. This example sets the default to UTF-16 and port 10003 to use UTF-8:

```
agent_1.source.source-1.charset.default=UTF-16
agent_1.source.source-1.charset.port.10003=UTF-8
```

The Syslog source is implemented using Apache Mina and exposes some of the tuning parameters available in that library. Generally speaking, these settings do not need to be changed, but in some virtual environments changing them might be necessary. The first parameters are the batch size and the read buffer, set to 100 events and 1,024 bytes, respectively, by default:

```
agent_1.source.source-1.batchSize=100
agent_1.source.source-1.readBufferSize=1024
```

The number of processors can also be controlled by configuration. The Syslog Source spawns two reader threads for every processor and attempts to auto-detect the number of available processors. In virtual environments, this is the most likely to cause problems, as the apparent number of processors is not necessarily indicative of actual processors available. To override auto-detection, set the number of processors using the `numProcessors` property:

```
agent_1.source.source-1.numProcessors=4
```

Finally, the Syslog source also has an UDP-based source. For this source, only the address binding and port are required with the type set to `syslogudp`:

```
agent_1.source.source-1.type=syslogudp
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=5140
```

No other options are available for this source, which, because it uses UDP, is usually only suitable for situations that do not require reliable event delivery.

HTTP Source

The HTTP source allows Flume to accept events from services that can make HTTP requests. For many potential event sources, this offers the ability to also add rich metadata headers to the event without having to implement a custom source. This event, like all other TCP-based sources, needs a binding and a port, and it sets the `type` property to `http`:

```
agent_1.source.source-1.type=http
agent_1.source.source-1.bind=0.0.0.0
agent_1.source.source-1.port=8000
```

Events are transmitted to the HTTP source using the POST command and return either a 200 or a 503 response code. The 503 response code is used when a channel is full or otherwise disabled. An application, upon receiving this event, should retry sending the event through some other mechanism.

After the event arrives at the HTTP source it must be converted into Flume events. This is done using a pluggable Handler class, with a `JSONHandler` and a `BlobHandler` class provided by Flume. The handler, which is set to the `JSONHandler` by default, is controlled by the handler property. This property takes a fully qualified class name of a class that implements the `HTTPSourceHandler` interface:

```
agent_1.source.source-1.handler=org.apache.flume.source.http.JSONHandler
```

The `JSONHandler` expects an array of JSON objects. Each of these objects contains a headers object as well as a body object, which are used to populate the Flume event's header metadata and body byte array, respectively. Even a single event must be wrapped in an array for processing, as in this example:

```
[{
  "headers":{
```

```
      "timestamp": "34746432",
      "source": "somehost.com"
    },
    "body":"the body goes here."
  }]
```

Note that the values for each of the header elements must be a string, even if the real value is a number. The Flume event object only supports strings in the header metadata.

The `BlobHandler` is much simpler, being designed to take in relatively large binary objects. No attempt is made by the handler to interpret the events.

WARNING

Events are buffered in memory, so very large elements could fill the Java heap allocated to the Flume process. This causes Flume to crash and, depending on the channel, could cause a loss of data. Therefore, it is important to restrict the size of objects sent to a BlobHandler.

Java Message Service (JMS) Source

The Java Message Service (JMS) source allows Flume to integrate with Java-based queuing systems. These systems are popular mechanisms for moving data around when the entire stack is Java-based. The JMS source has primarily been tested with ActiveMQ, but it should be able to attach to any JMS provider.

Connecting to a JMS provider requires some familiarity with JMS. It minimally requires providing an initial context factory name, a connection factory, and a provider URL. It also requires a destination name and a destination type that may be either a queue or a topic. For example, to connect to a local ActiveMQ server with a queue named `DATA`, you could use the following configuration:

```
agent_1.source.source-1.type=jms
agent_1.source.source-1.initialContextFactory=
  org.apache.activemq.jndi.ActiveMQInitialContextFactory
agent_1.source.source-1.connectionFactory= GenericConnectionFactory
agent_1.source.source-1.providerURL= tcp://localhost:61616
agent_1.source.source-1.destinationName= DATA
agent_1.source.source-1.destinationType= QUEUE
```

JMS sources support the notion of converters, which are able to convert inputs from the JMS queue to Flume events. Although it is possible to implement a custom converter, the default converter is able to convert messages consisting of objects, byte, or text messages to native Flume events. The properties on the JMS message are converted to headers in the Flume events.

Exec Source

The Exec source allows any command to be used as input to stream. It is often used to emulate the Tail Source that was part of Flume prior to version 1.0. Setting the type to `exec` and providing a command property is all that is required under normal circumstances. For example, this command tails a log file for `agent_1`:

```
agent_1.source.source-1.type= exec
agent_1.source.source-1.command= tail -F /var/log/logfile.log
```

The `-F` switch, which tells `tail` to account for log rotation, should be used in place of the more common `-f` switch. There are also a number of optional parameters that can be set as part of this Source. For commands that need it, a shell can be specified with the `shell` property. This allows for the use of shell features such as wildcards and pipes:

```
agent_1.source.source-1.shell= /bin/sh -c
```

When the command exits, it can be restarted. This is disabled by default; enable it by setting the restart property to `true`:

```
agent_1.source.source-1.restart= true
```

When restarting a command, the `restartThrottle` property controls how quickly the command is restarted (in milliseconds):

```
agent_1.source.source-1.restartThrottle= 1000
```

Normally, the Exec source only reads from the standard input. It can also read events from the standard error stream by setting the `logStdErr` property:

```
agent_1.source.source-1.logStdErr= true
```

To improve performance, the Exec source reads a batch of lines before sending them on to their assigned channel(s). The number of lines, which defaults to 20, is controlled with the `batchSize` property:

```
agent_1.source.source-1.batchSize= 20
```

Unidirectional Sources Do Not Support "Backpressure"

If at all possible, avoid sources like the Exec source in production environments. Although sources like the Exec source are very tempting, they render any reliability guarantees made by the Flume framework meaningless. The framework has no mechanism for communicating with the event generator. It cannot report channels that have filled or otherwise become unavailable. The end result being that the generator will continue to blindly write to the source, and data is potentially lost. Fortunately, the most common use case is to emulate the Tail source (which was removed for a reason), which can usually be better implemented using the Spool Directory source.

Spool Directory Source

The most common method for data recording for services that do not make use of a data motion system is logging to the file system. Early versions of Flume tried to take advantage of this by implementing a Tail source that allowed Flume to watch the logs as they were being written and move them into Flume. Unfortunately, there are a number of problems with this approach if the goal is to provide a reliable data motion pipeline. The Spool Directory source attempts to overcome the limitations of the Tail source by providing support for the most common form of logging—logs with rotation.

To use the Spool Directory source, the application must set up its log rotation such that when a log file is rolled it is moved to a spool directory. This is usually fairly easy to do with most logging systems. The Spool Directory source monitors this directory for new files and writes them to Flume. This minimally requires the definition of the `spoolDir` property along with setting the type to `spooldir`:

```
agent_1.source.source-1.type=spooldir
agent_1.source.source-1.spoolDir=/var/logs/
```

A common pattern for applications is to write to a file ending in `.log` and then to roll it to another log file ending in a timestamp. Normally the active log file causes errors for the Spool Directory source, but it can be ignored by setting the `ignorePattern` property to ignore files ending in `.log`:

```
agent_1.source.source-1.ignorePattern=^.*\.log$
```

When the Spool Directory Source finishes processing a file, it marks the file by appending `.COMPLETED` to the filename, which is controlled by the `fileSuffix` property. This allows log cleanup tools to identify files that can be safely deleted from the file system. The source can also delete the files when it is done processing them by setting the `deletePolicy` property to `immediate`:

```
agent_1.source.source-1.fileSuffix=.DONE
agent_1.source.source-1.deletePolicy=immediate
```

By default, the source assumes that the records in the files are text separated by newlines. This is the behavior of the `LINE` deserializer, which ships with Flume. The base Flume installation also ships with `AVRO` and `BLOB` deserializers. The `AVRO` deserializer reads Avro-encoded data from the files according to a schema specified by the `deserializer.schemaType` property. The `BLOB` deserializer reads a large binary object from each file, usually with just a single object per file. Like the HTTP BlobHandler, this can be dangerous because the entire Blob is read into memory. The choice of deserializer is via the `deserializer` property:

```
agent_1.source.source-1.deserializer=LINE
```

If needed, custom deserializers can be supplied. Any class that implements the `EventDeserializer.Builder` interface can be used in place of the built-in deserializers.

Implementing a Custom Source

It is rarely necessary, given the array of sources built in to Flume, but it is possible to implement a custom source object. In Flume, there are two types of sources: a Pollable source and an Event-Driven source. Pollable sources are executed by Flume in their own thread and queried repeatedly to move data onto its associated channels. An Event-Driven source is responsible for maintaining its own thread and placing events on its channels asynchronously.

In this section, an Event-Driven source connecting Flume to Redis is implemented to demonstrate the process. Redis is a key-value store discussed in-depth in Chapter 6, "Storing Streaming Data." In addition to providing storage, it also provides an ephemeral publish/subscribe interface that is used to implement the source.

A custom source always extends the `AbstractSource` class and then implements either the `Pollable` or the `EventDriveSource` interface. Most sources also implement the `Configurable` interface to process configuration parameters. In this case, the Redis source implements both the `Configurable` and the `EventDrivenSource` interfaces:

```
package wiley.streaming.flume;
import java.nio.charset.Charset;
import org.apache.flume.Context;
import org.apache.flume.EventDrivenSource;
import org.apache.flume.channel.ChannelProcessor;
import org.apache.flume.conf.Configurable;
import org.apache.flume.event.EventBuilder;
import org.apache.flume.source.AbstractSource;
import redis.clients.jedis.Jedis;
import redis.clients.jedis.JedisPool;
import redis.clients.jedis.JedisPoolConfig;
import redis.clients.jedis.JedisPubSub;
public class RedisSource extends AbstractSource
  implements Configurable, EventDrivenSource {
```

The Configurable interface has a single configure method, which is used to obtain the Redis hostname, port, and subscription channel. In this case, all of the configuration parameters have defaults:

```
String           redisHost;
int              redisPort;
String           redisChannel;
Jedis            jedis;
public void configure(Context context) {
  redisHost    = context.getString("redis-host", "localhost");
  redisPort    = context.getInteger("redis-port", 6379);
  redisChannel = context.getString("redis-channel", "flume");
}
```

The `EventDrivenSource` does not actually require any methods be implemented; it is only used to determine the type of source. Any threads or even handlers are set up and broken down and are handled in the `start` and `stop` methods. In this case, most of the work is done in the `start` method:

```
  @Override
  public synchronized void start() {
    super.start();
    processor = getChannelProcessor();
```

First, the `ChannelProcessor` is obtained for future use. Recall that the `ChannelProcessor` decides which Channel will be used based on the `Event`. The `ChannelProcessor` is used by the subscription thread that is established next:

```
    JedisPool pool = new JedisPool(
      new JedisPoolConfig(),redisHost,redisPort);
    jedis = pool.getResource();
    new Thread(new Runnable() {
      public void run() {
        jedis.subscribe(pubSub, redisChannel);
      }
    }).start();
  }
```

The Jedis client for Redis blocks its thread when the subscribe command is called, and all subsequent communication is sent to a `JedisPubSub` object. To avoid blocking the main Flume thread, the blocking operation is done in a separate thread in the `start` method. The `JedisPubSub` object has several different methods defined, but only one is necessary here:

```
  ChannelProcessor processor;
  JedisPubSub      pubSub = new JedisPubSub() {
    @Override
    public void onMessage(String arg0, String arg1) {
      processor.processEvent(
        EventBuilder.withBody(arg1, Charset.defaultCharset()));
    }
```

This method uses the `EventBuilder` class to construct an event that is then forwarded to the `ChannelProcessor` to be placed on a channel. The `stop` method unsubscribes from the channel, which unblocks the Jedis thread and lets the process complete:

```
  @Override
  public synchronized void stop() {
    pubSub.unsubscribe();
    super.stop();
  }
```

Using this source is very similar to using any other source. The type property is set to the source's fully qualified class name, and any configuration parameters are set:

```
  agent_1.source.source-1.type=wiley.streaming.flume.RedisSource
  agent_1.source.source-1.redis-channel=events
```

Flume Sinks

Flume sinks sit on the other side of a Flume channel, acting as a destination for events. Flume ships with a large number of built-in sinks, most of which are focused on writing events to some sort of persistent store. Some examples of sinks include an HDFS sink that writes data to Hadoop; a File Roll sink that persists events to disk; and an Elasticsearch sink which writes events to the Elasticsearch data store.

All of these sinks are important parts of a full-fledged data pipeline, but they are not particularly applicable to processing streaming data. As such, they will be mostly left aside in this section in

favor of the sinks that are more useful for streaming processing.

Avro Sink

The Avro sink's primary use case is to implement multi-level topologies in Flume environments. It is also a good way of forwarding events onto a streaming processing service for consumption. The Avro sink has a type property of `avro` and minimally requires a destination host and port:

```
agent_1.sinks.sink-1.type= avro
agent_1.sinks.sink-1.hostname= localhost
agent_1.sinks.sink-1.port= 4141
```

There are also a number of optional parameters that are used to control the behavior of the sink. The connection and reconnect timeouts can be controlled using the appropriate properties, with the ability to specify a reconnection interval. This can be useful when the destination is a load-balanced connection, as load can often only be balanced at connection time. By occasionally reconnecting, the load can be more evenly distributed:

```
agent_1.sinks.sink-1.connect-timeout= 20000
agent_1.sinks.sink-1.request-timeout= 10000
agent_1.sinks.sink-1.reset-connection-interval= 600000
```

Connections can also be compressed by setting the `compression-type` property to `deflate`, the only supported option:

```
agent_1.sinks.sink-1.compression-type= deflate
agent_1.sinks.sink-1.compression-level= 5
```

Thrift Sink

The Thrift sink is largely identical to the Avro sink. It also requires a hostname and a port and can be controlled with the same timeout behavior. However, it does not support Avro's compression or security features:

```
agent_1.sinks.sink-1.type= thrift
agent_1.sinks.sink-1.hostname= localhost
agent_1.sinks.sink-1.port= 4141
```

Implementing Custom Sinks

When communicating with stream processing software, it may be necessary to implement a custom sink. The process for implementing a custom sink is similar to that for implementing a custom source. The custom sink class extends the `AbstractSink` class and optionally implements the `Configurable` interface.

The following example implements a Redis sink to match the Redis source from earlier in the chapter. Rather than subscribing to a publish-subscribe channel, the Redis sink publishes events to the channel. Thus, the implementation begins with the same configuration as the Redis source:

```
public class RedisSink extends AbstractSink implements Configurable {
  String           redisHost;
  int              redisPort;
  String           redisChannel;
  Jedis            jedis;
public void configure(Context context) {
  redisHost    = context.getString("redis-host", "localhost");
  redisPort    = context.getInteger("redis-port", 6379);
  redisChannel = context.getString("redis-channel", "flume");
```

```
}
```

The `start` and `stop` methods are simpler in the implementation of the sink because the process does not need to be asynchronous to the primary thread:

```
@Override
public synchronized void start() {
  super.start();
  pool  = new JedisPool(new JedisPoolConfig(),redisHost,redisPort);
  jedis = pool.getResource();
}
@Override
public synchronized void stop() {
  pool.returnResource(jedis);
  pool.destroy();
  super.stop();
}
```

The `process` method retrieves events from the channel and sends them to Redis. For simplicity, only the body is transmitted in this example, but a more sophisticated sink might convert the output to another form so that it could preserve the header metadata:

```
public Status process() throws EventDeliveryException {
  Channel channel = getChannel();
  Transaction txn = channel.getTransaction();
  Status status = Status.READY;
  try {
    txn.begin();
    Event event = channel.take();
    if(jedis.publish(redisChannel,
      new String(event.getBody(),Charset.defaultCharset())) > 0) {
      txn.commit();
    }
  } catch(Exception e) {
    txn.rollback();
    status = status.BACKOFF;
  } finally {
    txn.close();
  }
  return status;
}
```

Sink Processors

Sink processors are a mechanism for controlling the parallelism of a channel. Essentially, Sink processors are used to define groups of sinks that are then used for load balancing or, possibly, for failover. For streaming applications, the failover application is not as useful as the load-balancing application.

To activate load-balancing sink processing on a channel, set the `processor .type` property of the sink to be `load_balance` and use the `processor.sinks` property to define the sinks that should be used for the group:

```
agent_1.sinkgroups= group-1
agent_1.sinkgroups.group-1.sinks= sink-1 sink-2
agent_1.sinkgroups.group-1.processor.type= load_balance
agent_1.sinkgroups.group-1.processor.selector= random
```

After this definition, the channel is assigned to the group rather than the sink, and events will be

randomly sent to `sink-1` or `sink-2`. The load-balancing processor can also use a round-robin selection method by changing the selector to `round_robin` instead of `random`.

Flume Channels

Flume channels are the temporary storage mechanism between a source and a sink. There are three different types of channel used in production Flume environments: memory channels, file channels and JDBC channels. These channels are listed in terms of increasing safety and decreasing performance.

Memory Channel

The Memory channel is the fastest of the available Flume channels. It is also the least safe because it stores its events in memory until a sink retrieves them. If the process goes down for any reason, any events left in the Memory channel are lost.

This channel is set by assigning it a type of `memory`. In addition, there are several optional parameters that can be set. The most useful of these are the `capacity` and `transactionCapacity` settings, which define the number of events that will be consumed before the channel is considered to be filled. Appropriate values depend greatly on the size of the events and the specific hardware used, so some experimentation is often required to determine optimal values for a given environment:

```
agent_1.channels.channel-1.type= memory
agent_1.channels.channel-1.capacity= 1000
agent_1.channels.channel-1.transactionCapacity= 100
```

File Channel

The File channel, which persists events to disk, is the most commonly used Flume channel. To increase the durability of the data placed in the channel, it persists it to disk for a time. Using a checkpoint and data log system, it is similar to the system used by Kafka. The exception is that it enforces size constraints rather than time constraints in the same way as the Memory channel.

Most File channel definitions can set the `channel.type` property and perhaps the checkpoint and data file locations as in this example:

```
agent_1.channels.channel-1.type= file
agent_1.channels.channel-1.checkpointDir= /home/flume/checkpoint
agent_1.channels.channel-1.dataDirs= /home/flume/data
```

Like Kafka, the File channel can specify multiple data directories to take advantage of JBOD configurations. Also, like Kafka, using a RAID0 configuration will probably yield better performance in practice due to the ability to parallelize writes at the block level.

Java Database Connection (JDBC) Channel

Despite the name, the JDBC channel does not actually allow connections to arbitrary databases from Flume. A better name for it would be the Derby channel as it only allows connections to an embedded Derby database. This channel is the slowest of the three, but it has the most robust recoverability. To use it, simply set the `channel.type` to be `jdbc`. To define the path of the database files, also set the `sysprop.user.home` property:

```
agent_1.channels.channel-1.type= jdbc
agent_1.channels.channel-1.sysprop.user.home= /home/flume/db
```

There are other properties that can be set, but they do not generally need to be changed.

Flume Interceptors

Interceptors are where Flume begins to blur the line between data motion and data processing. The interceptor model allows Flume not only to modify the metadata headers of an event, but also allows the Event to be filtered according to those headers.

Interceptors are attached to a source and are defined and bound in much the same way as channels using the `interceptors` property:

```
agent_1.sources.source-1.interceptors= i-1 i-2
```

The order in which interceptors are defined in the configuration is the order in which they are applied to the event. Because interceptors have the ability to drop an event entirely, awareness of this ordering can be important if the filtering interceptor depends on a header set by an earlier interceptor.

Flume provides a number of useful interceptors that can be used to decorate events or provide useful auditing information during stream processing. Some of the interceptors most commonly used are outlined in this section.

Timestamp Interceptor

The Timestamp interceptor adds the current timestamp as a metadata header on the event with the key `timestamp`. There is an optional `preserveExisting` property that controls whether or not the timestamp is overwritten if it already exists on the event:

```
agent_1.sources.source-1.interceptors.i-1.type=timestamp
agent_1.sources.source-1.interceptors.i-1.preserveExisting=true
```

Host Interceptor

The Host interceptor attaches either the hostname of the Flume agent or the IP address to the header metadata. The default header is called `host`, but it can be changed using the `hostHeader` property. It also supports the `preserveExisting` property:

```
agent_1.sources.source-1.interceptors.i-1.type=host
agent_1.sources.source-1.interceptors.i-1.hostHeader=agent
agent_1.sources.source-1.interceptors.i-1.useIP=false
agent_1.sources.source-1.interceptors.i-1.preserveExisting=true
```

This interceptor is useful for maintaining audit trails during processing.

Static Interceptor

The Static interceptor is used to add a static header to all events. It takes two properties, `key` and `value`, which are used to set the header:

```
agent_1.sources.source-1.interceptors.i-1.type=static
agent_1.sources.source-1.interceptors.i-1.key=agent
agent_1.sources.source-1.interceptors.i-1.value=ninetynine
```

The primary use case for this interceptor is to identify events that have taken different processing paths. By having agents add the path information to the headers, the event can be later tracked through the system.

UUID Interceptor

This interceptor adds a globally unique identifier to each event it processes. By default it sets the `id` header, but this can of course be changed:

```
agent_1.sources.source-1.interceptors.i-1.type=uuid
agent_1.sources.source-1.interceptors.i-1.headerName=id
agent_1.sources.source-1.interceptors.i-1.preserveExisting=true
```

As discussed at the beginning of the chapter, most of these systems implement, at least once, delivery semantics that can occasionally lead to duplicates. Depending on the application, it might be necessary to detect and remove these duplicates. By placing this at the beginning of the Flume topology, Events can be uniquely identified.

Regular Expression Filter Interceptor

The Regular Expression Filter interceptor applies a regular expression to the body of the Event, assuming that it is a string. If the regular expression matches, the event is either preserved or dropped depending on the `excludeEvents` property. If the property is set to `true`, events matching the regular expression are dropped. Otherwise, only events matching the regular expression are preserved. The regular expression itself is defined by the `regex` property:

```
agent_1.sources.source-1.interceptors.i-1.type=regex_filter
agent_1.sources.source-1.interceptors.i-1.regex=.*
agent_1.sources.source-1.interceptors.i-1.excludeEvents=false
```

This interceptor can be used to set up different paths for different types of events. After they are on different paths, they can be easily processed without having to introduce an if-else ladder in the processing code.

Integrating Custom Flume Components

As described in the previous sections, many of the components in Flume can be replaced with custom implementations. These are usually developed outside of the Flume source code, and two integration methods are supplied.

Plug-in Integration

The preferred method of integration is through Flume's plug-in facility. To use this method, create a directory called `plugin.d` in the Flume home directory. Each plug-in that should be included is placed in a subdirectory inside of `plugin.d`. For example, to include the examples from this book, the `plugin.d` structure would look like this:

```
plugin.d/
plugin.d/wiley-plugin/
plugin.d/wiley-plugin/lib/wiley-chapter-4.jar
```

If further dependencies are required, they can be placed in the `libext` subdirectory of the plug-in or built into the included jar directly. Native libraries are placed in the `native` subdirectory. When the Flume agent starts, it scans the plug-in directory and adds whatever jar files it finds there to the classpath.

Classpath Integration

If plug-in integration is not possible for some reason, Flume can also use a more standard classpath-based approach. To use this approach, use the `FLUME_CLASSPATH` environment variable when starting the Flume Agent to add libraries to the Flume agent for later use.

Running Flume Agents

Starting the Flume agent is done using the `flume-ng` script found in the Flume binary directory. The startup script requires a name, a configuration directory and a configuration properties file (some configuration options may reference other files in the configuration directory):

```
$ ./bin/flume-ng agent --conf conf \
>   -f conf/agent.properties --name agent-1
```

The `agent` command tells Flume that it is starting an agent. The `--conf` specifies a configuration directory, and the `-f` switch specifies the properties file containing the agent definitions. The `-name` switch tells Flume to start the agent named `agent-1`. After the agent is started, it begins processing events according to its configuration.

Conclusion

This chapter has discussed two systems for handling data flow in a distributed environment. Both Kafka and Flume are high-performance systems with sufficient scalability to support real-time streaming. Kafka is directed toward users who are building applications from scratch, giving them the freedom to directly integrate a robust data motion system. Flume can also be used by new applications, but its design makes it well suited to environments that have existing applications that need to be federated into a single processing environment.

The temptation at this point is to consider these two systems to be somehow mutually exclusive: Kafka for new things and Flume for "legacy" things. This is simply not the case. The two systems are complementary to each other, solving different usage patterns. In a complicated environment, Kafka might be used for the bulk data motion within the system, handling the bulk logging and collecting data from outlying data sources. It might also be used to directly feed the stream processing systems introduced in Chapter 5. However, Flume would be better suited to streaming that data into persistent data stores like Elasticsearch. With Flume, those components already exist and can be used directly, whereas with Kafka they would have to be written.

Now that data is flowing through the system, it is time for it to be processed and stored. At one point, this would have been a fairly complicated process. Fortunately, the systems introduced in Chapter 5 have greatly simplified these mechanisms.

Chapter 5
Processing Streaming Data

Now that data is flowing through a data collection system, it must be processed. The original use case for both Kafka and Flume specified Hadoop as the processing system. Hadoop is, of course, a batch system. Although it is very good at what it does, it is hard to achieve processing rates with latencies shorter than about 5 minutes.

The primary source of this limit on the rate of batch processing is startup and shutdown cost. When a Hadoop job starts, a set of input splits is first obtained from the input source (usually the Hadoop Distributed File System, known as HDFS, but potentially other locations). Input splits are parceled into separate mapper tasks by the Job Tracker, which may involve starting new virtual machine instances on the worker nodes. Then there is the shuffle, sort, and reduce phase.

Although each of these steps is fairly small, they add up. A typical job start time requires somewhere between 10 and 30 seconds of "wall time," depending on the nature of the cluster. Hadoop 2 actually adds more time to the total because it needs to spin up an Application Manager to manage the job. For a batch job that is going to run for 30 minutes or an hour, this startup time is negligible and can be completely ignored for performance tuning. For a job that is running every 5 minutes, 30 seconds of start time represents a 10 percent loss of performance.

Real-time processing frameworks, the subject of this chapter, get around this setup and breakdown latency by using long-lived processes and very small batches (potentially as small as a single record, but usually larger than that). An individual job may run for hours, days, or weeks before being restarted, which amortizes the startup costs to zero. The downside of this approach is that there is some added management overhead to ensure the job runs properly for a very long period of time and to handle the communication between components of the job so that multiple machines can be used to parallelize processing.

The chapter begins with a discussion of the issues affecting the development of streaming data frameworks. The problem of analyzing streaming data is an old one, predating the rise of the current "Big Data" paradigms.

Two frameworks for implementing streaming systems are discussed: Storm and Samza. Both are complete processing frameworks and are covered in depth in this chapter. In both cases, you can construct a server infrastructure that allows for distributed processing as well as well-structured application programing interfaces (APIs) that describe how computations are moved around these servers.

Distributed Streaming Data Processing

Before delving into the specifics of implementing streaming processing applications within a particular framework, it helps to spend some time understanding how distributed stream processing works. Stream processing is ideal for certain problems, but certain requirements can affect performance to the point that it would actually be faster to use a batch system. This is particularly true of processes requiring coordination among units.

This section provides an introduction to the properties of stream processing systems. At its heart, stream processing is a specialized form of parallel computing, designed to handle the case where data may be processed only one time. However, unlike the days of monolithic supercomputers, most of these parallel-computing environments are implemented on an unreliable networking layer that introduces a much higher error rate.

Coordination

The core of any streaming framework is some sort of coordination server. This coordination server stores information about the topology of the process—which components should get what data and what physical address they occupy.

The coordination server also maintains a distributed lock server for handling some of the partitioning tasks. In particular, a lock server is needed for the initial load of data into the distributed processing framework. Data can often be read far faster than it can be processed, so even the internal partitioning of the data motion system might need to use more than one process to pull from a given partition. This requires that the clients coordinate among themselves.

Both of the frameworks discussed in this chapter use ZooKeeper to handle their coordination tasks. Depending on load, it is possible to reuse the ZooKeeper cluster being used to manage the data motion framework without much risk.

Partitions and Merges

The core element of a stream processing system is some sort of scatter-gather implementation.

An incoming stream is split among a number of identical processing pipelines. Depending on the performance characteristics of each step in the pipeline, the results of computation may be further subdivided to maintain performance.

For example, if step two of the computation requires twice as much time as step one, it would be necessary to at least double the number of work units assigned to the second step in the pipeline compared to the first. In all likelihood, more than double the amount of units would be required because communications between work units introduces some overhead into the process.

The data exchange between various components of the processing application is usually a peer-to-peer affair. A component knows it must produce data to be consumed, and most systems will establish a mechanism for transferring data between processing units and controlling the amount of parallelism at each stage of the computation.

Transactions

Transactional processing is a problem with real-time systems. Being required to maintain a transaction puts significant overhead on the system. If it is possible to operate without transactions,

do so. If this is not possible, some frameworks can provide a level of transactional support when coupled with an appropriate data motion system.

The basic mechanism for transaction handling in real-time frameworks is the rollback and retry. The framework begins by generating a checkpoint of the input stream. This is easier to do in queuing or data motion systems where each element has a unique identifier that increases monotonically over the relevant time range. In Kafka, this is achieved by recording the offset for each of the partitions in the topic(s) being used as input to the real-time processing system.

Processing Data with Storm

Storm is a stream-processing framework developed by a company called BackType, which was a marketing analytics company that was analyzing Twitter data. Twitter acquired BackType in 2011, and Storm, a core piece of the BackType technology, was adopted and eventually open-sourced by Twitter. Storm 0.9.0 is the current version and is the focus of this section because there are some new features that simplify Storm's use cases.

Storm's focus is the development and deploying of fault-tolerant distributed stream computations. To do this, it provides a framework for hosting applications. It also provides two models for building applications.

The first is “classic” Storm, which frames the application as a directed acyclic graph (DAG) called a topology. The top of this graph takes inputs from the outside world, such as Kafka. These data sources are called *spouts*. These spouts then pass the data onto units of computation called *bolts*.

The second model for building applications, called Trident, is a higher-level abstraction on top of the topology. This model is more focused on common operations like aggregation and persistence. It provides primitives focused on this type of computation, which is arranged according to their processing. Trident then computes a topology by combining and splitting operations into appropriate collections of bolts.

This chapter covers both models of computation. The second model, using the Trident domain-specific language, is generally preferred because it standardizes a number of common stream processing tasks. However, it is sometimes advantageous to break with this common model and use the underlying topology framework.

Components of a Storm Cluster

When the time comes to deploy a Storm topology in production, you must construct a cluster. This cluster consists of three server daemons, all of which must be present to function properly: ZooKeeper, the nimbus, and the supervisors. In this section, a Storm cluster capable of supporting production workloads will be constructed using these three components as shown in Figure 5.1.

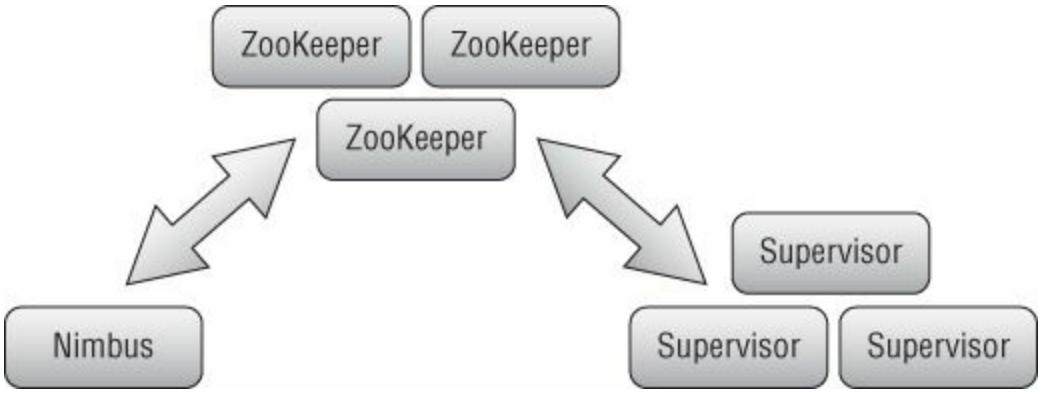


Figure 5.1

ZooKeeper

A distributed Storm cluster relies on ZooKeeper for coordinating the cluster. Communication between nodes is peer-to-peer, so the load on the ZooKeeper cluster is not very high as it is only used to manage metadata for each of the nodes. If using a data motion system that also relies on ZooKeeper,

such as Kafka, it is fine to use that same cluster for hosting Storm provided it has enough capacity.

ZooKeeper keeps track of all running topologies as well as the status of all supervisors.

The Nimbus

The nimbus is the ringleader of the Storm circus. It is responsible for the distribution of individual tasks for either spouts or bolts across the workers in the cluster. It is also responsible for rebalancing a Storm cluster in the event a supervisor has crashed.

At first glance, the nimbus would appear to be a single point of the failure, like the `JobTracker` or `NameNode` in a traditional Hadoop cluster. For Storm, this is not strictly the case. When the nimbus goes down, existing topologies continue to function unimpeded. The Nimbus is not involved in the moment-to-moment processing—just the distribution of processing tasks to the supervisor nodes.

When the nimbus crashes, the cluster is no longer able to manage topologies. This includes starting new topologies and rebalancing existing topologies in the event of failure. It is recommended that nimbuses be restarted as quickly as possible, but it will not immediately bring down a cluster.

The Supervisors

Supervisors in Storm are similar to `TaskTrackers` in Hadoop. They are servers that run on each of the worker nodes and manage the execution of local tasks. Each of the local tasks is either a spout or a bolt, and there can be many local tasks running under the control of each supervisor.

During normal operation, if the supervisor goes down then all of the tasks running under its control are also lost. Because Storm is designed to be distributed and fault tolerant, when this happens the Nimbus restarts these tasks under the control of a supervisor.

When a supervisor returns to a cluster (or more supervisors are added), running topologies are not automatically rebalanced. To rebalance the cluster after adding more nodes, the `rebalance` command should be run from the Storm command-line interface.

Configuring a Storm Cluster

This section demonstrates how to configure a Storm cluster for distributed mode processing. On distributed hardware, this would be the mode for production processing. It is also possible to run all services on a single machine, although it is often easier to use the local cluster mode described in the next section for development and debugging.

Prerequisites

To begin, obtain a Storm binary archive from http://storm-project.net website. The most recent version at the time of writing is 0.9.0.1, and this is the version recommended for installation.

If, for some reason, it is necessary to use an older version of Storm, you need to install ZeroMQ (also written 0MQ). This restriction is removed in 0.9.0 thanks to the Netty transport, but earlier versions of Storm require this package for nodes to be able to communicate with each other.

On Linux systems, ZeroMQ is often available from the package management system, but you should take care to install version 2.1.7. This is the only version that is known to work with Storm. Mac OS X users can obtain 0MQ from package libraries such as Homebrew or MacPorts, but Windows users have many more problems. It is easier to simply upgrade to Storm 0.9.0 than it is to make an older version of Storm work on Windows.

Configuring Storm

Storm keeps its base configuration in the `conf/storm.yaml` file. This does not contain information about topologies, but it does keep track of a few important features: the location of the ZooKeeper servers, the location of the nimbus server, and the transport mechanism.

It is assumed that a ZooKeeper cluster similar to the one configured in Chapter 3 is currently running. Like all ZooKeeper clients, Storm needs to know the name of at least one ZooKeeper server. More servers provide failover in case that server is temporarily unavailable. The servers are provided as a YAML list for the configuration parameter `storm.zookeeper.servers`:

```
storm.zookeeper.servers:
  - "zookeeper-node1.mydomain.net"
  - "zookeeper-node2.mydomain.net"
```

If you're running a distributed server on a single machine, just use `localhost` here:

```
storm.zookeeper.servers:
  - "localhost"
```

To be able to submit topologies, Storm also needs to know the location of the nimbus server:

```
nimbus.host: "storm-nimbus.mydomain.net"
```

If you're running everything locally, this should also be set to `localhost`:

```
nimbus.host: "localhost"
```

If you're using Storm 0.9.0, use the Netty transport instead of the ZeroMQ transport. Contributed by Yahoo!, the Netty transport offers better performance and easier setup and should be the default on all clusters. However, it is not the default for Storm clusters, so you need to configure it by adding the following statements to the configuration file:

```
storm.messaging.transport: "backtype.storm.messaging.netty.Context"
storm.messaging.netty.server_worker_threads: 1
storm.messaging.netty.client_worker_threads: 1
storm.messaging.netty.buffer_size: 5242880
storm.messaging.netty.max_retries: 100
storm.messaging.netty.max_wait_ms: 1000
storm.messaging.netty.min_wait_ms: 100
```

This change was actually much larger than simply adding Netty support. Storm now technically supports the ability to add other sorts of transport mechanisms.

Distributed Clusters

The default mode for Storm is to run as a distributed cluster of servers. One machine serves as the host for the nimbus and, usually, the Storm user interface (UI). Other machines run the supervisors and are added as capacity is needed. This section describes the steps needed to start this type of cluster, even if all the “machines” are actually running on the same development box.

Starting the Servers

If this is a development machine, Storm contains a development ZooKeeper install that can be run. You should start this before starting any of the servers:

```
storm-0.9.0.1$ ./bin/storm dev-zookeeper
```

It is useful to keep this running in its own console as it logs out connections and disconnections.

Now each of the servers can be started. On the machine acting as the Nimbus, start the Nimbus server using the `bin/storm nimbus` command. Additionally, start the web interface. It helps to manage the topologies:

```
storm-0.9.0.1$ ./bin/storm nimbus &
storm-0.9.0.1$ ./bin/storm ui &
```

Of course, in a production environment these should be placed into a startup script of some kind.

Similarly, the Storm supervisor should be started on all of the worker machines:

```
storm-0.9.0.1$ ./bin/storm supervisor &
```

If all has gone well, the Storm UI should reflect the correct number of supervisors. For example, a cluster running on a development machine should look something like Figure 5.2.

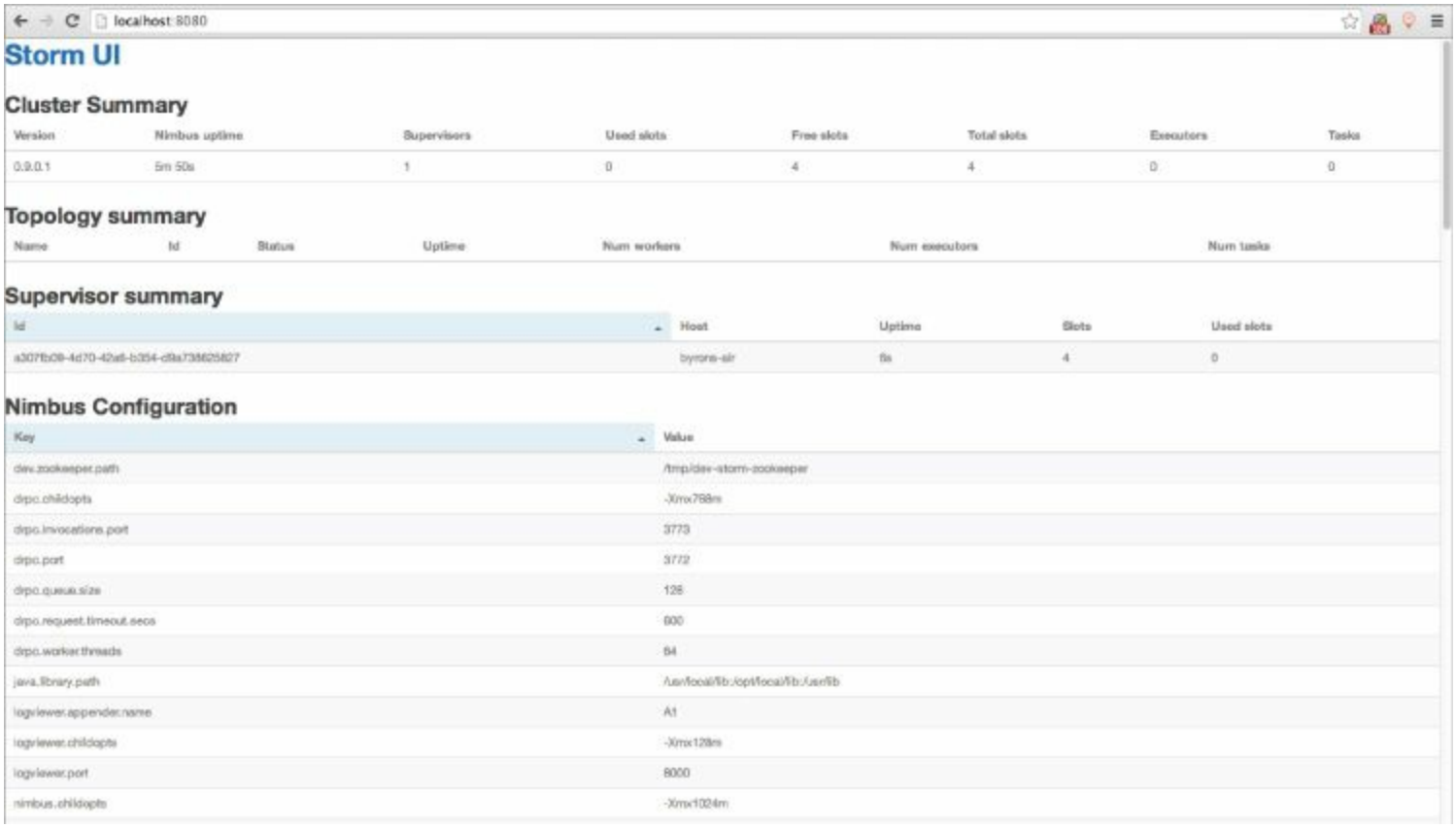


Figure 5.2

Controlling Topologies

Submitting a topology to a Storm cluster is similar to submitting a Hadoop job. The commands are even similar: `storm jar` instead of `hadoop jar`.

Topologies themselves are actually submitted in their main class using the `StormSubmitter` class:

```
Config conf = new Config();
StormSubmitter.submitTopology("my-topology", conf, topology);
```

The `Config` object can be used as is or overridden to set topology-specific configuration parameters. A main class implementing this call can be executed in the proper configuration context using `storm jar`:

```
$ ./bin/storm jar \
```

```
> streaming-chapter-5-1-shaded.jar \
> wiley.streaming.storm.ExampleTopology
246 [main] INFO backtype.storm.StormSubmitter
  - Jar not uploaded to master yet. Submitting jar...
346 [main] INFO backtype.storm.StormSubmitter - Uploading topology
  jar streaming-chapter-5-1-shaded.jar to
  assigned location: storm-local/nimbus/inbox/stormjar-18e1fded
  -074d-4e51-a2b1-5c98b2e133a6.jar
1337 [main] INFO backtype.storm.StormSubmitter - Successfully uploaded
  topology jar to assigned location:
  storm-local/nimbus/inbox/stormjar-18e1fded-074d
  -4e51-a2b1-5c98b2e133a6.jar
1338 [main] INFO backtype.storm.StormSubmitter - Submitting topology
  my-topology in distributed mode with conf {}
```

After a Topology has been submitted, it can be stopped using the Storm command-line client. The `kill` command along with the topology identifier stops the topology entirely. It can be paused and restarted without fully stopping using `deactivate` and `activate`, respectively. You can also pause and restart the topology from the Storm web interface by selecting the topology in the UI.

Starting a Topology Project

The easiest way to get started developing a topology project is to use Maven to include all of the requisite dependencies. The following is a minimal `pom.xml` file needed to start a Java-based Storm topology. It includes the minimal dependencies, including the link to the `clojars.org` repository where Storm's maven packages are hosted. It also includes the `shade` plug-in, which is used to build so-called “fat jars” that add other dependencies not included with Storm itself:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>group</groupId>
  <artifactId>storm-project</artifactId>
  <version>1</version>
  <repositories>
  <repository>
    <id>clojars.org</id>
    <url>http://clojars.org/repo</url>
  </repository>
  </repositories>
  <dependencies>
    <dependency>
      <groupId>storm</groupId>
      <artifactId>storm</artifactId>
      <version>0.9.0.1</version>
      <scope>test</scope>
    </dependency>
</dependencies>
  <build>
    <plugins>
      <plugin> <!-- Shade Plugin for building Fat Jars -->
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-shade-plugin</artifactId>
        <version>2.0</version>
        <configuration>
          <shadedArtifactAttached>true</shadedArtifactAttached>
        </configuration>
```

```
          <executions>
            <execution>
              <phase>package</phase>
              <goals><goal>shade</goal></goals>
            </execution>
          </executions>
        </plugin>
      </plugins>
    </build>
  </project>
```

This is all that is required to start developing Storm topologies in Java.

Local Clusters

When developing and testing Storm topologies, it is not necessary to have a complete cluster available, although that can be useful for tracking down packaging problems.

For testing purposes, Storm makes a special type of cluster called the LocalCluster available for use. This allows the entire topology to run in a single Java Virtual Machine (JVM) instance. The cluster is a "full" cluster in the sense that it executes inside a supervisor, and there is an embedded ZooKeeper instance running as opposed to a unit test environment where the server components are "mocked out."

To use a local cluster in a test framework is quite simple, simply configure a LocalCluster and then submit the topology to it as usual:

```
Config conf = new Config();
conf.setDebug(true);
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("example", conf, builder.createTopology());
Thread.sleep(60000);
```

The `sleep` command is needed because the cluster topology actually executes on background threads. Without it, the main method would finish and the process would shut down immediately without actually executing the topology. Putting a delay into the main class lets the topology execute for a little while.

Storm Topologies

The topology is Storm's mechanism for computational organization. The topology is represented by something called a directed acyclic graph (DAG).

A graph, of course, is a structure that contains vertices (nodes) connected together with edges. In this case, the edges are directed, which means that data only flows in a single direction along the edge.

These graphs are also acyclic, which means that the arrangement of edges may not form a loop. Otherwise, the data would flow infinitely around the topology. A simple topology is shown in Figure 5.3, with an input spout and two layers of bolts. The Storm framework can increase the bolt parallelism as needed to maintain performance.

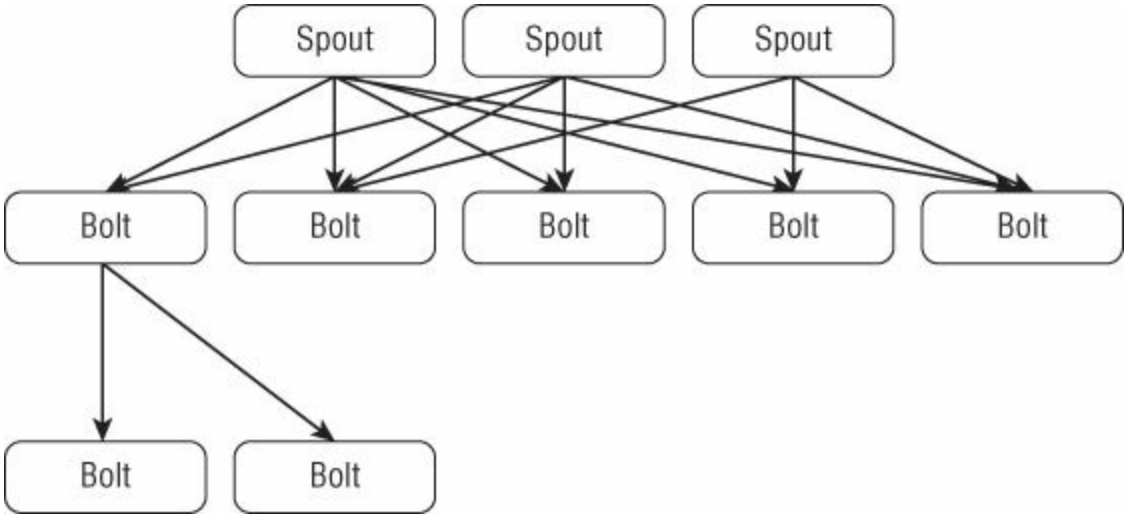


Figure 5.3

Topologies are built in Storm using the `TopologyBuilder` class and created by a call to `createTopology`:

```
public static void main(String[] args) {
  TopologyBuilder builder = new TopologyBuilder();
  defineTopology(builder,args);
  StormTopology topology  = builder.createTopology();
```

The `defineTopology` method is where the graph itself is constructed. Each vertex in the topology consists of a unique name and the definition of either a spout or a bolt. A spout is Storm's data input mechanism and is added using the `setSpout` method:

```
public static void defineTopology(TopologyBuilder builder
  ,String[] args) {
    builder.setSpout("input", new BasicSpout());
```

This creates a spout named “input” and returns a `SpoutDeclarer` that is used to further configure the `Spout`. Most topologies only contain a single `IRichSpout` type, but Storm often creates more than one `Spout` task to improve the parallelism and durability of the input process. The maximum number of tasks and other related variables are all set using the returned `SpoutDeclarer`.

Bolts are Storm's basic unit of computation. Like spouts, they must have a unique name, and they are added via the `setBolt` method. For example, this adds a new bolt named “processing” to the topology:

```
builder.setBolt("processing", new BasicBolt())
       .shuffleGrouping("input");
```

The `setBolt` method returns a `BoltDeclarer` object that is used to connect the bolt to the rest of the topology. Bolts are attached to the topology by using one of the grouping methods to define an input vertex. In this case the “input” spout vertex defined earlier is used.

Storm creates several tasks for each defined `Bolt` to improve parallelism and durability. Because this may affect computations like aggregation, the grouping methods define how data is sent to each of the individual tasks. This is similar to how partition functions are used in map-reduce implementations, but with a bit more flexibility. Storm offers a number of grouping methods to organize the flow of data in the topology.

Shuffle Groupings

In the earlier example, the `shuffleGrouping` method is used to distribute data among the `Bolt` tasks. This method randomly shuffles each data element, called a `Tuple`, among the `Bolt` tasks such that each task gets roughly the same amount of data.

Field Groupings

Another common grouping is the `fieldsGrouping` method. This grouping uses one or more of the named elements of a tuple, which are defined by the input vertex, to determine the task that will receive a particular data element. This is essentially equivalent to a SQL `GROUP BY` clause and is often used when implementing aggregation bolts. To group data from "input" by `key1` and `key2`, add the bolt as follows:

```
builder.setBolt("processing", new BasicBolt())
  .fieldsGrouping("input", new Fields("key1","key2"));
```

All and Global Groupings

The `allGrouping` and `globalGrouping` methods are exact opposites of each other. The `allGrouping` method ensures that all tuples in an event stream are transmitted to all running tasks for a particular bolt. This is occasionally useful, but it multiplies the number of events by the number of running tasks, so use it with care.

The `globlaGrouping` method does the opposite. It ensures that all tuples from a stream go to the bolt with the lowest numbered identifier. All bolt tasks receive an identifier number, but this ensures only one of them will be used. This should also be used with care because it effectively disables Storm's parallelism.

Direct Groupings

The `directGrouping` method is a special form of grouping that allows the output `Bolt` or `Spout` to decide which `Bolt` task receives a particular `Tuple`. This requires that the stream be declared to be direct when the bolt is created. The producer bolt must also use the `emitDirect` method instead of the `emit` method.

The `emitDirect` method takes an extra parameter that identifies the destination `Bolt`. The list of valid identifiers can be obtained from the `TopologyContext` when implementing a `Bolt`. This is shown in greater detail in the section that details implementing `Bolts`.

Custom Groupings

When all else fails, it is also possible to implement a custom grouping method. To do this, create a class that implements the `CustomStreamGrouping` interface.

This interface contains two methods. The first method, `prepare`, is called when the topology is instantiated and lets the grouping method assemble any metadata it may need to perform its tasks. In particular, this method receives the `targetTasks` variable, which is the list of Bolts that subscribe to the stream being grouped.

The second method is the `chooseTasks` method, which is the workhorse of the class. This method is called for each `Tuple` and the method returns a list of bolt tasks that should receive the `Tuple`.

A Round-Robin Custom Stream Grouping

This example implements a round-robin mechanism for distributing data among bolt tasks. It is no more efficient than a `shuffleGrouping`, but it demonstrates the technique.

First, the class is defined with a serialization version number. Storm makes heavy use of Java serialization when configuring a topology, so it is important to pay attention to what will be serialized and what will not. The three class fields are all set during the `prepare` statement so they should be defined as `transient` so they will not be included in the serialization:

```
public class RoundRobinGrouping implements CustomStreamGrouping {
 /**
   *
   */
  private static final long serialVersionUID = 1L;
  transient List<Integer> targetTasks;
  transient int           nextTask;
  transient int           numTasks;
```

Next, the `prepare` statement initializes each of the transient variables. A more complicated grouping might inspect the topology itself to assign weightings to tasks and other options, but this simple round-robin method just needs to hold onto the list:

```
  public void prepare(WorkerTopologyContext context,
              GlobalStreamId stream,
              List<Integer> targetTasks) {
    this.targetTasks = targetTasks;
    this.numTasks    = targetTasks.size();
    this.nextTask    = 0;
  }
```

Finally, the `chooseTasks` method is implemented to rotate between the various tasks:

```
  public List<Integer> chooseTasks(int taskId, List<Object> values) {
    LinkedList<Integer> out = new LinkedList<Integer>();
    out.add(targetTasks.get(nextTask));
    nextTask = (nextTask + 1) % numTasks;
    return out;
  }
```

Implementing Bolts

Now that data can be passed to bolts appropriately, it is time to understand how bolts themselves work. As shown in the previous section, bolts receive data from other bolts or spouts within the topology. They are event driven so they cannot be used to retrieve data; that is the role of the spout.

Although it is possible to implement bolts by implementing either the `IBasicBolt` or `IRichBolt` interfaces, the recommended method is to extend the `BaseBasicBolt` or `BaseRichBolt` classes. These base classes provide simple implementations for most of Storm's boilerplate methods.

Rich Bolts

To implement a `RichBolt`, start by extending this `RichBaseBolt` abstract class and implementing the three required methods. Like the custom grouping class from the previous section, the first is a prepare method that passes in information about the topology. For bolts, an extra parameter is added: the `collector` object. This object manages the interaction between the bolt

and the topology, especially transmitting and acknowledging tuples.

In a rich bolt, the `collector` is usually placed into an instance variable for use in the execute method. The `prepare` method is also a good place to create an object that cannot be serialized because they do not implement the `Serialization` interface. It is a good habit to declare all variables that are initialized by the prepare method as transient. While not all variables require the transient keyword, they can often be initialized and serialized unintentionally, leading to bugs that are difficult to track down. The following code implements a bolt that captures the output collector:

```
public class RichEmptyBolt extends BaseRichBolt {
  private static final long serialVersionUID = 0L;
  private transient OutputCollector collector;
  public void prepare(Map stormConf, TopologyContext context,
                      OutputCollector collector) {
    this.collector = collector;
  }
```

Most bolts produce one or more streams of output. A stream is a collection of `Tuple` objects that Storm serializes and passes to other bolt implementations as defined within the topology. These `Tuple` objects do not have a formal schema; all elements of the tuple are considered to be generic `Objects`, but the elements are named. These names are used both in the bolt's `execute` method and by certain grouping classes, such as `fieldGrouping`. The order of these named fields is defined in the bolt's `declareOutputFields` method. By default, the bolt has a single stream that is defined by calling `declare` on the `OutputFieldsDeclarer` that is passed into the method:

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("first","second","third"));
}
```

In addition to the default stream, a bolt can define other named streams for output. Using the `declareStream` method, give the stream a name and define the fields of the tuple. For example, to take the default stream and split it into two parts, declare two output streams:

```
declarer.declareStream("car", new Fields("first"));
declarer.declareStream("cdr", new Fields("second","third"));
```

When defining the topology, these additional streams are added to the grouping calls after the source name. For example, attaching a bolt to the `cdr` stream using a field grouping looks like this:

```
builder.setBolt("empty", new RichEmptyBolt());
builder.setBolt("process", new BasicBolt())
  .fieldsGrouping("empty", "cdr", new Fields("second"));
```

Processing incoming tuples is handled by the `execute` method inside the bolt. In `RichBolt` implementations, it has a single `Tuple` argument. This class implements the Java `List<Object>` collection interface, but also provides higher-level access to the fields as declared by the source bolt.

A bolt can do whatever it likes with the input tuple. For example, to produce the output streams declared in the earlier example, assuming the input tuple has elements of the same name, the `execute` method would look like this:

```
[[OPEN-LW-CODE80]]public void execute(Tuple input) {
  List<Object> objs = input.select(new Fields("first","second","third"));
  collector.emit(objs);
  collector.emit("car", new Values(objs.get(0)));
  collector.emit("cdr", new Values(objs.get(1),objs.get(2)));
```

```
    collector.ack(input);
  }[[CLOSE-LW-CODE80]]
```

The first line of this example is using a feature of the `Tuple` wrapper to extract a subset of the `Tuple` object into a known ordering. This ordering is the same as the declared output stream, so it can simply be emitted as is. The `emit` method assumes that the output is an object that implements `List<Object>`, which is converted into a `Tuple` after it arrives at the next bolt.

Next, the other two streams, `car` and `cdr`, are emitted. Because they only take a subset of the default stream, new output arrays need to be constructed. The default Java constructors for `List` objects are generally verbose, so Storm provides a class called `Values`. This class implements the `List<Object>` interface and provides convenience constructors to make it easy to produce new Tuple objects.

Finally, the tuple is acknowledged via `ack(input)`. This tells Storm that the tuple has been handled, which is important when the bolt is being used in a transactional fashion. If the processing of the tuple fails for some reason, the bolt may use `fail(input)` instead of `ack(input)` to report this fact.

The collector object also has a `reportError` method, which can be used to report on non-fatal exceptions in the flow. This method takes a `Throwable` object, and the message from the exception is reflected in the Storm UI. For example, a bolt that handles URLs can report on parsing errors:

```
public void execute(Tuple input) {
  try {
    URL url = new URL(input.getString(0));
    collector.emit(new Values(url.getAuthority(),
                              url.getHost(),
                              url.getPath()));
  } catch (MalformedURLException e) {
    collector.reportError(e);
    collector.fail(input);
  }
}
```

A Filter and Split Bolt

This example shows a bolt that takes an input stream and splits it into several different streams according to a regular expression on a field. It also shows how to introspect the incoming tuples to define the output streams programmatically.

The filters are defined during the configuration of the topology. The bolt defines a simple internal class called `FilterDefinition`, which holds the output stream, the name of the field to check, as well as the regular expression that will be evaluated against the field's value. This class implements `Serializable` so it will be properly serialized when the bolts are distributed across the cluster.

The `filter` function itself is implemented in a "chainable" style to make it easy to use in topology definitions:

```
public class FilterBolt extends BaseRichBolt {
  private static final long serialVersionUID = -7739856267277627178L;
  public class FilterDefinition implements Serializable {
    public String  stream;
    public String  fieldName;
    public Pattern regexp;
    public Fields  fields;
    private static final long serialVersionUID = 1L;
  }
  ArrayList<FilterDefinition> filters
  = new ArrayList<FilterDefinition>();
  public FilterBolt filter(String stream,String field,String regexp,
    String... fields) {
    FilterDefinition def = new FilterDefinition();
    def.stream = stream;
    def.fieldName = field;
    def.regexp = Pattern.compile(regexp);
    def.fields = new Fields(fields);
    filters.add(def);
    return this;
  }
```

This bolt's `prepare` method captures the collector output as well as preparing the filters for further use. In this case, the unique input streams are inspected to determine the fields present in their output tuples. If they match a particular filter, that filter is bound to that particular stream. As a performance improvement, the index of the filter element is retrieved as well:

```
  public class FilterBinding {
    public FilterDefinition filter;
    public int              fieldNdx;
  }
  transient OutputCollector                          collector;
  transient Map<GlobalStreamId,List<FilterBinding>> bindings;
public void prepare(Map stormConf, TopologyContext context,
      OutputCollector collector) {
    this.collector = collector;
    bindings = new HashMap<GlobalStreamId,List<FilterBinding>>();
    for(GlobalStreamId id : context.getThisSources().keySet()) {
      Fields fromId = context.getComponentOutputFields(id);
      ArrayList<FilterBinding> bounds =
       new ArrayList<FilterBinding>();
      for(FilterDefinition def : filters) {
        if(fromId.contains(def.fieldName)) {
```

```
            FilterBinding bind = new FilterBinding();
            bind.filter = def;
            bind.fieldNdx = fromId.fieldIndex(def.fieldName);
            bounds.add(bind);
          }
        }
        if(bounds.size() > 0) bindings.put(id, bounds);
      }
    }
```

Ideally, the analysis done during the prepare phase to determine the input fields could be used to programmatically determine the output fields. Unfortunately, the `declareOutputFields` method is called during topology construction and not during topology initialization. As a result, the tuple's elements to include from each filter are specified when the filter is defined and simply returned by `declareOutputFields`:

```
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    for(FilterDefinition def : filters)
      declarer.declareStream(def.stream, def.fields);
  }
```

The implementation of the filter itself makes use of the filter bindings created in the `prepare` method. Each `Tuple` passed to a `Bolt` travels with metadata about the origin of the `Tuple`, including the `GlobalStreamId`. This is obtained using `getSourceGlobalStreamId` and is used to look up the bound filters for this particular `Tuple`. If there are application filters, the tuple is then applied to each filter in the list and emitted to the appropriate stream if it matches, using the `select` method on the tuple to extract a tuple that matches the configured `Fields` element:

```
  public void execute(Tuple input) {
      List<FilterBinding> bound
        = bindings.get(input.getSourceGlobalStreamid());
      if(bound != null && bound.size() > 0) {
        for(FilterBinding b : bound) {
          if(b.filter.regexp.matcher(
              input.getString(b.fieldNdx)
             ).matches()) {
            collector.emit(b.filter.stream,
              input.select(b.filter.fields));
          }
        }
      }
    collector.ack(input);
    }
```

Basic Bolts

If a bolt meets certain criteria it may also be implemented as a `BasicBolt`, which removes some of the boilerplate involved in implementing a `RichBolt`. The `BasicBolt` implementation does not have a prepare step. It is assumed that the only nonserialized configuration required is to capture the `collector` object.

A `BasicBolt` also assumes that all tuples will be acknowledged using `ack`. This is implemented fairly deep in the Storm code, rather than being part of the `BasicOutputCollector` implementation.

A Logging Bolt

The logging bolt is a very simple `Bolt` implementation that is useful when testing topologies. All it does is print the results of an input tuple to the console. This can be very useful when testing topologies, and it is easy to implement with a `BasicBolt`. This bolt doesn't produce any data, and just uses the fact that `Tuple` implements a reasonable `toString` method:

```
public class LoggerBolt extends BaseBasicBolt {
  private static final Logger LOG
   = Logger.getLogger(LoggerBolt.class);
  private static final long serialVersionUID = 1L;
  public void execute(Tuple input, BasicOutputCollector collector) {
    LOG.info(input.toString());
  }
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
  }
}
```

Implementing and Using Spouts

The `Spout` is a special form of Storm topology element that is responsible for retrieving data from the outside world. Unlike `Bolts`, there is only an `IRichSpout` interface to implement and no base class to derive from.

When a `Spout` is created, the open method is called first. This allows the `Spout` to configure any connections to the outside world, such as connections to queue or data motion servers. Like a `Bolt`, the `Spout` should capture its `SpoutOutputCollector` from the open method for later use:

```
public class EmptySpout implements IRichSpout {
  transient SpoutOutputCollector collector;
  public void open(Map conf, TopologyContext context,
      SpoutOutputCollector collector) {
    this.collector = collector;
  }
```

Like a bolt, a spout also defines output streams. There is the usual default stream, defined by calling `declare`. Additionally, named streams can be defined by calling `declareStream`, the same as bolts:

```
public void declareOutputFields(OutputFieldsDeclarer declarer) {
  declarer.declare(new Fields("first","second","third"));
  declarer.declareStream("errors",new Fields("error"));
}
```

Spouts are, by definition, pull-based polling interfaces. The Storm infrastructure repeatedly calls the `nextTuple` method, which is similar to the `execute` method in a Bolt:

```
public void nextTuple() {
  Utils.sleep(100);
  collector.emit(new Values("one","two","three"));
}
```

Because Storm does use a polling interface, it is recommended that a small sleep command be added when there is no data to emit. This helps reduce the system load when the topology is idle. In the simple example here, a small delay serves much the same purpose.

In addition to being started, topologies can also be stopped, as well as paused and resumed. This

mostly affects the processing of the spout, which has methods that can be used to start and stop the polling process. Additionally, there is a close method when the topology shuts down to allow for the clean shutdown of consumer interfaces:

```
public void close() {
}
public void activate() {
}
public void deactivate() {
}
```

Finally, topologies can implement transactional semantics with an appropriate spout implementation. When a bolt executes an `ack` or `fail` command on a `Tuple` object, this event is passed up to the spout to allow the tuple's processing to either be committed or reprocessed:

```
public void ack(Object msgId) {
}
public void fail(Object msgId) {
}
```

In practice, these events are a holdover from older versions of Storm. For new transactional topologies, the recommendation is to use the Trident interface available from Storm 0.8.0 onward. This alternative interface, discussed later in this chapter, provides a cleaner mechanism for implementing transactional semantics in a topology.

The only other method to implement is the `getComponentConfiguration` method. This is used for additional configuration of spouts and will normally be unused and can simply return `null`:

```
public Map<String, Object> getComponentConfiguration() {
  return null;
}
```

A “Lorem Ipsum” Spout

Storm includes several mechanisms for testing topologies, which are covered a bit later on, but it is often useful to have a spout that can generate random data.

“Lorem Ipsum” is famous nonsense text that is often used by designers to simulate blocks of text for layout. The text itself is a mangled Latin text roughly a paragraph long, but modern implementations can generate arbitrarily long pieces of text.

There is a Java class, `LoremIpsum`, available via the Maven Central repository that can generate strings of arbitrary length from this original paragraph.

```
<dependency>
  <groupId>de.sven-jacobs</groupId>
  <artifactId>loremipsum</artifactId>
  <version>1.0</version>
</dependency>
```

To begin, the spout is defined with field names for the tuple that will be generated:

```
public class LoremIpsumSpout implements IRichSpout {
  private static final long serialVersionUID = 1L;
  String[] fields;
  public LoremIpsumSpout tuple(String...fields) {
    this.fields = fields;
    return this;
  }
  int maxWords = 25;
  public LoremIpsumSpout maxWords(int maxWords) {
    this.maxWords = maxWords;
    return this;
  }
```

When the spout's open method is called, the transient `LoremIpsum` class is instantiated because it does not implement `Serializable` and therefore cannot be configured prior to starting the topology. As usual, the `collector` element is also captured for later use:

```
transient LoremIpsum           ipsum;
transient Random               rng;
transient SpoutOutputCollector collector;
public void open(Map conf, TopologyContext context,
  SpoutOutputCollector collector) {
  this.collector = collector;
  ipsum = new LoremIpsum();
  rng   = new Random();
}
```

Because this Spout is completely self-contained, the transactional methods `ack` and `fail`, as well as the lifecycle management methods—`activate`, `deactivate`, and `close`—can be left as stub implementations.

The `nextTuple` implementation sleeps for a moment, to reduce load during test cases, and then emits a tuple of the appropriate length containing “lorem ipsum” text of varying length:

```
public void nextTuple() {
  Utils.sleep(100);
  ArrayList<Object> out = new ArrayList<Object>();
  for(String s : fields)
```

```
      out.add(ipsum.getWords(rng.nextInt(maxWords)));
    collector.emit(out);
  }
```

To test this `Spout`, use a simple `LocalCluster` implementation to see it in action. First construct a topology with two `LoremIpsum` spouts:

```
  TopologyBuilder builder = new TopologyBuilder();
  builder.setSpout("spout1", new LoremIpsumSpout()
    .tuple("first","second","third"));
  builder.setSpout("spout2", new LoremIpsumSpout()
    .tuple("second","third","forth"));
```

Then, just to give the topology something to do, attach a `FilterBolt` from the last section. Have it filter on two well-known parts of the "lorem ipsum" text. Note that the filter takes in both spouts:

```
  builder.setBolt("filter", new FilterBolt()
   .filter("ipsum", "first", ".*ipsum.*","first")
   .filter("amet","second",".*amet.*","second")
  ).shuffleGrouping("spout1")
     .shuffleGrouping("spout2")
    ;
```

To see the filtering functionality, use a `LoggerBolt` to print the output. It also identifies the source stream to show that the filter actually works:

```
    builder.setBolt("print", new LoggerBolt())
    .shuffleGrouping("filter","amet")
    .shuffleGrouping("filter","ipsum");
```

This is all then submitted to a `LocalCluster` for execution:

```
    Config conf = new Config();
    LocalCluster cluster = new LocalCluster();
    cluster.submitTopology("example", conf, builder.createTopology());
```

Finally, have the main class sleep for a bit. Because `LocalCluster` is running an embedded topology, most of the work is being done in background threads:

```
    //Sleep for a minute
    Thread.sleep(60000);
```

After running the topology, the output should look something like this:

```
  6206 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
    - source: filter:2,
    stream: ipsum, id: {},
    [Lorem ipsum dolor sit amet, consetetur sadipscing elitr,
    sed diam nonumy eirmod tempor invidunt ut]
  6306 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
    - source: filter:2,
    stream: amet, id: {}, [Lorem ipsum dolor sit amet,
    consetetur sadipscing elitr, sed diam nonumy eirmod tempor]
  6314 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
    - source: filter:2,
    stream: amet, id: {}, [Lorem ipsum dolor sit amet, consetetur
    sadipscing elitr, sed diam nonumy eirmod tempor invidunt ut
    labore et dolore magna aliquyam]
  6407 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
    - source: filter:2,
    stream: ipsum, id: {}, [Lorem ipsum dolor sit amet,
```

```
  consetetur sadipscing elitr, sed diam nonumy eirmod tempor
  invidunt ut labore et dolore magna]
6407 [Thread-22-print] INFO  wiley.streaming.storm.LoggerBolt
  - source: filter:2,
  stream: amet, id: {}, [Lorem ipsum dolor sit amet,
  consetetur sadipscing elitr, sed diam nonumy eirmod tempor
  invidunt ut labore et dolore magna aliquyam]
```

Connecting Storm to Kafka

With the introduction of Kafka 0.8, the integration between Kafka and Storm in 0.9 is in a state of flux. The `storm-kafka` spout implementation available in the `storm-contrib` library is currently targeted to version 0.7.2 of Kafka. A version of the Kafka spout designed for use with Kafka 0.8 is available through another project called `storm-kafka-0.8-plus` (https://github.com/wurstmeister/storm-kafka-0.8-plus). The project is also available via clojars.org, the same as Storm itself for inclusion in a Maven `pom.xml` file. It is included in a project using the following artifact:

```
<dependency>
  <groupId>net.wurstmeister.storm</groupId>
  <artifactId>storm-kafka-0.8-plus</artifactId>
  <version>0.2.0</version>
</dependency>
```

To use the `Spout`, a `KafkaConfig` object is first configured to point to the appropriate set of brokers and the desired topic. For example, the following code obtains the list of brokers from ZooKeeper. It then attaches itself to the `wikipedia-raw` topic using the `storm` Consumer Group:

```
BrokerHosts     hosts   = new ZkHosts("localhost");
SpoutConfig     config  = new SpoutConfig(hosts,
  "wikipedia-raw", "", "storm");
config.scheme = new SchemeAsMultiScheme(new StringScheme());
```

The `config.scheme` entry tells the spout to interpret incoming messages as strings rather than some more exotic entry. Other encodings would require a different scheme implementation specific to that format.

To use this in a topology, a `KafkaSpout` is created and then used like any other spout. This simple test topology reports on the events being produced on the Wikipedia-raw topic:

```
TopologyBuilder builder = new TopologyBuilder();
builder.setSpout("kafka", new KafkaSpout(config), 10);
builder.setBolt("print", new LoggerBolt())
  .shuffleGrouping("kafka");
Config conf = new Config();
LocalCluster cluster = new LocalCluster();
cluster.submitTopology("example", conf, builder.createTopology());
//Sleep for a minute
Thread.sleep(60000);
```

Connecting Storm to Flume

There is a fundamental “impedance mismatch” between Storm and Flume. Storm is a polling consumer, assuming a pull model. Flume's fundamental sink model is push based. Although some attempts have been made, there is no accepted mechanism for directly connecting Flume and Storm.

There are solutions to this problem, but all of them involve adding another queuing system or data

motion system into the mix. There are two basic approaches that are used "in the wild." The first is to use queuing systems that are compatible with the Advanced Message Queuing Protocol (AMQP). For Flume applications, the RabbitMQ queuing system is popular because there exists a Flume sink developed by Kenshoo (http://github.com/kenshoo/flume-rabbitmq-sink). Storm's `AMQPSpout` is then used to read from RabbitMQ.

The other option is to use a Kafka sink for Flume and then integrate Storm with Kafka as described in the previous section. At this point the only real reason to use Flume is to integrate with an existing infrastructure. You can find the Flume/Kafka sink at `https://github.com/baniuyao/flume-ng-kafka-sink`.

Distributed Remote Procedure Calls

In addition to the development of standard processing topologies that consume data from a stream, Storm also includes facilities for implementing Distributed Remote Procedure Calls (DRPC), which are used to easily implement distributed processing services, such as performing a large number of expensive calculations. Using a special spout and bolt combination, these topologies implement a complicated procedure that can be distributed across multiple machines.

This is not strictly something that is typically done in a real-time streaming analysis scenario, but it can certainly be used to scale parts of a real-time analysis pipeline. It can also be used to build component-based services that make an environment easier to manage.

The DRPC Server

DRPC requests are managed by a separate Storm server that manages the communication between the DRPC client and the Storm topology servicing those requests. In a distributed environment, this server is started with the Storm command-line utility:

```
$ ./bin/storm drpc
```

In testing environments, a Local DRPC server can be started. This is similar to the `LocalCluster` option used for testing:

```
TopologyBuilder builder = new TopologyBuilder();
LocalDRPC drpc = new LocalDRPC();
```

Writing DRPC Topologies

A DRPC topology is just like any other Storm topology, relying on a special `Spout` and `Bolt` combination to enact the DRPC functionality. The topology takes in a single `String` value via the Spout and returns a single `String` value via the `ReturnResults` bolt. For example, the following topology implements a "power" function that exponentiates a comma-separated list of doubles corresponding to the base value and the exponent. It returns a string representing the power output:

```
//Omit the drpc argument when submitting to the cluster
builder.setSpout("drpc", new DRPCSpout("power", drpc));
builder.setBolt("split", new SplitBolt()).shuffleGrouping("drpc");
builder.setBolt("power", new PowerBolt()).shuffleGrouping("split");
builder.setBolt("return", new ReturnResults()).shuffleGrouping("power");
```

The `SplitBolt` implementation splits the input comma-separated list into two `Double` values. It also passes along the *`return-info`* element of the tuple, which is used by `ReturnResults` to

submit the result back to the DRPC server:

```
public class SplitBolt extends BaseBasicBolt {
  private static final long serialVersionUID = -3681655596236684743L;
  public void execute(Tuple input, BasicOutputCollector collector) {
    String[] p = input.getString(0).split(",");
    collector.emit(
      new Values(
        Double.parseDouble(p[0]),
        Double.parseDouble(p[1]),
        input.getValue(1)
      )
    );
  }
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("x","y","return-info"));
  }
}
```

The `PowerBolt` implementation takes the *x* and *y* values and computes the power function. It returns this value as a `String` as required by the DRPC power and also makes sure to transfer the *return-info* object:

```
public class PowerBolt extends BaseBasicBolt {
  private static final long serialVersionUID = 1L;
  public void execute(Tuple input, BasicOutputCollector collector) {
    collector.emit(
      new Values(
        ""+Math.pow(input.getDouble(0),
        input.getDouble(1)),
        input.getValue(2)
      )
    );
  }
  public void declareOutputFields(OutputFieldsDeclarer declarer) {
    declarer.declare(new Fields("result","return-info"));
  }
}
```

Most of the time, topologies like this are not built by hand. In older versions of Storm the `LinearDRPCTopologyBuilder` class is used to construct these topologies. It automatically adds the appropriate `DRPCSpout` spout and `ReturnResults` bolt to the topology and ensures that the topology itself is linear. In newer versions of Storm, this has been deprecated in favor of the Trident version, which is created using the `newDRPCStream` method instead of the usual `newStream` method. The Trident domain-specific language is discussed in much more detail in the next section.

Trident: The Storm DSL

With version 0.8.0, Storm introduced a new domain-specific language that is intended to be the preferred method for developing Storm topologies. This domain-specific language is called Trident. In Storm 0.9.0 it has undergone extensive development.

The goal of the Trident interface is to provide a higher-level abstraction similar to what is found in Hadoop frameworks, such as Cascading. Rather than providing the very primitive spout and bolt interface, Trident operates on streams. These streams can be manipulated with high-level concepts,

such as joins, groups, aggregates, and filters. It also adds some primitives for state management and persistence, particularly in the case of aggregation.

Underneath the hood, Trident is still using the same bolt and spout interface. The difference is that each operation defined in a Trident topology does not necessarily result in a new bolt being created. This allows the topology to achieve higher performance by combining several operations into a single physical bolt and avoiding the communication overhead between bolts.

This section covers using this high-level construct to build topologies in Storm.

Trident Streams and Spouts

Trident topologies usually operate on a `Stream` object. The stream in Trident is conceptually very similar to the stream defined by a spout or a bolt in a normal Storm topology. The difference here is that the stream's structure is defined and modified in the topology definition rather than being an intrinsic part of a bolt.

Defining the stream structure along with the topology has a couple of advantages when writing topologies. First, by separating the tuple structure from the business logic, it is easier to write reusable components. Second, the topology becomes much easier to read. Because the data flow and data definition is in the same place, it is much easier to see the manipulations. In “traditional” topologies, determining how each stream is defined involves inspecting each `Bolt` individually. The disadvantage is that it can be harder to express computations using Trident's language than it would be to explicitly define the topology. Often this is simply a matter of preference.

Streams in Trident topologies are built around a `Spout`. These spouts can be the same `IRichSpout` implementations from standard topologies, or they can be one of the more specialized Trident spouts. Using a basic spout, such as the `LoremIpsumSpout`, is as simple as defining a stream using `newStream` in the `TridentTopology` class:

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout());
```

Local Operations: Filters and Functions

When constructing topologies, Trident considers certain operations to be “partition local” operations. What this means is that a chain of such operations can be executed locally within a single bolt. This enables Trident to make use of more efficient data structures and reduce the amount of communication required.

The most basic local operation is the `Filter`, which is implemented by extending the `BaseFilter` class. A single method, `isKeep`, is implemented to decide whether or not a Tuple's processing should continue. For example, this filter implementation randomly chooses to keep half of its results:

```
public class RandomFilter extends BaseFilter {
  private static final long serialVersionUID = 1L;
  Random rng = new Random();
  public boolean isKeep(TridentTuple tuple) {
    return rng.nextFloat() > 0.5;
  }
}
```

The next simplest operation is the `Function`, which is implemented by extending the

`BaseFunction` class (some Trident tutorials incorrectly have filters extend `BaseFunction` as well, but this is not the case in 0.9.0).

In functionality, `Function` objects are closest to the `BasicBolt` in the "traditional" Storm topology. They take in a single `Tuple` and can produce zero or more tuples, which are appended to the original tuple. If no tuples are produced, the original tuple is filtered from the stream. For example, this `Function` implementation adds a random number to each incoming `Tuple`:

```
public class RandomFunction extends BaseFunction {
  private static final long serialVersionUID = 1L;
  Random rng = new Random();
  public void execute(TridentTuple tuple, TridentCollector collector) {
    collector.emit(new Values(rng.nextDouble()));
  }
}
```

Both `Filter` and `Function` classes use the `each` method to attach themselves to a stream. For filters, the `each` method takes the form of `each(inputFields,filter)`:

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout()
  .tuple("first","second","third"))
  .each(new Fields(), new RandomFilter())
;
```

Note the empty `Fields` constructor. This is because the `RandomFilter` does not depend on any fields to evaluate the filter. It does not actually modify the `Tuple` passed through, which still contains the "`first`", "`second`", and "`third`" elements.

Similarly, a function's `each` argument takes one of two forms. The first is `each(function,outputFields)`, the form used by the `RandomFunction` implementation:

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout()
  .tuple("first","second","third"))
  .each(new RandomFunction(),new Fields("x"))
;
```

The second form allows the function to take input fields, the same as a filter operation. For example, a `SquareFunction` implementation would take a single parameter:

```
public class SquareFunction extends BaseFunction {
  private static final long serialVersionUID = 1L;
  public void execute(TridentTuple tuple, TridentCollector collector) {
    collector.emit(new Values(tuple.getDouble(0)*tuple.getDouble(0)));
  }
}
```

When assembled into a topology with the `RandomFunction`, it would take the `x` field as its input and produce a `y` field as output that would be appended to the tuple:

```
TridentTopology topology = new TridentTopology();
topology.newStream("lorem", new LoremIpsumSpout()
  .tuple("first","second","third"))
.each(new RandomFunction(),new Fields("x"))
.each(new Fields("x"), new SquareFunction(),new Fields("y"))
.each(new Fields("x","y"),new PrintFunction(), new Fields())
;
```

Running this topology for a bit would produce output that looks like this (only the `x` and `y` fields are pulled into the `PrintFunction` for brevity):

```
[0.8185738974522312, 0.6700632255901359]
[0.04147059259035091, 0.0017198100497948677]
[0.5122091469924017, 0.2623582102626838]
[0.032330581404519276, 0.0010452664939542477]
[0.49881777491999457, 0.24881917257613437]
[0.9140296292506043, 0.8354501631479971]
[0.807521215873791, 0.6520905140862857]
[0.03596640476063595, 0.0012935822714058964]
[0.7539011202358764, 0.5683668990929094]
```

Repartitioning Operations

In Trident, groupings are called repartitioning operations, but remain essentially unchanged from their equivalent operations in standard topologies. These operations imply a network transfer of tuples just like they do in a traditional topology:

- The `shuffle` partitioning evenly distributes tuples across target partitions.
- The `broadcast` partitioning sends all tuples to all target partitions.
- The `partitionBy` partitioning is equivalent to the `fieldsGrouping`. It takes a list of fields found in each tuple and uses a hash of their values to determine a target partition.
- The `global` partition sends all tuples to the same partition. Trident also adds a `batchGlobal` variant that sends all tuples from the same batch to a single partition. Different batches may be sent to a different partition.
- There is also a `partition` function that takes an implementation of `CustomStreamGrouping` to implement custom partition methods.

In addition to these partition methods, Trident also implements a special type of partition operation called `groupBy`. This partition operation acts very much like the reducer phase in a Map-Reduce job and is probably the most commonly used partition in Trident.

The `groupBy` operation first applies a `partitionBy` partition according to the fields specified in the `groupBy` operation. Within each partition, values with identical hash values are then grouped together for further processing in a `GroupedStream`. Aggregators can then be applied directly to these groups.

Aggregation

Trident has two methods of aggregation on streams, `aggregate` and `persistent-Aggregate`. The `aggregate` method applied to a stream with a function that implements `ReducerAggregator` or `Aggregator` effectively performs a `global` partition on the data, causing all tuples to be sent to the same partition for aggregation on a per-batch basis. An aggregate method that is given a `CombinerAggregator` first computes an intermediate aggregate for each partition and then performs a `global` partition on the output of these intermediate aggregates.

The built-in aggregators `Count` and `Sum` are both `CombinerAggregators`. Other custom aggregation methods can be implemented by extending the `BaseAggregator` class. An example of this is shown in the next section when a partition local aggregator is implemented.

The other aggregation method, `persistentAggregate`, works like an aggregate except that it records its output into a `State` object. These `State` objects often work with the `Spout` in a

Trident topology to ensure features like transactional semantics.

The `persistentAggregate` method returns a `TridentState` object rather than a `Stream` object. This object has a method `newValuesStream` that emits a tuple every time a key's state changes. This is useful for retrieving the results of an aggregation that is using local memory as its backing store, such as the `MemoryMapState` class.

At the moment, the `MemoryMapState` class is the only State implementation built into Trident. It is primarily intended for testing purposes with other State objects implemented to persist to more durable stores, such as `memcached`.

Partition Local Aggregation

Although the normal aggregation operations imply a global partitioning event, the `partitionAggregate` method allows for partitioning:

```
public class KeyDoubleAggregator
  extends BaseAggregator<Map<Object, Double>> {
  private static final long serialVersionUID = 1L;
  public Map<Object, Double> init(Object batchId,
    TridentCollector collector) {
    return new HashMap<Object,Double>();
  }
  public void aggregate(Map<Object, Double> val, TridentTuple tuple,
    TridentCollector collector) {
    Object key = tuple.get(0);
    if(val.containsKey(key))
      val.put(key, val.get(key) + tuple.getDouble(1));
    else
      val.put(key, tuple.getDouble(1));
  }
  public void complete(Map<Object, Double> val,
    TridentCollector collector) {
    for(Entry<Object,Double> e : val.entrySet()) {
      collector.emit(new Values(e.getKey(),e.getValue()));
    }
  }
}
```

The Classic "Word Count" Example

The Word Count example is the "Hello World" of Big Data processing, and no discussion of real-time streaming processing would be complete without it.

This example uses a stream of Wikipedia edits coming from Kafka into Trident to implement a word counting example. This data source is actually provided by the Samza package discussed in the next part of this chapter, and it provides a handy source of data for testing. This is configured using the `TransactionalTridentKafkaSpout` class:

```
TridentTopology topology = new TridentTopology();
TridentKafkaConfig     config   = new TridentKafkaConfig(
  new ZkHosts("localhost"),
  "wikipedia-raw",
  "storm"
);
config.scheme = new SchemeAsMultiScheme(new StringScheme());
topology.newStream("kafka",
  new TransactionalTridentKafkaSpout(config)).shuffle()
```

This spout emits a string of JSON that must be parsed and split into words for further processing. For simplicity, this function implementation only looks at the title element of the raw output:

```
.each(new Fields("str"),new Function() {
  private static final long serialVersionUID = 1L;
  transient JSONParser parser;
  public void prepare(Map conf, TridentOperationContext context) {
    parser = new JSONParser();
  }
  public void cleanup() { }
  public void execute(TridentTuple tuple,
    TridentCollector collector) {
    if(tuple.size() == 0) return;
    try {
      Object obj = parser.parse(tuple.getString(0));
      if(obj instanceof JSONObject) {
        JSONObject json = (JSONObject)obj;
        String raw = (String)json.get("raw");
        raw = raw.substring(2,raw.indexOf("]]"));
        for(String word : raw.split("\\s+")) {
          collector.emit(new Values(word));
        }
      }
    } catch (ParseException e) {
      collector.reportError(e);
    }
  }
}, new Fields("word")).groupBy(new Fields("word"))
```

The output of this function is a tuple for each word. By grouping on word and then applying a `persistentAggregate`, the count of each word over time can be obtained. Every time a word appears in the output, its aggregate will be incremented by the number of times it appears in the title and then aggregated:

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

```
  )
  .newValuesStream()
```

Finally, these values are read into the `Debug` function, which works very much like the `LoggerBolt` implemented earlier in this chapter:

```
  .each(new Fields("word","count"), new Debug("written"));
```

After running the topology for a bit, there should be an output on the console that looks something like this:

```
DEBUG(written): [Category:Water, 1]
DEBUG(written): [Fleurimont, 1]
DEBUG(written): [for, 1]
DEBUG(written): [creation/Orin, 1]
DEBUG(written): [Wikipedia, 1]
DEBUG(written): [talk:Articles, 1]
DEBUG(written): [of, 5]
DEBUG(written): [players, 2]
DEBUG(written): [F.C., 1]
DEBUG(written): [List, 4]
DEBUG(written): [Arsenal, 1]
DEBUG(written): [Colby, 1]
DEBUG(written): [Jamie, 1]
DEBUG(written): [Special:Log/abusefilter, 1]
DEBUG(written): [Special:Log/abusefilter, 2]
```

Processing Data with Samza

A recent newcomer to the real-time processing space is another project from LinkedIn called Samza. Recently open-sourced and added to the Apache Incubator family of projects, Samza is a real-time data processing framework built on top of the Apache YARN infrastructure. The project itself is still very young, especially compared to Storm, which has been around for a few years, but it is already possible to do useful things with it.

This section describes the Samza architecture and how to get started using it. The section first gives an overview of Apache YARN, which is used as Samza's server infrastructure and takes the place of the Storm nimbus/supervisor servers (in fact Storm can also be run on Apache YARN using Yahoo!'s Storm-YARN project from `https://github.com/yahoo/storm-yarn`). Next is a tour of the Samza application itself. Like Storm's Trident system, Samza provides some primitives for building common types of streaming applications and maintaining state within a processing application.

Apache YARN

Rather than implement its own server management framework, Samza off-loads much of its systems infrastructure onto Apache YARN. YARN, which stands for Yet Another Resource Negotiator, is used to manage the deployment, fault tolerance, and security of a Samza processing pipeline.

Background

The YARN project was originally born out of the limitations of the Hadoop project. The Hadoop project was built around a `JobTracker` server that managed the distribution of tasks, mappers, and reducers, to other servers running the `TaskTracker` server. A client that wanted to submit a job would connect to the `JobTracker` and specify the input set, usually a distributed set of data blocks hosted on Hadoop's distributed file system, as well as any supporting code or data that needed to be distributed to each node. The `JobTracker` would then break this request into small tasks and schedule each of them on the tracker, as shown in Figure 5.4.

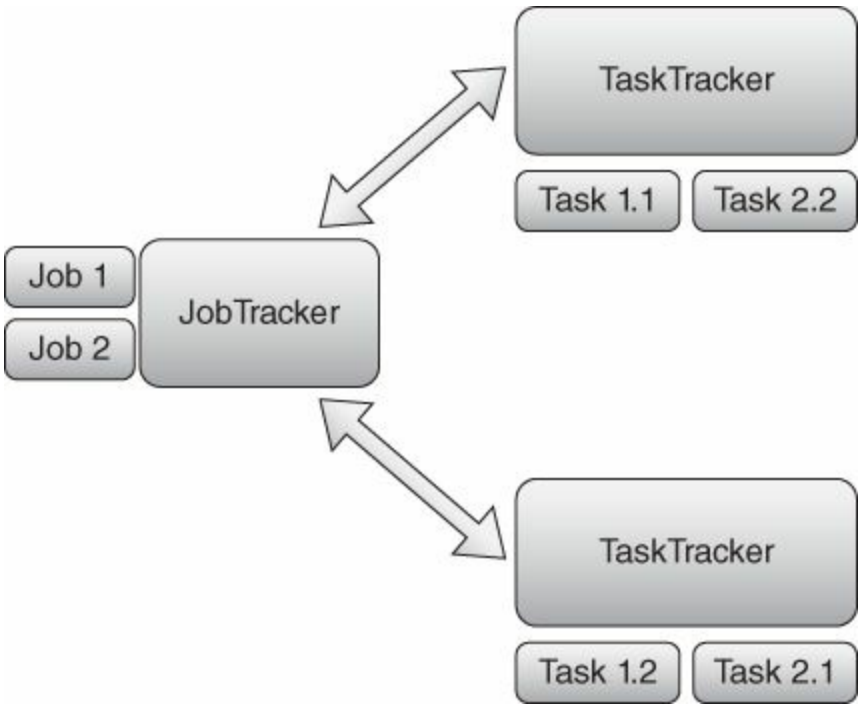


Figure 5.4

This works well on modestly sized clusters, but there's a practical limit in clusters with about 5,000 multicore servers. It also places practical limitations on the total number of tasks (either in a single job or spread across many jobs) because information about each of the tasks needs to be kept in memory on the `JobTracker` itself.

As physical hardware continues to scale and modern datacenters make it possible to host very large clusters, these scaling limitations began to take their toll on Hadoop's scalability. Additionally, new processing workloads, such as database-like applications and long-lived stream processing applications, were somewhat difficult to match to Hadoop's processing model, which assumes a long-lived but small set of reducer tasks coupled with a large number of short-lived mapper tasks.

To address the needs of both growing clusters and changing workloads, YARN was developed as Hadoop 2.

Architecture

In the abstract, the YARN architecture is not so different from the original Hadoop infrastructure. Rather than a `JobTracker` and a `TaskTracker`, the top-level servers are now the `ResourceManager` and the `NodeManager`, respectively. The important difference is that these servers now manage applications, not individual tasks.

An application in YARN consists of an `ApplicationMaster` and a number of containers that are hosted on each node. The `ApplicationMaster`, as might be guessed from the name, is in charge of coordinating a job and managing its assigned containers, which host the individual tasks. The relationship between these components is shown in Figure 5.5.

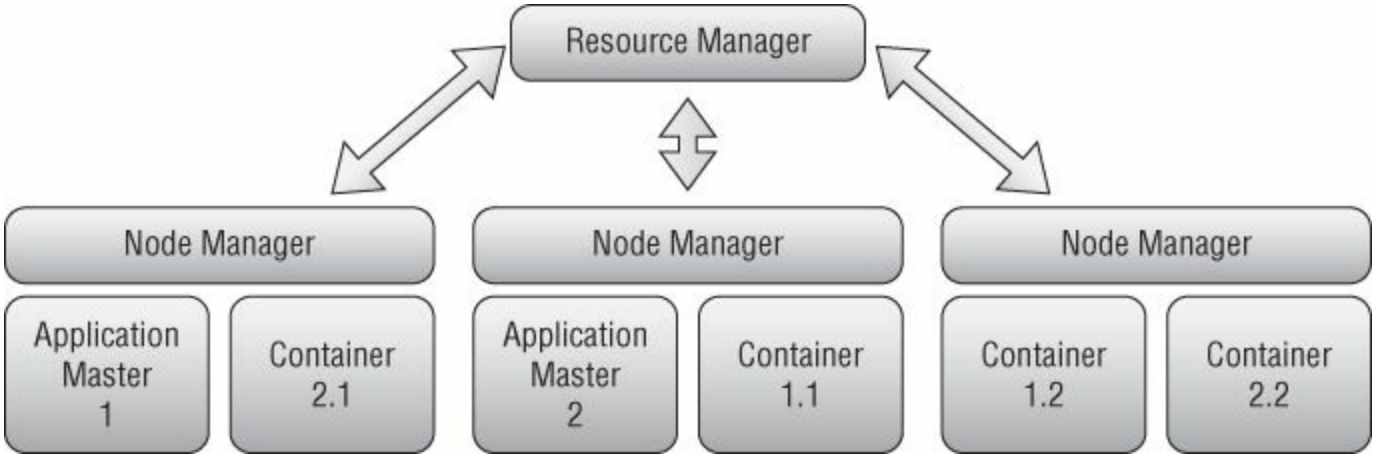


Figure 5.5

In Hadoop 2's Map-Reduce implementation, the `ApplicationMaster` serves as the `JobTracker`. The exception is that each `ApplicationMaster` only manages the task for a single job instead of managing a large number of jobs. This allows each `ApplicationMaster` to operate independently in Map-Reduce settings. It also allows for more sophisticated resource management and security models because jobs are now essentially completely independent.

Relationship to Samza

Samza is implemented as an application on top of YARN. The Samza application has the required `ApplicationManager` that is used to manage Samza `TaskRunners` hosted within YARN

`Containers`. The `TaskRunners` execute `StreamTasks`, which are the Samza equivalent of a Storm `Bolt`.

All of Samza's communication is hosted through Kafka brokers. Like HDFS DataNodes in a Hadoop Map-Reduce application, these brokers are usually co-located on the same machines hosting the Samza Containers. Samza then uses Kafka's topics and natural partitioning to implement many of the grouping features found in stream processing applications.

Getting Started with YARN and Samza

Although Hadoop 2 has been available for some time, it is still not particularly common in production environments, though that is changing rapidly. Most importantly for many users, Hadoop 2 is now supported by Amazon's Elastic MapReduce product as a general release, making it easy to spin up a cluster.

Apache YARN is also now supported by at least two of the major Hadoop distributions, with more being added. Using their respective cluster management tools to set up a YARN cluster is fairly painless. The only downside is that packaged distributions tend to have a somewhat arbitrary set of patches and versions that may lag the most recently released version of the Apache project.

Additionally, it is possible to spin up a cluster using the Apache packages either on a single node for experimentation or in a distributed fashion.

Single Node Samza

The easiest way to get started with Samza on a single node is to use the single-node YARN installation packaged with the Hello Samza project. This project includes Samza, ZooKeeper, Kafka, and YARN in a convenient package for development.

To get started, first check out the hello-samza Git repository from Github. This repository includes a script that will build and install a complete Samza installation:

```
$ git clone https://github.com/apache/incubator-samza-hello-samza.git
$ cd incubator-samza-hello-samza/
$ ./bin/grid bootstrap
```

JAVA_HOME Not Set

On some systems, such as Mac OS X, when `grid` is run, it returns a `JAVA_HOME not set` error:

```
$ ./bin/grid bootstrap
JAVA_HOME not set. Exiting.
```

On OS X, the following sets the `JAVA_HOME` environment variable to the appropriate value:

```
$ export JAVA_HOME=$(/usr/libexec/java_home)
```

The first time it is run, the grid script builds and installs Samza into the local Maven repository. This eliminates the need to check out and build the `git` repository separately as indicated in the tutorial on the Samza website. It also downloads appropriate versions of ZooKeeper and Kafka for use with Samza and starts them. The build process takes 3 to 5 minutes, and installing ZooKeeper and Kafka takes a few more minutes depending on network speeds. The `grid` application also downloads and installs a single node version of YARN. This download is around 100MB in size, so it is usually best to have a decent connection when first installing the Hello Samza project.

If all has gone well, pointing a web browser at http://localhost:8088 should show something like Figure 5.6.

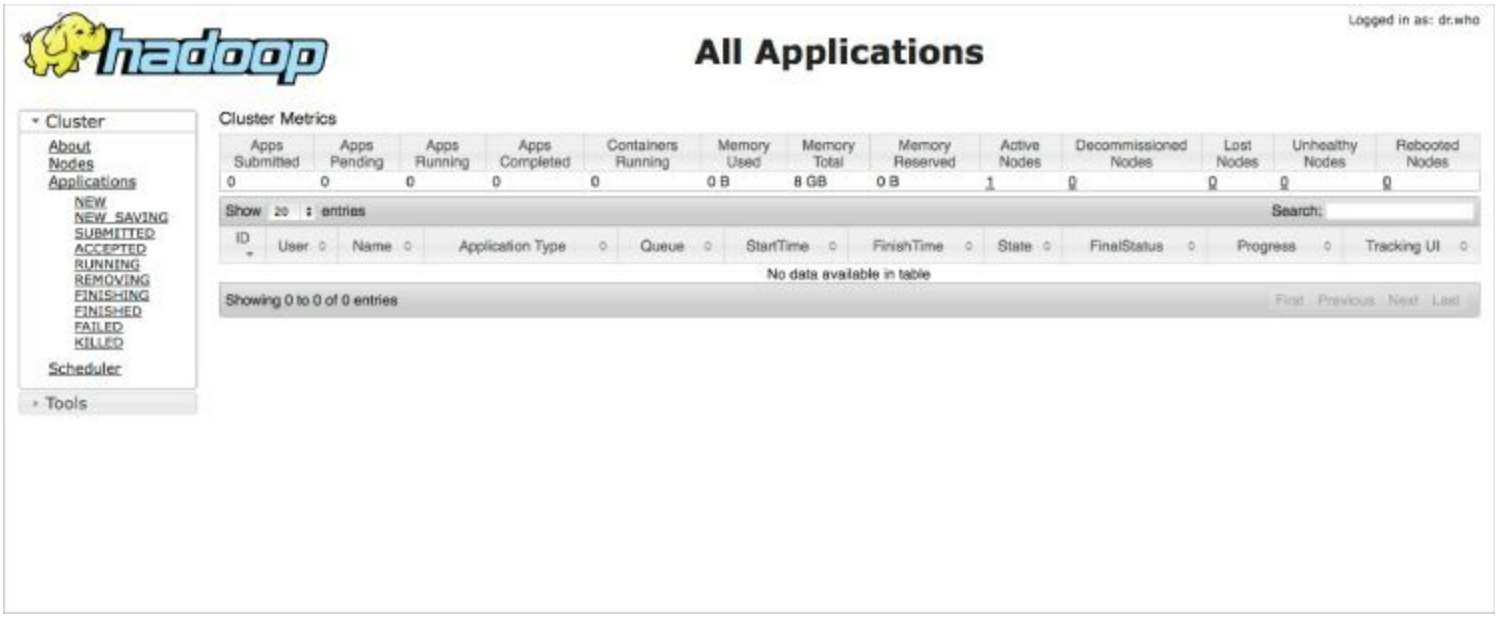


Figure 5.6

To shut it down, the grid script provides a command to stop all processes in the correct order:

```
$ ./bin/grid stop all
```

To start the grid again, simply use the `start` command:

```
$ ./bin/grid start all
```

Any processes that are already running as part of a different server pool, such as ZooKeeper, can be omitted. In that case, rather than using the `all` command simply specify the names of the individual servers to start.

The Hello Samza project also includes some example code that can be run. This chapter uses the first part of the example code for some of its examples. This Samza job reads the Wikipedia edit stream from the Wikipedia IRC servers and posts them as JSON to the `wikipedia-raw` topic on the local

Kafka cluster. To start this job on the grid, execute the following code:

```
$ ./deploy/samza/bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory
> --config-path=\
> file://$PWD/deploy/samza/config/wikipedia-feed.properties
```

After this job has been started, check that the edits are being posted to Kafka using the console consumer that ships with Kafka:

```
$ ./deploy/kafka/bin/kafka-console-consumer.sh \
> --zookeeper localhost:2181 --topic wikipedia-raw
```

Multinode Samza

Getting Samza going in a multinode environment is very much like setting up a Hadoop 2 environment, except that HDFS is not required unless other applications will be using it.

To begin, make sure a ZooKeeper cluster is available, referring to Chapter 3 if necessary for installation and configuration instructions. ZooKeeper is required for both Samza and the Kafka brokers.

Next, install and configure Kafka brokers on the machines destined for the YARN grid except for the machine to be used as the `ResourceManager`. Samza makes heavy use of Kafka for communication, so its brokers are usually co-located with the NodeManagers that make up the Samza YARN grid. For help with installing and configuring Kafka, refer to Chapter 4.

Now that ZooKeeper and Kafka have been installed, set up the YARN cluster. In this section it is assumed that you are constructing this YARN cluster from the Apache build rather than one of the commercial distributions. It is also assumed that the appropriate package manager for the operating system being used does not have appropriate packages available for Hadoop 2.2.0 (many still have older Hadoop 1.0.3 packages).

First, download and unpack the Apache binary distribution, which is at version 2.2.0 at the time of writing, onto each of the machines:

```
$ wget http://mirrors.sonic.net/
    apache/hadoop/common/hadoop-2.2.0/hadoop-2.2.0.tar.gz
$ cd /usr/local
$ sudo tar xvfz ~/hadoop-2.2.0.tar.gz
$ sudo ln -s hadoop-2.2.0/ hadoop
```

Apache YARN relies on a number of environment variables to tell it where to find its various packages. There are a number of places this can be set, but to have it active for all users it should be set in either `/etc/profile` or `/etc/profile.d/hadoop.sh`. On most systems, `/etc/profile` automatically includes all scripts in `/etc/profile.d`. If YARN was unpacked into the `/usr/local` directory, the `/etc/profile.d/hadoop.sh` file would look like this:

```
export YARN_HOME=/usr/local/hadoop
export HADOOP_MAPRED_HOME=$YARN_HOME
export HADOOP_COMMON_HOME=$YARN_HOME
export HADOOP_HDFS_HOME=$YARN_HOME
export PATH=${PATH}:${YARN_HOME}/bin:${YARN_HOME}/sbin
```

To see these changes reflected in the environment, log out and log back in again or open a new terminal. After doing that, checking the `YARN_HOME` environment variable should give

/usr/local/hadoop, and a check of the Hadoop version should return 2.2.0:

```
$ echo $YARN_HOME
/usr/local/hadoop
$ hadoop version
Hadoop 2.2.0
Subversion https://svn.apache.org/repos/asf/hadoop/common -r 1529768
Compiled by hortonmu on 2013-10-07T06:28Z
Compiled with protoc 2.5.0
From source with checksum 79e53ce7994d1628b240f09af91e1af4
This command was run using
/usr/local/hadoop-2.2.0/share/hadoop/common/hadoop-common-2.2.0.jar
```

Because this grid is being used for Samza, it is not necessary to configure HDFS. If the grid is also going to be used for Map-Reduce workloads, refer to the appropriate YARN documentation for the configuration of the NameNode and DataNode servers. The only thing that needs to be configured for Samza is the yarn-site.xml file found in /usr/local/hadoop/etc/hadoop (replacing resource-manager.mydomain.net with an appropriate hostname):

```
<?xml version="1.0"?>
<configuration>
<property>
<name>yarn.resourcemanager.hostname</name>
<value>resource-manager.mydomain.net</value>
</property>
</configuration>
```

It should now be possible to start the ResouceManager and NodeManager on the Kafka grid. To start the ResourceManager, log in to the machine and use yarn-daemon.sh to start the server:

```
$ yarn-daemon.sh –config $YARN_HOME/etc/hadoop start resourcemanager
```

Then, on each of the nodes in the Samza grid, start the NodeManager in the same way:

```
$ yarn-daemon.sh –config $YARN_HOME/etc/hadoop start nodemanager
```

The ResourceManager starts a web server on port 8088 by default; it can be checked to ensure each of the nodes has reported to the resource manager. The most common problem at this point is an incorrect firewall setting.

Integrating Samza into the Data Flow

Integrating Samza into an existing Kafka environment is straightforward, as Samza uses Kafka for all communication. If there is an existing set of brokers already handling production load, simply use MirrorMaker as described in Chapter 4 to mirror the desired topics into the Samza Kafka grid. From there, Samza has easy access to the incoming topics.

Alternatively, install the Samza grid on the same machines as the Kafka brokers used to collect data. This has some slight operational disadvantages, as it is always possible a processing job could lock up a machine and bring it down. However, it is likely more operationally efficient because Kafka brokers usually have spare processing cycles.

Samza Jobs

With a configured cluster and data being imported to the Kafka portion of the grid, it is time to implement a Samza Job. A Job in Samza parlance is roughly equivalent to a Bolt in Storm.

However, rather than being assembled into a `Topology`, `Jobs` in Samza are all independent entities, and any composition is simply a matter of reading or writing to a particular Kafka topic.

On the one hand, this potentially allows for significantly more efficient uses of resources and easier management of highly interconnected flows. For example, a single Samza `Job` can be responsible for taking an input stream and producing a "valid" input stream that can be used by any number of downstream `Job` implementations. New processes can also be easily added to take advantage of these streams, whereas before they might have had to reprocess the entire raw stream of data again with only a small change.

The downside of this approach is that the structure of the topology of jobs is now purely abstract. In the Storm model, a topology is a distinct thing, and all of the processing steps are controlled and monitored from a central location. In the Samza model, the topology is implied in the arrangement of topics, but never explicitly stated. This can lead to problems with managing changes in upstream `Job` implementations or the inadvertent introduction of cycles into the topology's structure.

Preparing a Job Application

A Samza `Job` is made up of two pieces. The first is the actual code, which is an implementation of the `StreamTask` interface. The second piece is the configuration of the Task, including the name of the input streams and the configuration of logging, monitoring, and other ancillary facilities. Any number of `Job` implementations can be hosted and packaged together for ease of deployment.

If not using the `grid` implementation previously described, it will be necessary to install the Samza Maven packages. These packages are not yet available on Maven Central or another Maven repository so they need to be built and installed locally. Check out the Git repository for Samza and then run the Gradle script using the provided `gradlew` driver:

```
$ git clone http://git-wip-us.apache.org/repos/asf/incubator-samza.git
$ cd incubator-samza
$ ./gradlew -PscalaVersion=2.8.1 clean publishToMavenLocal
```

When this is complete, the project containing the Job implementation should be updated to include the `samza-api` dependency in its `pom.xml` file:

```
<dependency>
<groupId>org.apache.samza</groupId>
<artifactId>samza-api</artifactId>
<version>0.7.0</version>
</dependency>
```

Configuring a Job

Samza's `Job` configurations are accomplished through the use of a Properties file that is passed to the Samza framework when submitting the `Job` to the YARN framework. The Properties file starts with a `Job` factory class specification and a `Job` name, along with a distribution package:

```
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=wordcount-split
yarn.package.path=
  file://${basedir}/target/
  ${project.artifactId}-${pom.version}-dist.tar.gz
```

The factory class will generally always be `YarnJobFactory`, and the name is currently set to `wordcount-split`, which is implemented in the next section. The `yarn.package.path` is

filled in by the build process and specifies the name of an archive that YARN transfers to each of the nodes. This contains any support JAR files that might be needed along with the code that implements the `Job`.

Next, the task is defined. This consists, minimally, of the `task.class` and `task .inputs` properties:

```
task.class=wiley.streaming.samza.WordSplitTask
task.inputs=kafka.wikipedia-raw
```

It can also include properties for a task's checkpoint feature, which uses Kafka to record the task's state and allows Samza to recover in the event of a failure:

```
task.checkpoint.factory=
org.apache.samza.checkpoint.kafka.KafkaCheckpointManagerFactory
task.checkpoint.system=kafka
task.checkpoint.replication.factor=1
```

Next up is Samza's comprehensive metric reporting system. It is entirely optional and, at the moment, has both a `snapshot` and `jmx` flavor available. This example uses both flavors at the same time:

```
metrics.reporters=snapshot,jmx
metrics.reporter.snapshot.class=
org.apache.samza.metrics.reporter.MetricsSnapshotReporterFactory
metrics.reporter.snapshot.stream=kafka.metrics
metrics.reporter.jmx.class=
org.apache.samza.metrics.reporter.JmxReporterFactory
```

The Serialization section of the `Job` configuration allows for the definition of the various `Serializers` and `Deserializers` (these are often combined into "SerDe" in processing jargon) used in the `Job`. This example uses the built-in JSON SerDe as well as the Metrics SerDe used by Samza for its metric snapshots:

```
serializers.registry.json.class=
org.apache.samza.serializers.JsonSerdeFactory
serializers.registry.metrics.class=
org.apache.samza.serializers.MetricsSnapshotSerdeFactory
```

Finally, the Input and Output systems are defined. These implement streams in the Samza environment and are pluggable systems, though only the Kafka stream is shipped with Samza at the moment. This configuration defines the Kafka system used by `task.inputs` as well as the output defined within the task itself. It also defines the metrics stream system used by the earlier metrics definition:

```
systems.kafka.samza.factory=
org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.consumer.auto.offset.reset=largest
systems.kafka.producer.metadata.broker.list=localhost:9092
systems.kafka.producer.producer.type=sync
# Normally, we'd set this much higher, but we want things to
# look snappy in the demo.
systems.kafka.producer.batch.num.messages=1
systems.kafka.streams.metrics.samza.msg.serde=metrics
```

The `systems.kafka.consumer.zookeeper.connect` and `kafka.producer.metadata .broker.list` values should be updated to reflect the deployment environment. If using the Hello Samza `grid`, these values are appropriate.

Task Communication

The `systems` and `serializers Job` properties in the previous section define the input and, to some extent, output mechanisms of the task to be implemented. These are made available in Samza via the `SystemConsumer` and `SystemProducer` interfaces, which define a task's input and output, respectively.

The `SystemConsumer` is defined by the `task.inputs` property in the `Job` configuration and is responsible for polling the input partitions. It produces `IncomingMessageEnvelope` objects that are then passed to the task. This object consists of three parts:

- The `key`, accessible via the `getKey` method. This is the key used to partition the data within the Kafka stream and corresponds directly to the key portion of a Kafka message.
- The `message`, accessible via the `getMessage` method. This typically contains the payload of the stream.
- The `StreamSystemPartition` object, accessible via the `getSystemStreamPartition` method. This contains metadata about the message, such as its originating partition and offset information. Most tasks do not need to access this information directly.

The `SystemProducer` is usually accessed through a `SystemStream` defined as a static final member of the task's class. For example, to define a Kafka output stream on the topic "`output`" add the following line to the class definition:

```
private static final SystemStream OUTPUT =
  new SystemStream("kafka","output");
```

This `SystemStream` object is used to initialize an `OutgoingMessageEnvelope` along with the message payload and an optional key payload. This object is then sent to the `MessageCollector`, which manages placing the outgoing message on the specified stream.

Implementing a Stream Task

The task itself is defined by the StreamTask interface, which is implemented by all task classes. This interface has a single process method, which takes `IncomingMessageEnvelope`, `MessageCollector`, and `TaskCoordinator` as arguments. The `IncomingMessageEnvelope` is a message from the `task.inputs` streams as described in the previous section, whereas the `MessageCollector` is used to emit events to output streams, which are defined within the class.

For example, this task handles the task of splitting the input of the `wikipedia-raw` topic into words, the same way that words were split up in the Trident example earlier in this chapter. In this case the JSON SerDe is being used to parse the incoming data (which is in JSON) and sent to the `wikipedia-words` topic.

```
public class WordSplitTask implements StreamTask {
  private static final SystemStream OUTPUT_STREAM
    = new SystemStream("kafka","wikipedia-words");
  public void process(IncomingMessageEnvelope envelope,
   MessageCollector collector,
   TaskCoordinator coordinator) throws Exception {
    try {
      @SuppressWarnings("unchecked")
```

```
      Map<String,Object> json =
        (Map<String,Object>)envelope.getMessage();
      String raw = (String)json.get("raw");
      raw = raw.substring(2,raw.indexOf("]]"));
      for(String word : raw.split("\\s+")) {
        HashMap<String,Object> val = new HashMap<String,Object>();
        val.put("word", word);
        collector.send(new OutgoingMessageEnvelope(OUTPUT_STREAM,val));
      }
    } catch(Exception e) {
      System.err.println(e);
    }
  }
}
```

Initializing Tasks, Windows

In addition to the `StreamTask` interface, a task may implement the `InitableTask` interface. This adds an `init` method that is called when the task object is created. This is used to establish connections to resources at run time that cannot be statically initialized.

Tasks that should periodically emit values, such as counting operations, can implement the `WindowableTask` interface. This interface adds a `window` method that is periodically called by the Samza framework.

For example, a task that consumes the output of `wikipedia-words` from the previous section and maintains counts of the individual words might use all three interfaces. In this case, the `init` interface is not strictly necessary, but it is included for completeness:

```
public class WordCountTask implements StreamTask,
  InitableTask, WindowableTask {
  private static final SystemStream OUTPUT_STREAM =
    new SystemStream("kafka","wikipedia-counts");
  HashMap<String,Integer> counts = new HashMap<String,Integer>();
  HashSet<String> changed = new HashSet<String>();
  public void window(MessageCollector arg0, TaskCoordinator arg1)
    throws Exception {
    for(String word : changed) {
      HashMap<String,Object> val = new HashMap<String,Object>();
      val.put("word", word);
      val.put("count", counts.get(word));
      arg0.send(new OutgoingMessageEnvelope(OUTPUT_STREAM,val));
    }
    changed.clear();
  }
  public void init(Config arg0, TaskContext arg1) throws Exception {
    counts.clear();
    changed.clear();
  }
  public void process(IncomingMessageEnvelope arg0,
    MessageCollector arg1,
    TaskCoordinator arg2) throws Exception {
    @SuppressWarnings("unchecked")
    Map<String,Object> json = (Map<String,Object>)arg0.getMessage();
    String word = (String) json.get("word");
    counts.put(word,
      (counts.containsKey(word) ? counts.get(word) : 0) + 1);
    changed.add(word);
  }
```

```
}
```

The properties file that goes along with this Task specifies a 10-second windowing cycle:

```
# Job
job.factory.class=org.apache.samza.job.yarn.YarnJobFactory
job.name=word-count
# YARN
yarn.package.path=
 file://${basedir}/target/
 ${project.artifactId}-${pom.version}-dist.tar.gz
# Task
task.class=wiley.streaming.samza.WordCountTask
task.inputs=kafka.wikipedia-words
task.window.ms=10000
# Serializers
serializers.registry.json.class=
org.apache.samza.serializers.JsonSerdeFactory
# Systems
systems.kafka.samza.factory=
org.apache.samza.system.kafka.KafkaSystemFactory
systems.kafka.samza.msg.serde=json
systems.kafka.consumer.zookeeper.connect=localhost:2181/
systems.kafka.consumer.auto.offset.reset=largest
systems.kafka.producer.metadata.broker.list=localhost:9092
systems.kafka.producer.producer.type=sync
systems.kafka.producer.batch.num.messages=1
```

Packaging a Job for YARN

To package `Jobs` for YARN, you must create a distribution archive. This archive contains not only the JAR file for the `Job` implementation, but all of the configuration files and dependencies for the project. It also includes shell scripts for starting `Jobs` using YARN. This archive will be distributed by YARN to each of the nodes that will run the `Job`.

The easiest way to construct this archive is to use the Maven assembly plug-in. This plug-in is added to the `plugins` section of the `pom.xml` file:

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <version>2.3</version>
  <configuration>
  <descriptors>
    <descriptor>src/main/assembly/src.xml</descriptor>
   </descriptors>
   </configuration>
   <executions>
   <execution>
     <id>make-assembly</id>
     <phase>package</phase>
     <goals>
       <goal>single</goal>
      </goals>
    </execution>
    </executions>
  </plugin>
```

This plug-in references an assembly XML file that is usually located in `src/main/assembly.src`. It defines the files that should be included as well as the dependencies to include. It begins by defining the type of assembly to build—in this case a `tar.gz`

file:

```
<assembly
xmlns=
"http://maven.apache.org/plugins/maven-assembly-plugin/assembly
 /1.1.2"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-
plugin/assembly/1.1.2
http://maven.apache.org/xsd/assembly-1.1.2.xsd"
>
  <id>dist</id>
  <formats>
    <format>tar.gz</format>
  </formats>
  <includeBaseDirectory>false</includeBaseDirectory>
```

Next, include various nondependency files. This includes, most importantly, the configuration files for the `Job`s included in the artifact:

```
  <fileSets>
    <fileSet>
      <directory>${basedir}/..</directory>
      <includes>
        <include>README*</include>
        <include>LICENSE*</include>
        <include>NOTICE*</include>
      </includes>
    </fileSet>
  </fileSets>
  <files>
    <file>
      <source>${basedir}/src/main/config/word-split.properties</source>
      <outputDirectory>config</outputDirectory>
      <filtered>true</filtered>
    </file>
    <file>
      <source>${basedir}/src/main/config/word-count.properties</source>
      <outputDirectory>config</outputDirectory>
      <filtered>true</filtered>
    </file>
  </files>
```

This section includes the shell scripts from the `samza-shell` artifact as well as specifies the dependencies that should be included in the distribution `tar.gz` file:

```
  <dependencySets>
    <dependencySet>
      <outputDirectory>bin</outputDirectory>
      <includes>
        <include>org.apache.samza:samza-shell:tgz:dist:*</include>
      </includes>
      <fileMode>0744</fileMode>
      <unpack>true</unpack>
    </dependencySet>
    <dependencySet>
      <outputDirectory>lib</outputDirectory>
      <includes>
        <include>org.apache.samza:samza-core_2.8.1</include>
        <include>org.apache.samza:samza-kafka_2.8.1</include>
        <include>org.apache.samza:samza-serializers_2.8.1</include>
```

```
        <include>org.apache.samza:samza-yarn_2.8.1</include>
        <include>org.slf4j:slf4j-log4j12</include>
<!--         <include>wiley:streaming-chapter-5</include> -->
        <include>org.apache.kafka:kafka_2.8.1</include>
      </includes>
      <useTransitiveFiltering>true</useTransitiveFiltering>
    </dependencySet>
  </dependencySets>
</assembly>
```

Executing Samza Jobs

After building the distribution archive, copy it to an appropriate location. For a multinode installation, this would be a machine with the appropriate YARN client installed.

Unpack the archive into an appropriate directory, such as `deploy`:

```
$ mkdir deploy
$ cd deploy
$ tar xvfz ../target/streaming-chapter-5-1-dist.tar.gz
$ cd ..
```

Then, use the included `run-job.sh` script to submit the `Jobs` to YARN. For example, to submit the `WordSplitTask` and the `WordCountTask` examples for this chapter, execute the following commands:

```
$ ./deploy/bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory
> --config-path=\
> file://$PWD/deploy/config/word-split.properties
$  ./deploy/bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory
> --config-path=\
> file://$PWD/deploy/config/word-count.properties
```

Then check the `wikipedia-counts` topic for data to ensure the jobs are running properly:

```
$ ./deploy/kafka/bin/kafka-console-consumer.sh \
> --zookeeper localhost:2181 --topic wikipedia-counts
```

Conclusion

This chapter introduced two frameworks for stream processing: the more mature Storm framework and the newer Samza framework. Both frameworks are fairly new and have a long way to go before they would be considered to be complete frameworks. However, both can and are used to build useful streaming applications today. Both can be adapted to fill nearly any need and will become more complicated as they mature.

After the processing is done, it usually needs to be saved somewhere to be useful. The next chapter covers storage options for these streaming frameworks so that the data can be used by front-end applications for visualization or other tasks.

Chapter 6
Storing Streaming Data

One of the primary reasons for building a streaming data system is to allow decoupled communication and access between different aspects of the system. A key system is the storage and backup mechanism for both raw data as well as data that has been processed by one or more of the processing environments covered in the previous chapter.

Processing the data is one thing, but for it to be delivered to the end user it needs to be stored somewhere. That storage location could be the processing system, using something like Storm's Distributed Remote Procedure Calls (DRPC) and in-bolt memory storage. However, in a production environment this simply isn't practical. First, the data usually need to persist for a time, which means the memory requirements become prohibitive. Second, it means that maintenance for the processing system necessitates an outage of any external interfaces, despite the fact that the two have nothing to do with each other. Finally, it is usually desirable to persist results to tertiary storage (disks or "cloud" storage devices) so that the data may be more easily analyzed for long-term trends.

This chapter considers how to store data after it has been processed. There are a number of storage options available for processing systems that need to deliver their data to some sort of front-end interface, typically either an application programming interface (API) or a user interface (UI). Although there are dozens of potential options, this chapter surveys some of the more common choices. Systems with different philosophies and constraints are intentionally chosen to highlight these differences. This allows for more informed decision-making when considering storage of specific applications.

For longer term storage and analysis, a batch system often makes more sense than a streaming data system. Largely, this is because a streaming system chooses to trade off relatively expensive storage options, such as main memory, for lower random access latency. A batch system takes the opposite side of this trade off, choosing high capacity storage with high random access latency, such as traditional spinning platters. Fortunately, whereas a batch system's random access performance is usually not sufficient for streaming systems, its linear read performance is often very good making them an excellent choice for offline analysis.

Hadoop has now become the gold standard for these batch environments. This chapter describes setting up and integrating Hadoop into the streaming environment.

This chapter also discusses the basics of using Hadoop as a gateway to existing business intelligence infrastructures. The reason for this is that, although this book is focused on real-time streaming data analysis, no modern analytics system exists in a vacuum. A system *must* integrate with other pieces of an organization's environment if it hopes to gain widespread adoption.

Consistent Hashing

Several of the data stores described in the remainder of this chapter support the distribution of data across several servers. This allows them to scale more easily as the size of the data grows. It also generally improves performance for both updates and queries. A popular technique to implement the distribution of data is known as consistent hashing.

Most data stores have some notion of a "key" element. In a relational database it is called a primary key and in key-value stores it is simply called the keys. The most important element of the key is that there can only be a single entry for it in the data store (relational databases that do not have a specified primary key generally create an arbitrary primary key that is hidden from the user).

Each of these primary keys corresponds to a numerical value that is then assigned to a specific server. The mechanism used to perform this assignment varies, but the effect is that a specific primary key is always assigned to the same server for storage and querying.

This helps to distribute the load across a variety of servers, but does so at the expense of reliability. If any of the servers crashes, then the data stored on that server becomes unavailable and, as the number of servers increases, the probability that a server will have crashed at a given time increases.

To improve the reliability of the system, the data is instead consistently hashed. In consistent hashing the servers are first placed into a specific stable ordering, called a ring. When a primary key is modified the initial server selection proceeds as before, but the data is also modified on the next `k` servers in the ring (the reason for the ring is so that the servers to modify "wraps around" to the beginning of the list if the primary server is within `k` servers of the end of the list).

If one of the servers becomes unavailable, clients can simply update and query the remaining server. If the server is returned to the pool, it can then copy its data from one of the other servers before considering itself up-to-date.

The data store undergoes a similar process when adding a new server or permanently removing a server from the ring. When adding a server, the new server will copy data appropriate to its location in the ring. When removing a server, a pre-existing server will now need to hold the data from that ring, which should be copied from the existing servers. Both of these operations can happen in the background; since there are other servers in the ring that can serve up queries for data, the server being updated is missing. Updates from a client can simply be made to the new server without checking.

“NoSQL” Storage Systems

So-called “NoSQL” storage systems have grown in popularity over the last few years, largely in step with the rise of Big Data applications. This section covers a broad range of different approaches that are largely characterized by high-performance reads and writes, usually at the expense of the usual requirements of a transactional database, the complexity of the queries, or (most often) both.

For many Big Data and, in particular, real-time streaming applications, these are considered to be acceptable tradeoffs. Especially when using the Lambda Architecture discussed later in this chapter, the requirements of a data store for real-time analysis can be quite relaxed.

Although there are dozens of different storage options available for streaming applications, this section discusses just three. They have been chosen to demonstrate the range of possible choices and, in a production system, it is likely that more than one of these systems will be in use as they have different strengths and weaknesses. These systems are also generally a complement rather than a replacement for the relational database.

The first of these stores is the simplest: a key-value store. There are many different key-value stores, but Redis is one of the more popular options. It has performance similar to that of Memcached, but provides native support for higher order data structures that are very useful for many streaming analysis applications. Although not natively distributed, it focuses on very high-performance single-machine applications.

The next store is MongoDB, which is a document store. It is schema-less and has proven popular for applications that maintain rich profiles or data that can be naturally ordered into documents. It supports master-slave replication as well as sharding (partitioning). It can support very high write performance, also near Memcached levels, at the expense of safety. In modern versions of the database, this tradeoff can be tuned at the client level.

Finally, Cassandra is a decentralized database system that takes features from both key-value stores and tabular databases using elements from both Amazon's DynamoDB as well as Google's BigTable. Early versions suffered from a difficult-to-use query language, which essentially exposed internal data structures. However, more recent versions have introduced a much more usable query language that strongly resembles SQL, making it a much more viable option.

The rest of this section discusses the usage of each option in turn. The question of which one is best is left largely unanswered. This is because there is no “best” technology for all applications. Does this mean that multiple database technologies might be used in a single environment? Absolutely. This is why the last few chapters have been concerned with building a flexible data-flow system: Data needs to flow and be transformed between systems.

Redis

Redis is a simultaneously simple and complicated key-value store. It is simple in that it makes little attempt to solve two problems often solved by key-value stores: working sets larger than main memory and distributed storage. A Redis server can only serve data that fits in its main memory. Although it has some replication facilities, it does not support features such as eventual consistency, and even though Redis has been in the works for some time, even sharding and consistent hashing are provided by outside services, if they're provided at all. In many ways, Redis is more closely related to caching systems like Memcached than it is a database.

Unlike caching systems, Redis provides server-side support for high-level data structures with atomic update capability. The basic structures are lists, sets, hash tables, and sorted sets in addition to the basic key-value functionality. It also includes the capability to expire keys and a publish-subscription mechanism that can be used as a messaging bus between, for instance, the real-time streaming processing system and a front end. Later chapters provide more detail, but when keys are updated a front-end server can be notified of the event so that the user interface may be updated.

Getting Set Up

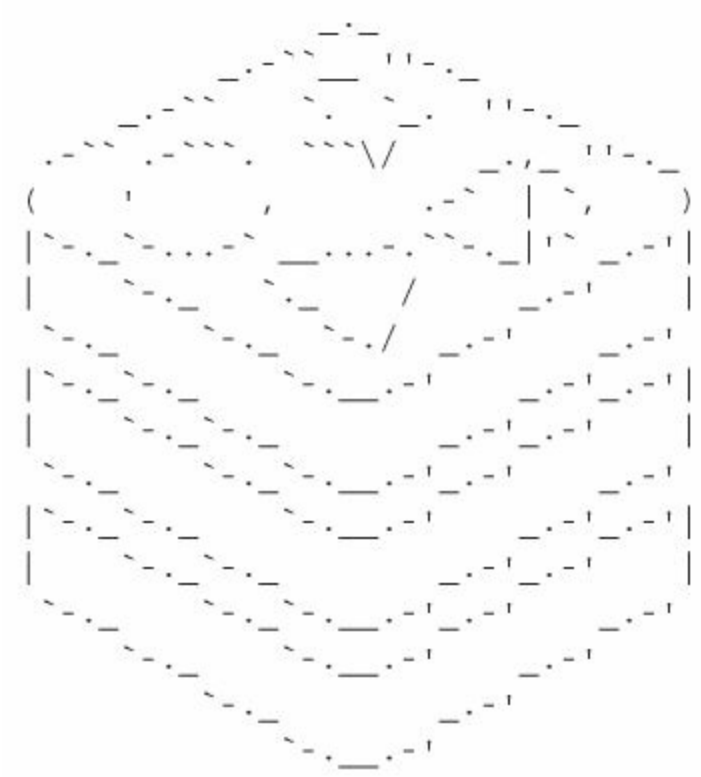
On most systems, Redis is available through the standard package manager. However, if Redis is out-of-date or otherwise unavailable through a package manager, it is easy to build it on Unix-like systems. After downloading the appropriate archive, the usual `make && make install` will build a suitable version of Redis. By default, this build is 64 bit, but it is possible to build a 32-bit version. Building a 32-bit version will save some memory, but limits Redis' total footprint to 4GB.

NOTE

For users of Amazon's EC2 service, Redis is now included as part of the Elasticache. The version usually lags the most recent release, but it makes it really easy to get started using Redis in either a development or a production environment. Just select redis from the drop-down (it shows memcache by default).

Redis ships with a fairly good set of default configuration parameters, but there are some options that should be considered when using Redis. The configuration entries are usually held in the `redis.conf` file passed to the server when it starts.

```
$redis-server
```




```
[23908] 09 Nov 14:16:16.249 # Server started, Redis version 2.6.16
[23908] 09 Nov 14:16:16.250 * The server is now ready to accept
  connections on port 6379
```

Working with Redis

The easiest way to start exploring Redis is using the `redis-cli` tool to connect to a redis server. Having started a local copy of the redis server, starting the redis client should look something like this:

```
$ redis-cli
redis 127.0.0.1:6379> info
# Server
redis_version:2.6.16
redis_git_sha1:00000000
[...]
# Memory
used_memory:1082448
used_memory_human:1.03M
used_memory_rss:319488
```

```
used_memory_peak:1133520
used_memory_peak_human:1.08M
used_memory_lua:31744
mem_fragmentation_ratio:0.30
mem_allocator:libc
[...]
# Replication
role:master
connected_slaves:0
# CPU
used_cpu_sys:69.16
used_cpu_user:35.95
used_cpu_sys_children:0.00
used_cpu_user_children:0.00
# Keyspace
> db0:keys=1,expires=0,avg_ttl=0
```

The preceding output shows the results of the `info` command. Some of the output has been omitted to save space, but there are some items of interest that become useful when administering and monitoring a Redis instance.

Because Redis is a key-value store, the most basic commands are for setting keys. Keys in the Redis world are generally considered to be string values, even when they are being used to represent integer or floating-point values. To set or get a key, use the `SET <key> <value>` and `GET <key>` commands, respectively. If there are several keys to be set or retrieved in a single call, then you can use the `MSET <key1> <value1> <key2> <value2>` and `MGET <key1> <key2>` commands instead of issuing multiple calls. In all cases, if a key does not exist the `GET` command and its variants return a `NULL` value. If only an existence check is required, the `EXISTS` command is also available. The existence command is usually used in concert with Redis' scripting functionality, which is discussed later.

The next-most-complicated data structure in Redis is the hash table. This form of key allows for the storage of subkeys similar to the `Map` collection in Java or a Dictionary type in other languages such as JavaScript. Instead of the simple `GET` and `SET` commands, you use `HSET <key> <subkey> <value>` and `HGET <key> <subkey>` to access elements of the hash table with `HMSET` and `HMGET` allowing for updates or retrieval of multiple subkeys from a hash table. To retrieve all subkeys and their values, use `HGETALL` with `HKEYS` and `HVALUES` retrieving all subkey names or subkey values respectively.

Both normal keys and hash tables support atomic counters. For normal keys, the `INCR` and `INCRBY` commands increment the value of the key by 1 for the former or the value specified by the command. In both cases, the string value contained within the key is interpreted as a 64-bit signed integer for the purposes of arithmetic. Decrementing a counter is handled by the `DECR` and `DECRBY` commands. The `INCRBYFLOAT` command works like `INCRBY`, except that the string contained in the value is interpreted as a double-precision floating-point value.

Atomic counters for hash tables operate on subkeys. Rather than implement the full suite of counter commands, hash tables only implement a variant of `INCRBY` and `INCRBYFLOAT`, called `HINCRBY` and `HINCRBYFLOAT`, respectively.

Tip

At first glance, a Hash with its subkeys is unnecessary because there is already a key-value store. However, there are many cases where using a hash makes more sense than simply using keys. First, if there are a few different metrics to be stored, the Hash storage mechanism can actually be more efficient than using separate keys. This is because Redis' hash table implementation uses ziplists for small hashes to improve storage performance. Second, if these metrics are all related then the `HGETALL` command can be used to retrieve all of the metrics in a single command. The nature of the hash structure still lets all of the keys be updated independently so there is no real disadvantage. Finally, it is common to expire data in the Redis database after some time. Largely, this is done to control the memory footprint of the database. Using a hash table means only having to expire a single key, which helps to maintain consistency in reporting.

The list data structure in Redis is often used to implement various queue-like structures. This is reflected in the command set for the list data type, which is focused on pushing and popping elements from the list. The basic commands are `LPUSH` (`RPUSH`) and `LPOP` (`RPOP`). Redis lists are bidirectional, so the performance is the same for both left and right inserts. Using the same “side” of command (for example, `LPUSH` and `LPOP`) results in stack, or Last-In-First-Out (LIFO), semantics. Using opposite-sided commands (for example, `LPUSH` and `RPOP`) results in queue, or First-In-First-Out (FIFO), semantics.

Implementing a Reliable Queue

Redis provides a fairly simple mechanism for implementing a reliable queue using the `RPOPLPUSH` (or its blocking equivalent) and the `LREM` commands. As the name suggests, the `RPOPLPUSH` command pops an element off the tail of a list, like the `RPOP` command, and then pushes it onto the head of a second list, like the `LPUSH` command, in a single atomic action. Unlike separate `RPOP` and `LPUSH` commands, this ensures that the data cannot be lost in transit between the clients. Instead, the item to be processed remains on an "in-flight" list of elements currently being processed.

The command returns the element moved between the two lists for processing. When processing is complete, the `LREM` is executed against the processing list to remove the element from the processing list. This simple example shows the basic process for implementing the queue:

```
redis 127.0.0.1:6379> LPUSH queue item1 item2 item3 item4 item5
(integer) 5
redis 127.0.0.1:6379> RPOPLPUSH queue processing
"item1"
redis 127.0.0.1:6379> LRANGE queue 0 100
1) "item5"
2) "item4"
3) "item3"
4) "item2"
redis 127.0.0.1:6379> LRANGE processing 0 100
1) "item1"
redis 127.0.0.1:6379> LREM processing 0 item1
(integer) 1
redis 127.0.0.1:6379> LRANGE processing 0 100
(empty list or set)
redis 127.0.0.1:6379>
```

The "Implementing a Reliable Queue" example demonstrates pushing several items onto the `queue` list. Next an element is moved from the `queue` list to the `processing` list. Finally, the element is removed from the processing list.

When implementing this using a client, it is usually more common to use the `BRPOPLPUSH` command. This is a blocking version of the normal command that will wait until an element is pushed onto the list. This avoids the need to poll on the client side. The command supports a timeout feature in case the client needs to perform some other function.

Another client is usually implemented to handle elements that have been placed on the processing list but never removed. The cleanup client can periodically scan the list, and if it sees an element twice, push it back onto the `queue` list for reprocessing. There is no need to remove the original item from the processing queue, a successful `LREM` will remove all copies of the element and the processing queue should generally remain small, so size is not a concern.

The other two data types implemented by Redis are Sets and Sorted Sets. Like the Java `Set` collection, these data structures maintain lists of distinct values. Most of the Set and Sorted Set operations have `O(n)` processing time in the size of the set.

Basic sets are fairly simple constructs and useful for keeping track of unique events or objects in a real-time stream. This can become infeasible when the size of the set gets very large, mostly due to the space requirements, which are linear in the number of unique elements in the set. Adding elements to the set is accomplished through the `SADD` command, which takes a *`KEY`* and any number of elements to add. It returns the number of elements newly added to the set. Elements are deleted from

the set using the SREM command. It has the same arguments as the SADD command. The SPOP command works similarly to SREM, but returns and removes a random element from the set. The SMOVE command combines the SADD and SREM commands into a single atomic operation to move elements between two different sets.

Set membership queries are done using the SISMEMBER command. This command, unfortunately, only takes a single element as its argument. To implement "any" or "all" variants of membership queries, multiple SISMEMBER commands must be issued, which is accomplished using Redis' scripting capabilities, discussed later in this section.

Redis also supports set operations between different set objects in the database. The SUNION, SINTER, and SDIFF commands return a bulk reply containing, respectively, the union, intersection, or difference of two sets. It is also possible to store the result into another set by adding the STORE modifier to each of the commands. For example, to store the union of two sets into a third set, you use the SUNIONSTORE command. You can obtain the size of any set using the SCARD command.

Sorted Sets add a score to each element of the set, which defines the sort order for the elements within the set. This is useful for maintaining leaderboards and other similar structures. Generally, the commands for updating a sorted set are the same as those used for a normal set, but prefixed by a Z instead of an S and taking an extra "score" argument. In the case of the ZADD command, the score comes before each element to be added. If the element already exists, the existing score is updated with the new score.

The union and intersection operations are supported by sorted sets as the ZUNIONSTORE and ZINTERSTORE commands. These operations work a bit differently than their unsorted counterparts because they have to handle the possibility of a shared element having two different scores. Both operations take an optional WEIGHTS parameter, which defines a multiplicative factor to apply to the score of each set. For example, WEIGHTS 2 3 would multiply the first set's scores by 2 and the second set's scores by 3. By default, these two scores would be added together for each element present in both sets. This can be modified by the optional AGGREGATE parameter. This parameter takes a value of SUM (the default), MIN, or MAX. When SUM is selected the scores of elements contained in two or more sets will be added together after their weight has been applied. When MIN is used, the smallest weighted value is used. For MAX, the largest weighted value is used.

Sorted sets introduce a few new commands related to using the scores associated with each element. The ZINCRBY command increments the score of an element in the set. If the element is not present, it is assumed to have a score of 0.0 and is inserted into the sorted set at that time.

The ZCARD command works the same way as SCARD, returning the cardinality of the entire set. In addition, the ZCOUNT command returns the number of elements between two scores. For example,

```
ZCOUNT myzset 1 3
```

returns the number of elements between 1 and 3 inclusive (that is, $1 \leq x \leq 3$). To return an exclusive count (that is, $1 < x < 3$), a (is prepended to the scores like so:

```
ZCOUNT myzset (1 (3.
```

To retrieve the elements within a range rather than the count, the ZRANGEBYSCORE and ZREVRANGEBYSCORE commands are used. Their arguments are the same as ZCOUNT, but it is important to remember that ZREVRANGEBYSCORE expects its first argument to be larger than its second argument. For example,

```
ZREVRANGEBYSCORE myzset 1 3
```

does not return any elements, whereas

```
ZREVRANGEBYSCORE myzset 3 1
```

returns the expected results. Both commands take an optional `WITHSCORES` parameter that will return the element and its associated score. They also both support a `LIMIT` parameter that takes an offset and a count for implementing paged interfaces. The offset requires that the sorted set be traversed so it can be slow for very large offsets.

For applications like leaderboards, the `ZRANGE` and `ZREVRANGE` return the elements between positions `start` and `stop`. These positions are 0-indexed so as to retrieve the top 10 values by score. For example,

```
ZREVRANGE myzset 9 0
```

returns the appropriate values. Like many programming languages, the same effect can be achieved with `ZRANGE` and negative positions. The –1 position is the last element of the set and so on.

Scripting

Redis 2.6 introduced a built-in Lua scripting engine to eventually take the place of its existing transaction facility. This functionality is accessed through the `EVAL` command, which has the following form:

```
EVAL <script> <num keys> <key 1> ... <key n> <arg 1> ... <arg n>
```

Scripts can sometimes be quite long, so Redis also provides an `EVALSHA` command, which uses the fixed-size SHA1 hash in place of the actual script. If the redis server does not know about a particular script it returns a special error that instructs the `EVALSHA` to fall back to the normal `EVAL` command. Most clients do this implicitly when executing an `EVAL` command, so it is usually not necessary to take this into account.

All scripts, no matter the client connection, execute in a single Lua interpreter and can be considered to be essentially atomic operations when compared against other scripts.

Most Recent Event Tracking

The inclusion of a scripting language directly in the Redis engine makes it easy to do a few things that are otherwise difficult to get right. One of those things is "Most Recent Event Tracking." This is often used for things like attribution modeling where some target event is correlated against the most recent previous event.

Doing this in most key-value stores usually involves simply assuming that events are being processed in time "ordered enough" such that it is unlikely that any two events for a given user will get "out of order." In that case, a simple `SET` operation suffices.

However, in many cases, the ordering assumption does not hold or more complicated arrangements are needed. For example, it may be that some event types are more important than others, so even if an event occurs later, it should not become the "most recent" if it is a lesser event. In a normal key-value system, this requires multiple round-trips to the server. First, the current value is obtained so the update logic can be applied. If the update succeeds, a second round-trip updates the event.

Using Lua scripting, this update process can be done entirely on the server side. Assuming that the events are JSON objects with a `ts` field representing the timestamp of the event, an update script would look something like this:

```
if redis.call("EXISTS",KEYS[1]) == 1 then
  local current  = tonumber(    cjson.decode(     redis.call("GET",KEYS[1])    )["ts"]
)
  local new      = tonumber(cjson.decode(ARGV[1])["ts"])
  if(new >= current) then
    redis.call("SET",KEYS[1],ARGV[1])
    return 1
  else
    return 0
  end
else
  redis.call("SET",KEYS[1],ARGV[1])
  return 1
end
```

To try it out, save this script as `timestamps.lua` or use the one included in the book's source code. After starting the Redis server, try setting a few different events with timestamps:

```
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":0}'
(integer) 1
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":2}'
(integer) 1
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":10}'
(integer) 1
$ redis-cli EVAL "$(cat timestamps.lua)" 1 user '{"ts":5}'
(integer) 0
$ redis-cli GET user
"{\"ts\":10}"
```

Even though timestamps are presented out of order, the most recent timestamp is still preserved. More complicated tests of the JSON objects could be included after line 4 of the preceding code as well. This also has the advantage of being an atomic operation because updates to the user are only done through the script, which will always execute in the same Lua interpreter.

Publish/Subscribe Support

In addition to being a key-value store, Redis can serve as a simple message bus. This is particularly useful in streaming applications as it allows a processing system like Storm to notify a front end of important events. An example of this is given in Chapter 7, "Delivering Streaming Metrics," which uses this facility to implement a basic real-time dashboard.

Events in Redis are simple text values that are published to a channel using the `PUBLISH` command:

```
PUBLISH achannel "some message"
```

These `PUBLISH` commands are delivered to clients who have subscribed to one or more channels using the `SUBSCRIBE` command. Conveniently, these commands are replicated just like any other command in Redis so they will be delivered to clients who have subscribed to a slave of the master. However, all publish commands must be issued on the master.

Replication

Redis supports basic master-slave replication. It generally works fairly well, but as of the 2.6 series, a loss of synchronization between the master and slaves requires a full replication. For large installs, this can take a long time, and if it fails, must begin again. This is usually a larger problem when attempting to replicate Redis databases across datacenters.

The current beta version of Redis, which will be the 2.8 series when released, supports partial resynchronization. This alleviates many of the difficulties that arise with unreliable network links and Redis where the cluster becomes unavailable because the slave is unable to complete an initial synchronization in a reasonable time.

Setting up a Redis master-slave replication environment is very simple. The master usually does not require any changes, and slaves simply add a `slaveof` directive to their configuration file:

```
slaveof <master ip address> <port>
```

This can even be applied from the Redis command-line client without restarting the server application using the `CONFIG SET` command. However, you should take care to ensure that the directive has been added to the server's configuration, otherwise the change will be lost if the server restarts for any reason. Because a Redis master requires no special configuration, it is actually possible to slave a Redis server to another slave. The applications for this are very limited, and *you should not use it in a production environment*, but it can sometimes be useful for maintaining a local development Redis server with access to production data.

Clustering/Sharding Databases

At the moment, Redis does not have a native sharding solution like many other databases in the "NoSQL" world. There has been a sharded version of Redis called Redis Cluster in the works for some time, but it has never been released in a production form.

The only real option for sharding a Redis database right now is Twitter's twemproxy tool (also known as nutcracker). This is a proxy server for both Redis and Memcached that implements sharding features for both of these key-value stores. Using twemproxy, servers are combined into pools and writes are distributed across members of the pool using consistent hashing. This allows for both redundancy and partitioning of the data across multiple servers. The drawback is that not all Redis operations are supported, so if complicated scripting or the publish/subscribe mechanism are used then it may not be possible to use twemproxy without additional architectural considerations, such as keeping a sharded and unsharded server pool.

MongoDB

Unlike Redis, MongoDB is a schema-less document store. It has been developed by 10gen, a company that provides enterprise-level sales and support for MongoDB installations with either a Software-as-a-Service or an On-Premise model depending on the needs of the organization. It also provides a Community edition, which does not provide many of the security features available to the Enterprise edition. It is the version discussed in this section.

The MongoDB Model

The documents in a MongoDB are schema-less data structures represented as JSON objects, which would be analogous to records in a traditional database system or a value in a key-value store.

Although each of these documents has a unique identifier that can be used to address it directly, the intention is that documents will be manipulated *en masse* by grouping them into *collections*. These collections are roughly analogous to a table in a relational database system, and it is expected that document objects contained within a collection will largely share the same structure.

These collections can be queried through a JSON-based query language. To improve performance, elements of each document can be indexed using a secondary-index system. MongoDB provides a variety of interesting indexing options that make it a popular choice for certain applications, particularly those that deal with physical spaces, because it has out-of-the-box support for indexing and querying geographic information.

Getting Set Up

To get set up with MongoDB, the 10gen MongoDB website has packages available for a number of platforms at http://mongodb.org/downloads. Unlike Redis and many other NoSQL options, MongoDB even supports Windows as a first-class citizen. For deployment on Linux systems, Ubuntu and Debian packages are available, making deployment to platforms like EC2 fairly easy as well.

The MongoDB server executable is called `mongod` and starting it without arguments causes it to look for a `/data/db` directory. You can override this by passing the `--dbpath` argument on the command line:

```
$ mkdir db
$ ./mongod --dbpath ./db
[initandlisten] MongoDB starting : pid=6722 port=27017 dbpath=./db
[initandlisten]
[initandlisten] ** WARNING: soft rlimits too low. Number of files is
  256, should be at least 1000
[initandlisten] db version v2.4.8
[initandlisten] git version: a350fc38922fbda2cec8d5dd842237b904eafc14
[initandlisten] build info: Darwin bs-osx-106-x86-64-2.10gen.cc 10.8.0
  Darwin Kernel Version 10.8.0: Tue Jun  7 16:32:41 PDT 2011;
  root:xnu-1504.15.3~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
[initandlisten] allocator: system
[initandlisten] options: { dbpath: "./db" }
[initandlisten] journal dir=./db/journal
[initandlisten] recover : no journal files present, no recovery needed
[FileAllocator] allocating new datafile ./db/local.ns, filling with
  zeroes...
[FileAllocator] creating directory ./db/_tmp
[FileAllocator] done allocating datafile ./db/local.ns, size: 16MB, took
[FileAllocator] allocating new datafile ./db/local.0, filling with
[FileAllocator] done allocating datafile ./db/local.0, size: 64MB, took
```

```
  0.327 secs
[initandlisten] command local.$cmd command: { create: "startup_log",
  size: 10485760, capped: true }
  ntoreturn:1 keyUpdates:0  reslen:37 470ms
[websvr] admin web console waiting for connections on port 28017
[initandlisten] waiting for connections on port 27017
```

Most of the startup information for the server is informational, but there are some items of note. First, the warning about soft `rlimits` is important. Most operating systems (Linux included) have defaults that assume something much closer to a "consumer" system than a "server" system, especially a database server. As a result, many of the limits on memory and open files are set unusably low for production environments. Even distributions that bill themselves as "server" distributions suffer from this problem. For development it is probably not necessary to raise this limit, but a production server should set this limit fairly high because a large database could have thousands of resources in use at any given moment.

The other item to note is that MongoDB pre-allocates some files when it starts for the first time. These files are simply filled with zeros because they will be memory mapped by MongoDB and filled with data. If the size of the data on disk is roughly known, then these empty files can be created before starting MongoDB for faster initial performance. Otherwise, MongoDB pauses each time it needs to create a new file, which can be quite often if the server is joining an existing cluster of servers and replicating all existing data. On UNIX-like systems, this is easily accomplished using the `dd` command to create empty files. For example, this MongoDB install contains only the initial system database called `local`:

```
$ ls -lh
total 163840
drwxr-xr-x  2 bellis  staff    68B Dec  6 21:29 journal
-rw-------  1 bellis  staff    64M Dec  2 20:07 local.0
-rw-------  1 bellis  staff    16M Dec  2 20:07 local.ns
```

To add a new database called big, files of 2GB each are created using the `dd` command. The database is expected to require approximately 4GB of space, so 2 files are created:

```
$ dd if=/dev/zero of=./big.0 bs=1048576 count=2048
2048+0 records in
2048+0 records out
2147483648 bytes transferred in 8.599453 secs (249723278 bytes/sec)
$ dd if=/dev/zero of=./big.1 bs=1048576 count=2048
2048+0 records in
2048+0 records out
2147483648 bytes transferred in 13.700213 secs (156748195 bytes/sec)
$ ls -lh
total 8552448
-rw-r--r--  1 bellis  staff   2.0G Mar  7 11:42 big.0
-rw-r--r--  1 bellis  staff   2.0G Mar  7 11:42 big.1
drwxr-xr-x  2 bellis  staff    68B Dec  6 21:29 journal
-rw-------  1 bellis  staff    64M Dec  2 20:07 local.0
-rw-------  1 bellis  staff    16M Dec  2 20:07 local.ns
```

Using MongoDB Databases and Collections

A MongoDB database is a namespace in which a group of collections is kept. By default, MongoDB has a single database called `local` that contains system-level collections. Although collections can be created in this database, doing so is probably a bad idea because this collection is not included in replication or sharding. In fact, the `local` database contains the specialized collections used to

manage replication and sharding.

MongoDB's on-disk representation uses the name of the database as the name of the file along with a segment number. These are created lazily so a database will not "exist" until its first collection is created. To see this, try using the mongo client to switch to a database called `mine` from a fresh install:

```
> use mine;
switched to db mine
> show databases;
local  0.078125GB
test  (empty)
```

Notice that the `local` database is listed, but not the `mine` database. However, after executing a `createCollection` command, the database does exist:

```
> db.createCollection("first");
{ "ok" : 1 }
> show dbs;
local  0.078125GB
mine  0.203125GB
test  (empty)
```

Checking the directory given at startup by `dbpath` also shows that the first segment files now exist on disk:

```
$ ls
_tmp    journal    local.0    local.ns  mine.0    mine.1    mine.ns
mongod.lock
```

Collections, created with the `db.createCollection` command are a fairly loose concept in MongoDB. They are roughly analogous to tables, but because MongoDB is schema-less there is no requirement that any document inside of a collection resemble any other. Each document inside of a collection is represented as a JSON object, and all documents have a primary key field `_id` that is usually an automatically generated `ObjectId` object. However, the `_id` field can be any type of object, except an `Array`, so long as it is unique within a collection. For the purposes of streaming analysis, it is often useful to use this field as a form of compound primary key to make it possible to update a specific document without having to first query the collection to find the appropriate `_id` value (or use a more complicated update query).

Documents may not contain fields that start with `$`. This character is reserved for special objects in a MongoDB document, most commonly a `DBRef` object that allows MongoDB documents to be linked together. These objects are JSON objects containing `$ref`, `$id`, and, optionally, `$db` fields. The `$id` field identifies the value of the `_id` field of the target document; the `$ref` field identifies the target collection; and the `$db` field identifies the target database if it is different than the current database. For example, a reference to a document in the first collection might look like this:

```
{$ref:"first",$id:ObjectId("5126bc054aed4daf9e2ab772"),$db:"mine"}
```

Although documents in collections can be highly nested, a document can be at most 16MB in size. This limitation is largely arbitrary on the part of MongoDB's developers, but they are correct in their assertion that very large documents should be a warning sign during data modeling.

Capped Collections

MongoDB supports creating new collections as a "capped collection." These are fixed-size

collections designed to support high-throughput operations. For example, MongoDB uses capped collections to implement the operation log for its own replication strategy.

Capped collections implement a FIFO queue with a few restrictions. First, capped collections may not be sharded. Among other things, it would make it difficult to maintain the FIFO queue. Second, while documents written to a capped collection may be changed, they may not grow in size. For example, changing a string field from “foo” to “bar” is fine, but changing from “foo” to “foobar” is not allowed. Finally, documents cannot be deleted from a capped collection. They are removed as they “age out,” but there is no way to delete a specific document.

Converting a Collection to A Capped Collection

It is possible to convert a normal collection to a capped collection. The command is not part of the standard set of operations, but you can access it via the `runCommand` method. For example, to convert the `first` collection to a capped collection holding 1MB of data use:

```
> db.runCommand({"convertToCapped": "first", size: 1024*1024});
```

Take care when running this command as it takes a global write lock. All write operations on the database will block until the conversion has been completed. Never attempt it on a server in rotation.

Capped collections can be queried in the usual way, but more often a "tailable cursor" is used to read data in the order it was inserted. This is more or less equivalent to using the Unix `tail -f` command.

The most common use case cited for capped collections is as a data transport mechanism. In the architectures used in this book, it would replace either Kafka or Flume in the data transport layer. However, capped collections suffer from a major flaw as a data transport tool because the collection is capped to a size, not to a time. As discussed in Chapter 5, "Processing Streaming Data," this has failure modes that are very difficult to monitor. As such, using MongoDB's capped collections for this application is not recommended.

Basic Indexing

The primary role of a collection is to maintain a set of indexes common to a group of documents. The assumption is that the indexes represent important points of commonality for the document in the MongoDB collection.

Indexes in MongoDB are defined using the `db.collection.createIndex` function and a JSON object that defines the direction of sorting for each field contained within the index:

```
> db.first.ensureIndex({foo:1,bar:-1});
> db.first.stats()
{
  "ns" : "mine.first",
  "count" : 0,
  "size" : 0,
  "storageSize" : 8192,
  "numExtents" : 1,
  "nindexes" : 2,
  "lastExtentSize" : 8192,
  "paddingFactor" : 1,
  "systemFlags" : 1,
  "userFlags" : 0,
  "totalIndexSize" : 16352,
  "indexSizes" : {
    "_id_" : 8176,
    "foo_1_bar_-1" : 8176
  },
  "ok" : 1
}
```

This command creates an index on the `first` collection created earlier on two fields: `foo` and `bar`. The `foo` field is sorted in an ascending natural ordering for its contents (numerical for numerical values and lexical for string values), whereas `bar` is sorted in descending order. Using dot-notation, it is possible to index on nested fields in a document. For example, the following

defines an index on the `zip` subfield of the `addr` object in a document:

```
> db.first.ensureIndex({"addr.zip":1});
> db.first.getIndexes();
[
  {
    "v" : 1,
    "key" : {
      "addr.zip" : 1
    },
    "ns" : "mine.first",
    "name" : "addr.zip_1"
  }
]
```

When the field being indexed contains an `Array` object rather than a scalar object, MongoDB automatically creates a multikey index. Each element of the array is added to the index individually, so an array containing four elements would be added to the index four times. This even works with arrays that contain objects. In the previous example, if the `addr` field contained an `Array` of address objects, each of the address objects' `zip` fields would be added to the index.

Geospatial Indexing

Aside from the basic scalar field indexes, MongoDB supports several specialized types of index that make it popular for certain applications. The first is the geospatial index, which operates either on coordinate pairs or GeoJSON objects. Coordinate pairs are an older format specific to MongoDB, represented either as an array of two elements or as an object containing `lng` and `lat` fields. These two representations are equivalent:

```
{lng: 10, lat: 20}
[10,20]
```

GeoJSON objects are a newer open source format used to describe geospatial features. To represent a coordinate pair using GeoJSON, the format looks like this:

```
{type:"Point",coordinates:[10,20]}
```

In addition to Point types, MongoDB also supports Line and Polygon types when indexing.

MongoDB can index these coordinates by specifying `2dsphere`, `2d`, or `haystack` instead of a number when adding a field to an index. For most applications, `2dsphere` is probably the most appropriate choice for indexing. However, if the area to be indexed is geospatially small, `haystack` indexing can improve query performance.

Full Text Indexing

When an index is created with a field's index type set to `text` rather than a number, MongoDB's full text indexing is enabled. This is currently a beta feature, so this capability must be enabled when starting the server by adding `--setParameter textSearchEnabled=true` to the command line.

In many ways, this type of index is similar to the multikey index used when the field is an array. The primary difference is that the field is expected to be a string, which is tokenized and stemmed to form the words entered into the index. MongoDB also maintains a list of language-specific stop words that are dropped from the indexing process.

Queries for a set of search terms are computed using a scoring system. MongoDB maintains scores for different words for each language it supports, but this may be overridden when creating the index. The optional parameter is given in a later section.

This form of indexing requires a fair bit of processing power, so it should be used with caution. Inserting documents into a full text indexed collection will also incur a processing cost associated with the tokenization and stemming process, so it is not recommended for high-throughput collections.

Other Indexing Options

When the index is created it is automatically given a name. This name can be at most 125 characters when combined with the collection name, which can be exceeded when building complicated indexes. To avoid this, or for aesthetic reasons, indexes can be named using the options object when creating the index. The options object also allows for a variety of other optional parameters, given in Table 6.1.

Table 6.1 Table 6-1: Optional Parameters for MongoDB Index Creation

Name	Type	Description
`background`	`true/false`	Causes the index to be built in the background. If this is `false`, the database will be unresponsive while the index is built. If the collection is large, this can mean minutes of downtime.
`unique`	`true/false`	Requires that all the values in this index are unique. This is set on the index automatically created for the `_id` field.
`name`	string	Overrides the auto-generated name of the index
`dropDups`	`true/false`	Creates a unique index by deleting documents with the same index value after the first has been found. Use with caution.
`sparse`	`true/false`	Only include documents that contain the fields specified by the index. Can result in smaller indexes if many documents do not contain a particular field.
`expireAfterSeconds`	integer	The time-to-live for documents according to the timestamp used for this field. Setting this option causes the index to become a TTL index. It cannot be used on compound indexes, and the field *must* be a date type.
`v`	version	Different versions of MongoDB use different indexing formats and schemes. The scheme can be chosen with this parameter. This parameter should never be used.
`weights`	JSON	For full-text indexes, this allows different weights to be assigned to different words. This is used in calculating the search score during queries.
`default_language`	string	Defines the language used by the full-text indexing engine. It defines the stoplist and stemming procedure as well as the default scores for words.
`language_override`	string	Defines a field that can be used to override the language used by the full-text indexer at the document level. It defaults to the field `language`.

Source:
http://docs.mongodb.org/manual/reference/method/db.collection.ensureIndex/#db.collection.ensureInd

Inserts and Updates

MongoDB supports the usual insert and update operations for its collections. The `insert` command can insert both single elements and, if passed an `Array`, multiple documents in a single command:

```
> db.first.insert({metric:"test",n:1});
> db.first.insert([{metric:"test2",n:1},{metric:"test3",n:1}]);
> db.first.find();
{ "_id" : ObjectId("529c1bb69c88a09b200d9cac"), "metric" : "test",
  "n" : 1 }
{ "_id" : ObjectId("529c1bca9c88a09b200d9cad"), "metric" : "test2",
  "n" : 1 }
{ "_id" : ObjectId("529c1bca9c88a09b200d9cae"), "metric" : "test3",
  "n" : 1 }
```

The `insert` command automatically creates the `_id` key if it is not specified. Most streaming applications will want to explicitly specify the `_id` to make updates easier:

```
> db.first.insert({_id:"test:201312010000"
  ,ts:new Date(2013,11,01,00,00),metric:"test",n:1});
> db.first.find();
{ "_id" : "test:201312010000",
  "ts" : ISODate("2013-12-01T08:00:00Z"), "metric" : "test", "n" : 1 }
```

Attempting to insert a document with a duplicate key fails with an error:

```
> db.first.insert({_id:"test:201312010000",
  ts:new Date(2013,11,01,00,00),metric:"test",n:1});
  E11000 duplicate key error index: mine.first.$_id_
  dup key: { : "test:201312010000" }
```

Updating documents is accomplished via the `update` command. This command takes a query, which will often simply be a match on the `_id` field, and a replacement document. By default, this completely replaces the matching document(s), requiring that the entire document be replicated. For example, this is probably not the desired result for this update:

```
> db.first.update({_id:"test:201312010000"},{n:2});
> db.first.find()
{ "_id" : "test:201312010000", "n" : 2 }
```

Notice that the entire document has been replaced, when the desired result was probably to simply update the `n` field. To do that the special `$set` field is used to define the fields to update:

```
bb.first.drop();
> db.first.insert({_id:"test:201312010000",
  ts:new Date(2013,11,01,00,00),metric:"test",n:1});
> db.first.update({_id:"test:201312010000"},{"$set":{n:2}});
> db.first.find()
{ "_id" : "test:201312010000",
  "ts" : ISODate("2013-12-01T08:00:00Z"), "metric" : "test", "n" : 2 }
```

In addition to `$set`, updating can also increment a numerical field using `$inc`:

```
> db.first.update({_id:"test:201312010000"},{"$inc":{n:2}});
> db.first.find()
{ "_id" : "test:201312010000",
  "ts" : ISODate("2013-12-01T08:00:00Z"),
  "metric" : "test", "n" : 4 }
```

A MongoDB Metric Collection

Creating a simple metric collection in MongoDB is easy using the `upsert` feature of the update command. To begin, create a metric collection and define an index with a time-to-live to allow the data to expire after seven days:

```
> db.createCollection("metrics");
{ "ok" : 1 }
> db.metrics.ensureIndex({ts:1},{expireAfterSeconds:86400*7});
> db.metrics.getIndexes();
[
  {
    "v" : 1,
    "key" : {
      "_id" : 1
    },
    "ns" : "mine.metrics",
    "name" : "_id_"
  },
  {
    "v" : 1,
    "key" : {
      "ts" : 1
    },
    "ns" : "mine.metrics",
    "name" : "ts_1",
    "expireAfterSeconds" : 604800
  }
]
```

The ideal situation would be to use the `_id` field as a date; each one will be unique, but this unfortunately does not work. Instead the timestamp to be aggregated must be kept in two keys. In the long run, this is probably desirable because metrics may be broken down to something more granular. In this example, a `customer_id` will be added to the `_id` to allow for aggregation of metrics for each customer.

The aggregation itself is accomplished through update commands:

```
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.visitors":1}},{upsert:true});
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.clicks":1}},{upsert:true});
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.views":1}},{upsert:true});
> db.metrics.update({_id:"1:201312010000"},
  {$set:{customer_id:1,
  ts:new Date(2013,12-1,01,00,00)},
  $inc:{"metrics.visitors":1}},{upsert:true});
> db.metrics.find();
{ "_id" : "1:201312010000", "customer_id" : 1,
  "metrics" : {
    "clicks" : 1,
  "views" : 1,
```

```
      "visitors" : 2
    },
    "ts" : ISODate("2013-12-01T08:00:00Z")
  }
```

Notice that the `upsert` option is set to **`true`**. This causes the document to be created if one that matches it does not exist. This is why the `customer_id` and `ts` fields are set on each call, to ensure they are set to the appropriate value if the document must be created.

Queries and Aggregation

The update command has used simple queries to identify the document to change. MongoDB, in general, has a fairly rich query language, although it can often be cumbersome as it is expressed as JSON, just like any other document. Queries are conducted using the `find` command, and they rely heavily on the indexing of collections to achieve performance. If an index is not available, then all of the documents in a collection will be scanned.

The queries used in the previous section are simple matching queries, equivalent to using an = in the `WHERE` clause of a SQL query. If the field being queried is an array, this equality query returns the document if any of the elements of the array matches the query value.

To use inequalities to specify range queries, the `$lt` and `$gt` fields can be used (along with `$lte` and `$gte` for inclusive inequalities). For example, to find all entries with a timestamp after midnight on December 1st, the following query could be used:

```
> db.metrics.find({customer_id:1,ts:{$gte:new Date(2013,11,01)}});
```

To match any of several options, the `$in` field can be used in the query along with an array:

```
> db.metrics.find({customer_id:{$in:[1,2,3,]}});
```

Metric Collection Query Performance

MongoDB has a method called `explain`, which can be called on a query to show the query plan that will be executed by a particular query. This is particularly useful for debugging performance bottlenecks in MongoDB environments.

For example, the query plan to search for the metrics from a particular customer reveals that a scan would be required under the current indexing scheme:

```
> db.metrics.find({customer_id:1}).explain()
{
  "cursor" : "BasicCursor",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
  }
}
```

However, if the timestamp is used, the index used to implement the time-to-live feature is used to improve performance:

```
> db.metrics.find({customer_id:1,ts:{$gt:new Date(2013,11,01)}})
  .explain()
{
  "cursor" : "BtreeCursor ts_1",
  "isMultiKey" : false,
  "n" : 0,
  "nscannedObjects" : 0,
  "nscanned" : 0,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 2,
  "indexBounds" : {
    "ts" : [
      [
        ISODate("2013-12-01T08:00:00Z"),
        ISODate("0NaN-NaN-NaNTNaN:NaN:NaNZ")
      ]
    ]
  }
}
```

Of course, a scan of all the customers would still be required. If the number of customers is small, this may not be a problem, but the hope is that there would be a very large number of customers. In that case, it might seem like a good idea to add a customer index:

```
> db.metrics.ensureIndex({customer_id:1});
> db.metrics.find({customer_id:1,ts:{$gt:new Date(2013,11,01)}})
  .explain()
{
  "cursor" : "BtreeCursor ts_1",
  "isMultiKey" : false,
  "n" : 0,
  "nscannedObjects" : 0,
  "nscanned" : 0,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 2,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "ts" : [
      [
        ISODate("2013-12-01T08:00:00Z"),
        ISODate("0NaN-NaN-NaNTNaN:NaN:NaNZ")
      ]
    ]
  }
}
```

Unfortunately, MongoDB can really only use a single index in a given query, so adding the `customer_id` index cannot help with this query. To improve performance, a compound index would have to be added:

```
> db.metrics.ensureIndex({customer_id:1,ts:1});
> db.metrics.find({customer_id:1,ts:{$gt:new Date(2013,11,01)}})
  .explain()
{
  "cursor" : "BtreeCursor customer_id_1_ts_1",
  "isMultiKey" : false,
  "n" : 0,
  "nscannedObjects" : 0,
  "nscanned" : 0,
  "nscannedObjectsAllPlans" : 0,
  "nscannedAllPlans" : 0,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 2,
  "indexBounds" : {
    "customer_id" : [
      [
        1,
        1
      ]
    ],
    "ts" : [
      [
        ISODate("2013-12-01T08:00:00Z"),
        ISODate("0NaN-NaN-NaNTNaN:NaN:NaNZ")
      ]
    ]
```

```
    }
  }
```

Of course, the old `customer_id` index is still used by the original customer query:

```
> db.metrics.find({customer_id:1}).explain()
{
  "cursor" : "BtreeCursor customer_id_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 3,
  "indexBounds" : {
    "customer_id" : [
      [
        1,
        1
      ]
    ]
  }
}
```

However, the same query can use the compound query as well, so the single `customer_id` index is now redundant and can be dropped:

```
> db.metrics.dropIndex("customer_id_1");
{ "nIndexesWas" : 4, "ok" : 1 }
> db.metrics.find({customer_id:1}).explain()
{
  "cursor" : "BtreeCursor customer_id_1_ts_1",
  "isMultiKey" : false,
  "n" : 1,
  "nscannedObjects" : 1,
  "nscanned" : 1,
  "nscannedObjectsAllPlans" : 1,
  "nscannedAllPlans" : 1,
  "scanAndOrder" : false,
  "indexOnly" : false,
  "nYields" : 0,
  "nChunkSkips" : 0,
  "millis" : 0,
  "indexBounds" : {
    "customer_id" : [
      [
        1,
        1
      ]
    ],
    "ts" : [
      [
        {
          "$minElement" : 1
        },
```

```
        {
          "$maxElement" : 1
        }
      ]
    ]
  }
}
```

Like SQL databases, MongoDB also offers facilities for grouping and aggregating data in queries. The original facility for aggregation was either the `group()` or `mapReduce()` commands, but versions of MongoDB after 2.2 also support an optimized `aggregate()` command.

Unlike SQL, the pipeline command uses a pipeline approach for computing its results, taking an array of filtering and grouping commands used to reach a final result. This is easiest to understand in action, so first build a collection with some example data:

```
> abc = ['A','B','C','D','E','F','G','H','I','J','K','L',
  'M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'];
> db.createCollection("aggtest");
> for(var i=0;i<1000;i++) {
... db.aggtest.insert({
... first:abc[Math.floor(Math.random()*abc.length)],
... second:abc[Math.floor(Math.random()*abc.length)],
... count:Math.floor(1000*Math.random())
... });
... }
> db.aggtest.find({})
{ "_id" : ObjectId("53213bc8ae5fcad63d0563e9"),
  "first" : "S", "second" : "W", "count" : 762 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ea"),
  "first" : "E", "second" : "V", "count" : 381 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563eb"),
  "first" : "Q", "second" : "O", "count" : 143 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ec"),
  "first" : "C", "second" : "I", "count" : 601 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ed"),
  "first" : "B", "second" : "C", "count" : 413 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ee"),
  "first" : "M", "second" : "D", "count" : 790 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563ef"),
  "first" : "S", "second" : "Q", "count" : 699 }
{ "_id" : ObjectId("53213bc8ae5fcad63d0563f0"),
  "first" : "A", "second" : "M", "count" : 615 }
... other output omitted
Type "it" for more
```

The first stage of an aggregation pipeline is usually a filtering step that acts like the `WHERE` clause of a SQL statement. It is identified by a `$match` statement, as in this example, which selects all of the elements with the “A” as their value for the “first” element:

```
> db.aggtest.aggregate([{$match:{first:"A"}}]);
{
  "result" : [
    {
      "_id" : ObjectId("53213bc8ae5fcad63d0563f0"),
      "first" : "A",
      "second" : "M",
      "count" : 615
    },
    {
```

```
        "_id" : ObjectId("53213bc8ae5fcad63d0563f4"),
        "first" : "A",
        "second" : "F",
        "count" : 806
      },
      {
        "_id" : ObjectId("53213bc8ae5fcad63d056402"),
        "first" : "A",
        "second" : "Q",
        "count" : 377
      },
...more content omitted...
    {
        "_id" : ObjectId("53213bc9ae5fcad63d0567c5"),
        "first" : "A",
        "second" : "G",
        "count" : 769
      }
    ],
    "ok" : 1
  }
```

Other filtering options are `$limit` and `$skip`. Mostly used for testing as an initial filter, the `$limit` filter restricts the number of elements entering the aggregation, as in this example:

```
> db.aggtest.aggregate([{$limit:1}]);
{
  "result" : [
    {
      "_id" : ObjectId("53213bc8ae5fcad63d0563e9"),
      "first" : "S",
      "second" : "W",
      "count" : 762
    }
  ],
  "ok" : 1
}
```

The `$limit` command is more typically used after a grouping and sorting operation to limit the output to the user. Similarly, the `$skip` command will ignore some number of documents entering the filter. Combined with `$limit`, it is often used after grouping, as well as to implement pagination:

```
> db.aggtest.aggregate([{$skip:10},{$limit:1}]);
{
  "result" : [
    {
      "_id" : ObjectId("53213bc8ae5fcad63d0563f3"),
      "first" : "M",
      "second" : "E",
      "count" : 437
    }
  ],
  "ok" : 1
}
```

After filtering commands are applied in the pipeline, group management commands are applied. The most commonly used command is the `$group` operator, which specifies an identifier field and some number of accumulators. For example, to sum the “count” field for each of the values of the “first” field, the pipeline would be declared as follows:

```
> db.aggtest.aggregate([{$group:{_id:"$first",
  total:{$sum:"$count"}}}]);
{
  "result" : [
    {
      "_id" : "V",
      "total" : 18224
    },
    {
      "_id" : "Y",
      "total" : 15299
    },
    {
      "_id" : "D",
      "total" : 20929
    },
    {
      "_id" : "I",
      "total" : 13257
    },
    {
      "_id" : "N",
      "total" : 16601
    },
...other output omitted...
    {
      "_id" : "E",
      "total" : 21444
    }
  ],
  "ok" : 1
}
```

The previous example used the `$sum` accumulator, but there are a number of other accumulators that can be used including `$avg` to compute the average, `$min` to compute the minimum and `$max` to compute the maximum. The `_id` field can contain multiple values allowing for grouping on multiple fields, but the output of the grouping is always unsorted. To sort the output, a `$sort` operation can be applied to the output:

```
> db.aggtest.aggregate([{$group:{_id:"$first",
  total:{$sum:"$count"}}},{$sort:{total:-1}}]);
{
  "result" : [
    {
      "_id" : "H",
      "total" : 27990
    },
    {
      "_id" : "M",
      "total" : 27188
    },
    {
      "_id" : "L",
      "total" : 25070
    },
    {
      "_id" : "A",
      "total" : 24148
    },
    {
```

```
        "_id" : "O",
        "total" : 21865
      },
...other output omitted...
      {
        "_id" : "Q",
        "total" : 12831
      }
    ],
    "ok" : 1
}
```

This can, as mentioned, be combined with `$skip` and `$limit` commands. For example, to find the top three elements:

```
> db.aggtest.aggregate([{$group:{_id:"$first",
  total:{$sum:"$count"}}},{$sort:{total:-1}},{$limit:3}]);
{
  "result" : [
    {
      "_id" : "H",
      "total" : 27990
    },
    {
      "_id" : "M",
      "total" : 27188
    },
    {
      "_id" : "L",
      "total" : 25070
    }
  ],
  "ok" : 1
}
```

Replication

Replication is an important subject for MongoDB. In early versions of the database, the claim was that single-machine durability could not be trusted, and a lot of emphasis was placed on the need to use replicas to ensure data durability. Version 2.0 and later support journaling writes that improve single-machine durability, but multi-machine replication is still the preferred strategy for MongoDB durability.

To handle replication, MongoDB uses a system called a Replica Set. Like Kafka's in-sync replicas, this approach is still essentially a master-slave architecture, but the master is elected by a quorum of set members rather than explicitly assigned. The current master is called the Primary, and it keeps track of all of the other members—called Secondaries—via a heartbeat mechanism. All writes are conducted against the Primary, but reads can be conducted against any of the servers in the Replica Set.

NOTE

Replication in MongoDB is asynchronous. Reads from the Primary always return the most recent data, but a read from a Secondary may not have received the most recent data yet. To further complicate the situation, MongoDB's developers allow a replication delay to be explicitly set as well. This creates replicas that are intentionally lagging the Primary by a specified amount. This can be useful for certain types of disaster recovery scenarios. To prevent clients from inadvertently reading from these lagging replicas, MongoDB also includes the ability to hide replicas in the set, making them unavailable to clients.

In addition to maintaining heartbeat connections with the Primary, Secondaries maintain a heartbeat with each other. If contact with the Primary is lost for more than 10 seconds, the Secondaries elect a new Primary from among themselves, allowing for mostly automatic failover. This may result in a rollback of data that was written to the Primary before it lost contact with the rest of the cluster.

Sharding

In addition to replication, MongoDB also supports auto-balancing sharding for data being stored in collections (except capped collections). The implementation is quite similar to the approach used by `twemproxy` to cluster Redis, except that data may move between shards over time.

MongoDB's sharding implementation is designed to transition smoothly from an unsharded environment to a sharded one. The usual recommendation, when starting to use MongoDB, is to begin with an unsharded dataset in a normal Replica Set and then add shards as the dataset grows to the point that the working set starts to get close to available RAM on a single server.

The reason this works is because MongoDB shards are simply Replica Sets. To implement the sharding itself, MongoDB introduces an auxiliary server called `mongos`. In combination with a configuration server, which is simply another MongoDB Replica Set, the `mongos` server acts as an intermediary between applications and the database shards. It is responsible for distributing queries and collating results for return to a client. In this sense, `mongos` is very similar to the `twemproxy` server used to shard Redis and Memcached instances. A sharded MongoDB cluster may have any number of `mongos` servers running at any given time. A common strategy is to run a `mongos` process on each application server to simplify application configuration.

The `mongos` server differs from `twemproxy` in that it is also responsible for cluster rebalancing. Unlike many sharding approaches, which rely on a predetermined shard key, MongoDB attempts to automatically balance data and load across the cluster. This is handled by a `balancer` process, initiated by one of the `mongos` servers attached to the cluster. Normally this process is automatic and started when `mongos` detects sufficient imbalance in cluster resources. However, because rebalancing the cluster places a load on the system, it is possible to configure the balancer to only run during specific time windows when load is expected to be low (for example, early morning hours for most Internet applications).

To start using sharding in MongoDB, first a set of configuration servers must be established. These configuration servers play the same role as ZooKeeper plays for Kafka, providing configuration metadata. Most importantly, it will contain information about how keys are distributed among shards. Rather than use a separate application like ZooKeeper, MongoDB uses specially configured MongoDB databases to act as configuration managers. This is activated by adding `--configsvr` to `mongod`'s command-line arguments:

```
$ mkdir configdb
$ ./mongod --configsvr --dbpath ./configdb
...
[initandlisten] command local.$cmd command: {
  create: "startup_log",
  size: 10485760,
  capped: true
} ntoreturn:1 keyUpdates:0  reslen:37 168ms
[initandlisten] ******
[initandlisten] creating replication oplog of size: 5MB...
[initandlisten] ******
[websvr] admin web console waiting for connections on port 28019
[initandlisten] waiting for connections on port 27019
```

Notice that the configuration server starts itself on a different port than the normal MongoDB server. This makes it possible to run a normal shard server and a configuration server on the same machine if desired. These servers are lightly loaded, so having them share resources with another server is usually not a problem.

If attempting to run a sharded server on a development machine, now is the time to stop any running `mongod` processes. The `mongos` server runs on the same port as `mongod` and, because clients will be attaching to `mongos`, it is usually easier to change the `mongod` port(s). To test mongos on a single-machine, after starting the configuration server, start the `mongod` server on a different port:

```
$ ./mongod  --dbpath ./db/ --port 27018
 [initandlisten] MongoDB starting : pid=9653 port=27018 dbpath=./db/
   64-bit
 [initandlisten]
 [initandlisten] ** WARNING: soft rlimits too low.
  Number of files is 256, should be at least 1000
 [initandlisten] db version v2.4.8
 [initandlisten] git version: a350fc38922fbda2cec8d5dd842237b904eafc14
 [initandlisten] build info: Darwin bs-osx-106-x86-64-2.10gen.cc 10.8.0
  Darwin Kernel Version 10.8.0: Tue Jun  7 16:32:41 PDT 2011;
  root:xnu-1504.15.3~1/RELEASE_X86_64 x86_64
  BOOST_LIB_VERSION=1_49
 [initandlisten] allocator: system
 [initandlisten] options: { dbpath: "./db/", port: 27018 }
 [initandlisten] journal dir=./db/journal
 [initandlisten] recover : no journal files present, no recovery needed
 [websvr] admin web console waiting for connections on port 28018
 [initandlisten] waiting for connections on port 27018
```

The `mongos` server can then be started on the normal port, pointing to the configuration server running on the same machine:

```
$ ./mongos --configdb localhost
 [mongosMain] MongoS version 2.4.8 starting: pid=9657 port=27017 64-bit
  (--help for usage)
 [mongosMain] git version: a350fc38922fbda2cec8d5dd842237b904eafc14
 [mongosMain] build info: Darwin bs-osx-106-x86-64-2.10gen.cc 10.8.0
  Darwin Kernel Version 10.8.0: Tue Jun  7 16:32:41 PDT 2011;
  root:xnu-1504.15.3~1/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
 [mongosMain] options: { configdb: "localhost" }
 [mongosMain] starting upgrade of config server from v0 to v4
 [mongosMain] starting next upgrade step from v0 to v4
 [mongosMain] writing initial config version at v4
  Mon Dec  2 20:07:18.120
 [mongosMain] upgrade of config server to v4 successful
```

```
[Balancer] about to contact config servers and shards
[websvr] admin web console waiting for connections on port 28017
[Balancer] config servers and shards contacted successfully
[mongosMain] waiting for connections on port 27017
```

On production systems, there should be at least three configuration servers. The `mongos` server is then provided with a comma-separated list of at least some of those servers (more than one to provide failover). A single configuration server should only be used for testing purposes.

Once the `mongos` server is running, the mongo client can be used to configure the cluster itself. The `sh.addShard` function is used to add shards to a cluster (this command is only available when attached to a `mongos` server). If the server is a standalone server, only the hostname is needed:

```
mongos> sh.addShard("localhost:27018")
{ "shardAdded" : "shard0000", "ok" : 1 }
```

To add a replica set, simply add one of the servers belonging to the replica set (older versions of MongoDB might require that all servers be added as a comma separated list):

```
mongos> sh.addShard("replset1/localhost:27018")
{ "shardAdded" : "shard0000", "ok" : 1 }
```

Finally, sharding is activated for a particular database:

```
mongos> sh.enableSharding("mine");
{ "ok" : 1 }
```

There are more fine-grained options for controlling the sharding of specific collections within a database, but this is the basic use case. With sharding enabled, MongoDB can now support databases with a working set much larger than possible on a single-machine.

Cassandra

Cassandra is a database system that implements features similar to Amazon's DynamoDB (one of Cassandra's original authors worked on the DynamoDB project) and Google's BigTable.

Unfortunately, with an implementation similar to BigTable came a “query language” also similar to BigTable. Based on the internal Thrift data structures, it was cumbersome to use and unfamiliar to most developers. This led to a reputation of Cassandra being difficult to use and hard to maintain and a “dead end.”

This reputation was furthered when Facebook, the original developers of Cassandra, “abandoned” it in favor of HBase when implementing the Facebook instant messaging feature. Of course, this had more to do with a consistency model than anything else—an instant messaging application needs to be strongly consistent. The user should not “lose” messages if their application happens to use a server in the cluster that is in a state inconsistent with the last server they used on the last call. Cassandra's data model is eventual consistency, making it a poor choice for this application. HBase is strongly consistent, which has its own issues, but makes it much more appropriate for that sort of application. Most real-time analytics applications can tolerate eventual consistency because the data is a flow, and the changing set of consistent servers acts more like an application delay than anything else.

The introduction of the Cassandra Query Language (CQL) has done a lot to ease the pain of working with Cassandra. With CQL 3.0, essentially all of the original BigTable-like structure has been abstracted into an SQL-like language that should be familiar to most developers. There are some limitations to the CQL language, limited aggregation functions among other things, but the Cassandra

design is such that it favors extensive denormalization of tables. This increases the storage overhead and introduces a write-time aggregation cost, but this is not as much of a concern in a scale-out architecture like Cassandra.

This section introduces Cassandra's architecture and data model. Cassandra is similar to other DynamoDB- and BigTable-inspired databases like HBase and Voldemort, and many of the concepts introduced in this section are applicable to those other technologies. The internal data model used by Cassandra is also covered, primarily for historical reasons, because CQL abstracts much of these internals with its own somewhat incompatible data model.

Server Architecture

The Cassandra server architecture is a "master-less" cluster of nodes, the goal being to eliminate single points of failure and allow linear scaling. To achieve this, Cassandra uses a distributed hash table approach to data management. In this model, each row of data is assigned a partition key. This key defines which server stores a particular piece of information. To improve data durability, Cassandra can also replicate this data to other nodes through consistent hashing.

Early versions (prior to 1.2) used a server-based consistent hashing technique that required a fair bit of maintenance to calculate and assign tokens to each node to tell it which parts of the hash range each node would store. Newer versions of Cassandra introduced the concept of the virtual node, called a "vnode," which breaks the hash range into much smaller pieces that are then randomly distributed among the nodes.

Each of these virtual nodes maintains its own indexes for the data contained within that partition as well as a specialized data structure called a Bloom Filter (discussed in detail in Chapter 10, "Approximating Streaming Data with Sketching") that helps to quickly determine if a query needs any data from a particular virtual node.

A node in a cluster maintains a map of other servers through a peer-to-peer communication protocol. Called the "gossip" protocol, each server exchanges state information with up to three other servers in the cluster roughly once every second. This information also contains state information for other servers, so a given node will quickly learn the entire cluster topology as its gossips with its peers.

In addition to the gossip protocol, the "snitch" protocol determines the local network topology. This helps Cassandra optimize its request routing and discover nodes that are in an active, but degraded, state. In practice, "sick" nodes arise for a variety of reasons and are fairly common as a result. There are different kinds of snitches optimized to various deployments. For example, clusters running in Amazon's EC2 have two snitches: `EC2Snitch` and `EC2MultiRegionSnitch`. The former is designed to operate well in a situation where the cluster is spread across multiple availability zones in a single region (the most common case), whereas the latter is designed to optimize across regions.

Because each node has, through gossip and snitch, a complete understanding of the cluster topology, any node can act as a query server. When a client connects to the cluster it can chose any node it likes, which then becomes the coordinator for that client until the connection is closed. This would be roughly equivalent to every MongoDB server hosting `mongod` in addition to `mongos`.

Setting Up a Cluster

Setting up a Cassandra cluster is relatively easy from a software perspective because each server is essentially identical. For good performance, the recommendation is to use multi-core machines with

8GB to 16GB of RAM. Very large heaps in Cassandra actually result in reduced performance due to the need to perform garbage collection. The usual recommendation is somewhere around an 8GB heap (depending on available RAM). In Amazon EC2, this corresponds to Large or Extra Large instances.

A Cassandra server should have as fast a disk as possible. If available, Solid State Drives (SSDs) are a good match to Cassandra's access pattern. If SSDs are not available, a group of disks merged via RAID0 is also a good choice. Cassandra, like Kafka and HDFS, has the ability to use several disks in a JBOD (Just a Bunch of Disks) configuration, but RAID0 can achieve higher performance. Using a RAID0 configuration also eliminates imbalances in the data distribution between drives.

Network attached storage (NAS) is not recommended for Cassandra installations, except as a backup medium. Network storage systems such as NFS or Elastic Block Storage (EBS) contend for network I/O resources, resulting in degraded performance.

Setting up the server itself is straightforward. Installation instructions for Debian are available at the Apache Cassandra website (http://cassandra.apache.org). Datastax, a commercial provider of Cassandra and active committer to the code, also provides distributions for a number of platforms including Windows and OS X. For users wanting to "roll their own" distribution, a binary tarball is also available from the Apache website. Most modern Unix-like operating systems should be able to use this tarball directly, assuming Java has been properly installed.

For Cassandra, an appropriate version of Java means Java 7 or higher. Trying to start Cassandra with an older Java 6 installation, which is still very common, results in errors like this:

```
$ java -version
java version "1.6.0_65"
Java(TM) SE Runtime Environment (build 1.6.0_65-b14-462-11M4609)
$ ./bin/cassandra -f
xss =  -ea -javaagent:./bin/../lib/jamm-0.2.5.jar
  -XX:+UseThreadPriorities
  -XX:ThreadPriorityPolicy=42
  -Xms1024M -Xmx1024M
  -Xmn256M -XX:+HeapDumpOnOutOfMemoryError -Xss256k
Exception in thread "main"
  java.lang.UnsupportedClassVersionError
  org/apache/cassandra/service/CassandraDaemon :
  Unsupported major.minor version 51.0
```

A major version of 51 refers to Java 7 (Java 6 is 50; Java 8 is 52). After Java 7 or higher is installed, Cassandra can be started with either `cassandra` or `cassandra -f`, which runs the server in the foreground.

Configuration Options

Most configuration options for Cassandra 2.0 are reasonable "out of the box," but there are a few that are often changed to conform to a particular installation. These options can be found in `conf/` directory. The `cassandra.yaml` specifies the cluster name, which defaults to "Test Cluster":

```
# The name of the cluster. This is mainly used to prevent machines in
# one logical cluster from joining another.
cluster_name: 'Test Cluster'
```

By default, it will write to `/var/lib/cassandra`, so that directory must be created with appropriate permissions. To change this, there are three sections of the `cassandra.yaml` file that

specify the data output directories:

```
# Directories where Cassandra should store data on disk.  Cassandra
# will spread data evenly across them, subject to the granularity of
# the configured compaction strategy.
data_file_directories:
    - /var/lib/cassandra/data
# commit log
commitlog_directory: /var/lib/cassandra/commitlog
# saved caches
saved_caches_directory: /var/lib/cassandra/saved_caches
```

For multi-node clusters, `listen_address` and `rpc_address` should be set. The first parameter, `listen_address`, is the network interface the node will use for gossip while the latter is used for client connections. Leaving both blank usually works, but sometimes it is necessary to specify a specific network address to use. By default, `listen_address` is set to "localhost," which usually doesn't work:

```
# Address to bind to and tell other Cassandra nodes to connect to. You
# _must_ change this if you want multiple nodes to be able to
# communicate!
#
# Leaving it blank leaves it up to InetAddress.getLocalHost(). This
# will always do the Right Thing _if_ the node is properly configured
# (hostname, name resolution, etc), and the Right Thing is to use the
# address associated with the hostname (it might not be).
#
# Setting this to 0.0.0.0 is always wrong.
listen_address:
# The address to bind the Thrift RPC service and native transport
# server -- clients connect here.
#
# Leaving this blank has the same effect it does for ListenAddress,
# (i.e. it will be based on the configured hostname of the node).
#
# Note that unlike ListenAddress above, it is allowed to specify 0.0.0.0
# here if you want to listen on all interfaces, but that will break
# clients that rely on node auto-discovery.
rpc_address:
```

Finally, a multi-node cluster server needs a list of seed servers it can use to bootstrap itself into the cluster. This list does not need to include every server in the cluster, but enough to ensure that at least one server is available when the server starts. Simply modify `seeds` to be a comma-delimited list of servers:

```
# any class that implements the SeedProvider interface and has a
# constructor that takes a Map<String, String> of parameters will do.
seed_provider:
    # Addresses of hosts that are deemed contact points.
    # Cassandra nodes use this list of hosts to find each other and
    # learn the topology of the ring.  You must change this if you are
    # running
    # multiple nodes!
    - class_name: org.apache.cassandra.locator.SimpleSeedProvider
      parameters:
          # seeds is actually a comma-delimited list of addresses.
          # Ex: "<ip1>,<ip2>,<ip3>"
          - seeds: "127.0.0.1"
```

CQL: The Cassandra Query Language

CQL is the recommended mechanism for development of all new Cassandra applications. It implements a subset of the Structured Query Language (SQL), making it familiar to many developers. However, it does not implement many of SQL's features and the structure of Cassandra tables may be a surprise to many developers.

CQL commands can be executed through three different mechanisms. The first is the CQL shell, `cqlsh`, which is a Python application shipped with Cassandra. This is different than `cassandra-cli`, which uses the Thrift API. It can be found in the same directory as the Cassandra server binary. The second is through a "native" CQL driver. Newer versions of Cassandra implement a protocol that works only with CQL statements, similar to protocols used by relational databases. Finally, CQL commands can be executed using older clients that employ Cassandra's Thrift protocol using the `execute_cql3_query` remote procedure call (RPC) command.

Keyspaces and Column Families

All Cassandra data is held in a keyspace. This is a form of namespace, roughly equivalent to a database in MongoDB or a schema in a relational database. When keyspaces are created, they define a replication strategy as well as a replication factor. The replication factor is simply the number of copies of any given piece of data kept in the cluster. The strategy defines how these replicas are distributed. To get started, create a `metrics` keyspace with a replication factor of 1 using `cqlsh`:

```
cqlsh> CREATE KEYSPACE metrics WITH REPLICATION =
  {'class':'SimpleStrategy','replication_factor':1};
```

These settings can be changed later using the `ALTER KEYSPACE` command. In particular, if the cluster is to be used in a production environment with multiple nodes, the replication strategy defined by `class` should be the `NetworkTopologyStrategy`. This allows for the use of multiple datacenters while the `SimpleStrategy` used in this example is confined to a single datacenter. Even if a single physical datacenter is used, it is common practice to define "datacenters" of clusters that maintain different workloads. For example, one "datacenter" can be dedicated to accepting real-time random access reads and writes, whereas another "datacenter" is dedicated to scan-heavy map-reduce style processing. This is often done with Hadoop because Cassandra can act as a replacement for Hadoop's native HDFS storage layer.

To create a table, switch to the `metrics` keyspace via the `USE` command and then execute `CREATE TABLE` to store some time-series metrics:

```
cqlsh> USE metrics;
cqlsh:metrics> CREATE TABLE counts (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  PRIMARY KEY (customer_id,metric,ts)
);
```

It is not necessary to select the keyspace; dot-notation can be used to access the table directory. The table from the last example would be accessed as `metrics.counts`.

This table contains a customer identifier, a metric name and a timestamp. All three of them form the primary key because there can only be a single count for the combination of customer, metric, and

time. In Cassandra, columns are serialized and deserialized to and from byte arrays corresponding to their types. These types, which are similar to those found in a relational database, are summarized in Table 6.2.

Table 6.2 Table 6-2: CQL Data Types

Name	Description
`ASCII`	A string encoded as US-ASCII (characters 0–127)
`BIGINT`	A 64-bit signed long value (equivalent to a Java long)
`BLOB`	An arbitrary byte array
`BOOLEAN`	A `true` or `false` Boolean value
`COUNTER`	A 64-bit distributed counter value. This gets special handling in Cassandra
`DECIMAL`	A variable precision decimal value
`DOUBLE`	64-bit IEEE-754 compliant floating-point values
`FLOAT`	32-bit IEEE-754 compliant floating-point values
`INET`	A structure representing IPv4 or IPv6 addresses
`INT`	A 32-bit signed integer (equivalent to a Java `int`)
`LIST<?>`	A list of values where the `?` is a valid CQL type. For example, `LIST<TEXT>` would be a list of UTF-8 strings. Collections may not contain other collections so `LIST<LIST<TEXT>>` is not allowed
`MAP<?,?>`	An associative array of CQL types. Like the `LIST` type, `MAP` takes two valid scalar CQL types
`SET<?>`	The `SET` collection type is similar to the `LIST` type, except that it only stores unique values
`TEXT`	A string encoded as UTF-8
`TIMESTAMP`	A timestamp encoded as a 64-bit integer (`BIGINT`) in milliseconds since the UNIX epoch
`UUID`	A universally unique identifier value. For Windows users, this is the same as a GUID
`TIMEUUID`	A Type 1 UUID with a timestamp component. This type allows multiple clients to perform inserts with the timestamp but without collisions. Time range queries can be performed on columns of this type. When performing inserts the `NOW()` function is used to generate a `TIMEUUID`
`VARCHAR`	A UTF-8 encoded string
`VARINT`	An arbitrary precision integer value

Source:
http://www.datastax.com/documentation/cql/3.1/webhelp/index.html#cql/cql_reference/cql_data_type
wbk_zdt_xj

Counter Columns

In Cassandra, atomic counter columns cannot be combined with other nonkey columns in the column factory. A counter column may be combined with other counter columns. Defining the original table without metric in the primary key results in an error:

```
cqlsh:metrics> CREATE TABLE counts (
  customer_id INT PRIMARY KEY,
  metric TEXT,ts TIMESTAMP,
  value COUNTER
);
Bad Request: Cannot add a counter column (value) in
  a non counter column family
```

Defining a second counter on the original table is allowed:

```
cqlsh:metrics> CREATE TABLE counts_2 (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  value_2 COUNTER,
  PRIMARY KEY (customer_id,metric,ts));
```

If for some reason counter columns and noncounter columns are needed, it is probably easiest to simply define two tables and update both simultaneously using a batch update statement.

Internally, Cassandra represents this table as a data structure known as a column family. Despite appearing to contain a large number of rows, the column family actually only contains a row for each `customer_id` and a large number of columns for each `metric` and `ts` combination. This is really only important to note because there is a limit on the number of columns a given row can have: 2 billion columns or 2 gigabytes of storage.

These limitations can be reached quite easily in some time-series implementations. To overcome them, Cassandra allows multiple keys to be used as the row identifier. The disadvantage to doing this is that the row key is also used to partition the data across the Cassandra cluster. This means that all queries, inserts, or updates must contain all of the elements of the row key.

If the query will always include the `customer_id` and the `metric`, merging the `customer_id` and `metric` fields would create rows identified by `customer_id:metric` combinations with a column for each timestamp:

```
cqlsh:metrics> CREATE TABLE counts_composite (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  value_2 COUNTER,
  PRIMARY KEY ( (customer_id,metric) ,ts)
) WITH CLUSTERING ORDER BY (ts DESC);
```

Adding the `CLUSTERING ORDER` command tells Cassandra to sort each of the columns in descending order instead of the natural order for a timestamp column, which would be ascending.

Like most relational databases, you can alter tables after they've been created using the `ALTER TABLE` command. The most common use case is to add a column to an existing table or to remove an existing column. Adding a new column does not cause any validation of existing rows.

Dropping a column will also eventually cause the deletion of the data associated with that column, but this does not happen until a major compaction occurs.

In some cases, the type of a column can be changed, but this does not modify the bytes used to store the data for existing columns. If the data cannot be deserialized by the new column type then errors result when querying the data. For instance, converting a `TEXT` column to an `INT` column will probably not have the desired effect.

Inserting and Updating Data

Inserting and updating data in Cassandra is accomplished through the familiar `INSERT` and `UPDATE` CQL statements. These two statements generally work like their SQL counterparts, but there are some differences.

`INSERT` statements are composed in the same way as they are in SQL with the caveat that all columns to be updated must be included in the statement (this is good practice in any case). For example, inserting gauge-style values into a time-series table:

```
cqlsh:metrics> CREATE TABLE stats(
  customer_id INT,
  metric TEXT,
  ts TIMEUUID,
  value INT,
  PRIMARY KEY( (customer_id,metric), ts)
) WITH CLUSTERING ORDER BY (ts DESC);
cqlsh:metrics> INSERT INTO
  stats(customer_id,metric,ts,value)
VALUES
  (1,'hits',NOW(),10);
cqlsh:metrics> INSERT INTO
  stats(customer_id,metric,ts,value)
VALUES (
  1,'hits',NOW(),15);
cqlsh:metrics> INSERT INTO
  stats(customer_id,metric,ts,value)
VALUES
  (1,'hits',NOW(),30);
cqlsh:metrics> SELECT * FROM stats;
customer_id | metric | ts                                   | value
------------+--------+--------------------------------------+------
          1 |   hits | 1ae93880-5fb8-11e3-9b3a-a1e3c690259e |    30
          1 |   hits | 19925b10-5fb8-11e3-9b3a-a1e3c690259e |    15
          1 |   hits | 175897b0-5fb8-11e3-9b3a-a1e3c690259e |    10
```

By default, `INSERT` statements overwrite data with the same primary key. This is not the usual behavior for SQL databases, so it may come as a surprise to some. This can cause problems in some situations, and, starting with CQL3, Cassandra now supports a feature called Lightweight Transactions that recover the SQL-like behavior by adding an `IF NOT EXISTS` statement to the end of the `INSERT` statement. For example, a users table with a `created_on` field set to `NOW()` will be changed after every insert unless the `IF NOT EXISTS` statement is used:

```
cqlsh:metrics> CREATE TABLE users(email TEXT PRIMARY KEY,created_on
  TIMEUUID);
cqlsh:metrics> INSERT INTO users(email,created_on)
           ... VALUES ('byron@domain',NOW());
cqlsh:metrics> SELECT email,dateOf(created_on) AS created_on
           ... FROM users;
```

```
 email        | created_on
--------------+--------------------------
 byron@domain | 2013-12-07 19:40:24-0800
(1 rows)
cqlsh:metrics> INSERT INTO users(email,created_on)
           ... VALUES ('byron@domain',NOW());
cqlsh:metrics> SELECT email,dateOf(created_on) AS created_on
           ... FROM users;
 email        | created_on
--------------+--------------------------
 byron@domain | 2013-12-07 19:41:09-0800
(1 rows)
cqlsh:metrics> INSERT INTO users(email,created_on)
           ... VALUES ('byron@domain',NOW())
           ... IF NOT EXISTS;
 [applied] | email        | created_on
-----------+--------------+--------------------------------------
     False | byron@domain | 9351e360-5fba-11e3-9b3a-a1e3c690259e
cqlsh:metrics> SELECT email,dateOf(created_on) AS created_on
           ... FROM users;
 email        | created_on
--------------+--------------------------
 byron@domain | 2013-12-07 19:41:09-0800
(1 rows)
```

Updates also work mostly like their SQL counterparts, except that they perform an `upsert` by default. Much like `INSERT` overwrites data instead of failing, `UPDATE` performs an implicit insert if the primary key does not exist in the dataset. This is actually quite useful for `COUNTER` columns, which cannot be created using `INSERT` statements:

```
cqlsh:metrics> DROP TABLE counts;
cqlsh:metrics> CREATE TABLE counts(
  customer_id INT,
  metric TEXT,
  value COUNTER,
  PRIMARY KEY (customer_id, metric)
);
cqlsh:metrics> UPDATE counts SET
  value = value + 1
WHERE customer_id = 1 AND metric = 'test';
cqlsh:metrics> SELECT * FROM counts;
 customer_id | metric | value
-------------+--------+-------
           1 |   test |     1
(1 rows)
```

Update statements also support a check-and-set operation by appending an `IF` statement with the same form as a `WHERE` clause to the end of the statement. The update only succeeds if the `IF` statement evaluates to `true`.

Both `INSERT` and `UPDATE` statements also support expiring data using a time-to-live statement. Simply append `USING TTL <seconds>` to the end of an `INSERT` statement. For `UPDATE` statements, you need to put it after the table name: `UPDATE <table> USING TTL <seconds>`.

To improve performance, `insert` and `update` statements can be wrapped in a `BATCH` statement. This vastly improves performance when writing to a number of different tables or processing a large number of requests. To use a `BATCH` statement, simply start a command with `BEGIN BATCH`. Write

statements as you normally would, and when you're ready to submit, add an `APPLY BATCH` to the end of the statement.

Metrics with Cassandra

As shown in the examples in this section, Cassandra is highly amenable to capturing metric data using atomic counters. The one thing it lacks is the ability to perform any form of server-side aggregation. In this case, the recommended strategy is to produce all aggregates at write time using batch updates.

To produce aggregates at both the customer level and at the system level, use two tables for aggregation:

```
cqlsh:metrics> CREATE TABLE customer_counts (
  customer_id INT,
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER,
  PRIMARY KEY ( (customer_id,metric) , ts) )
WITH CLUSTERING ORDER BY (ts DESC);
cqlsh:metrics> CREATE TABLE system_counts (
  metric TEXT,
  ts TIMESTAMP,
  value COUNTER, PRIMARY KEY ( metric , ts) )
WITH CLUSTERING ORDER BY (ts DESC);
```

To update each table, a `BATCH` command is used. This is not a transaction, but it does allow Cassandra to perform an efficient update:

```
cqlsh:metrics> BEGIN COUNTER BATCH
           ... UPDATE customer_counts
               SET value = value + 1
               WHERE customer_id = 1
               AND metric = 'impressions'
               AND ts = '2013-12-01T00:00:00';
           ... UPDATE system_counts
               SET value = value + 1
               WHERE metric = 'impressions'
               AND ts = '2013-12-01T00:00:00';
           ... APPLY BATCH;
```

Reading from Cassandra

Like inserting data, reading data from Cassandra uses SQL-like `SELECT` statements. Cassandra does not have many query options, so `SELECT` statements tend to be fairly simple.

Other Storage Technologies

This chapter has focused on a few different persistent stores popular for real-time applications. They were chosen to show some of the range of available stores beyond the relational database. This section briefly mentions some of the other technologies available, but not discussed in this book.

Relational Databases

Many relational database implementations are backed by a high-performance data store. The performance hit introduced by transactions are maintained at the application level rather than being intrinsic to the database itself. Some relational databases even allow access to these high-performance stores. For example, PostgreSQL exposes a key-value store through its `hstore` module, which allows access through the SQL interface. It also allows for some indexing of the key-value store.

Other databases, especially commercial column stores, are designed to ingest large amounts of raw data that can then be efficiently queried. In cases where the access patterns are not well defined, it might be best to stream data into a column store for access. The downside is usually related to the cost required to achieve acceptable ingest performance, but this is often the case when balancing query flexibility and performance.

Distributed In-Memory Data Grids

Data grids have been available in various forms for a number of years. They are mostly marketed as commercial products from well-known vendors such as VMware and Oracle. To increase adoption, many of them, now have a Community Edition that can be used without charge for evaluation purposes. One such product is Hazelcast.

They are primarily designed to provide a distributed view of data that behaves as if it was simply being held in the application's main memory. This tends to be programming language specific, with Java being a favorite, and implements distributed versions of the native collections such as `List` and `Map` interfaces for Java.

Choosing a Technology

When persistence technologies are discussed there are inevitably arguments over which technology is best. Like arguments about which text editor or integrated development environment is "best," these arguments are entertaining but ultimately meaningless. There are really only technologies that work well in a given environment for a given application.

Developers experienced with a specific environment may be able to successfully generalize that experience to other applications, which may make it the best choice simply because it offers the shortest time to market even though another technology's characteristics would have been an objectively better match for the application. Drawn from experiences good and bad, this section provides some rules of thumb when considering a particular technology.

Key-Value Stores

Key-value persistence approaches, like Redis, work best when the aggregation structure is well known in advance. They also generally require that the queries are fairly simple because this is managed entirely by the client side. This makes them most appropriate for applications with limited interaction, such as dashboard applications. Building a simple dashboard with Redis is discussed in Chapter 7.

Document Stores

By design, document stores excel when there are a large variety of metrics to store with a natural grouping. For example, a customer might have a large number of customizable metrics that are specific to his business needs.

They also work well when the query pattern is such that pulling the document into the working set essentially "warms" it up, providing faster access for subsequent queries.

Distributed Hash Table Stores

Stores like Cassandra, Voldemort, BigTable, and DynamoDB are a bit of a mix between a relational database and a key-value store. Like key-value stores, they work best when the patterns of aggregation are well known so that they can be denormalized by the processing infrastructure. The batch update feature of Cassandra exists essentially to encourage this sort of multitable update.

Unlike key-value stores, they usually provide efficient range or set querying. They also usually support secondary indexing to improve query performance. This can be arranged in key-value stores like Redis using features like sorted sets, but it becomes inconvenient when the queries get complicated.

Unlike relational databases, these types of stores usually do not support *ad hoc* aggregation very well. They do not typically support `GROUP BY` and `SUM` operations, requiring client-side aggregation. This makes them a poor choice for more exploratory data environments unless their integration with map-reduce environments will be used to perform the aggregation.

In-Memory Grids

These tend to work well in instances where distributed hash table or key-value stores work well. Due to the fact that all data is held in-memory, accessing and updating the data is usually very fast. However, it also makes them more expensive than disk-based distributed hash tables (DHTs). Most

provide some querying and indexing capabilities, making them similar to DHT approaches.

They also work well when the storage is tightly coupled to the application. For example, a legacy application that assumes a scale-up architecture being adapted to a scale-out architecture that can be fed from a real-time streaming system.

Relational Databases

While NoSQL stores have been in vogue, it has been claimed that relational databases are somehow threatened. This is simply untrue; relational databases are alive and well and very good at what they do. They excel at ad hoc aggregation and querying. This is especially true of column-oriented databases, which have specialized indexing and storage mechanisms to optimize many typical aggregation queries.

One thing they usually don't do well is very high-speed ingest. “Real-time” in the relational world usually refers to a minutes or hours delay in processing a data stream rather than milliseconds. (There are exceptions; several companies offer databases that specialize in high-performance ingest.) It usually works best to manage them through a slower ETL process, which will be discussed shortly.

Warehousing

Many businesses already have an existing business intelligence infrastructure. This usually consists of a relational database environment loaded by some sort of extract-transform-load (ETL) tool.

At large organizations this is typically managed by a database team using a set of formal processing tools. At smaller organizations this may be more *ad hoc* with a small MySQL database loaded by some scripts that nobody remembers writing. In both cases, there is a bit of a problem. The velocity of streaming data overwhelms the ETL infrastructure or the database due to the constant updates from the real-time processing system. Furthermore, when (not if) bugs are introduced, the ability to recover from the error and reprocess data helps improve operational efficiency and uptime. With the introduction of modern tools for handling this processing, the warehousing process can be more consistently implemented and scale more readily even at smaller organizations that cannot invest in a complicated ETL infrastructure.

Hadoop as ETL and Warehouse

Since its public introduction in 2007, Hadoop has become an almost *de facto* standard for the development of large-scale processing and ETL tasks. It has accomplished this feat in spite of a fairly limited processing model due to the fact that it essentially solved the fundamental management problem of large scale batch environments: binding computation to the data.

In large-scale ETL pipelines, the individual operations to be performed on the data are usually fairly trivial. In database terms, they can usually be boiled down to a sequence of `GROUP BY` statements in the `WHERE` clause with `SUM` and perhaps `DISTINCT` statements in the `SELECT` clause. The larger issue is managing the flow of data around a cluster of machines in a way that allows for efficient processing.

Ingesting Data from Kafka

Kafka comes with a very simple ingestion mechanism for Hadoop. Unfortunately, it's a bit too simple to use in a production environment with any confidence. A better choice is the Camus tool, which LinkedIn uses for its own Kafka-to-Hadoop ingestion. Although it's currently used in production, Camus's open source lifecycle is still fairly new, so there is no prepackage library available at the time of writing.

By default, Camus assumes that the data to be processed looks a lot like LinkedIn's own internal data format, which is Avro based. Most people do not work at LinkedIn, so this typically requires building a custom importer that understands what to do with the data.

To get started building a custom importer, first check out the Camus repository from Github so the required dependencies can be installed into a local Maven repository. This book uses Kafka 0.8, which has not yet been merged to the master branch:

```
$ git clone https://github.com/linkedin/camus
Cloning into 'camus'...
remote: Counting objects: 2539, done.
remote: Compressing objects: 100% (991/991), done.
remote: Total 2539 (delta 774), reused 2393 (delta 661)
Receiving objects: 100% (2539/2539), 37.99 MiB | 950.00 KiB/s, done.
Resolving deltas: 100% (774/774), done.
Checking connectivity... done
$ cd camus/
```

```
$ git checkout camus-kafka-0.8
Branch camus-kafka-0.8 set up to track remote branch camus-kafka-0.8
from origin.
Switched to a new branch 'camus-kafka-0.8'
$ mvn install
```

After the Maven packages have been locally installed, they can be used in a custom loader project's Maven file:

```
<dependencies>
    <dependency>
      <groupId>com.linkedin.camus</groupId>
      <artifactId>camus-api</artifactId>
      <version>0.1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>com.linkedin.camus</groupId>
      <artifactId>camus-etl-kafka</artifactId>
      <version>0.1.0-SNAPSHOT</version>
    </dependency>
    <dependency>
      <groupId>org.apache.hadoop</groupId>
      <artifactId>hadoop-core</artifactId>
      <version>1.0.3</version>
    </dependency>
 </dependencies>
```

A file called `camus.properties` that is located in the `jar` file for the custom loader controls the Camus loading process. Either copy this file from the `camus-example` project or use the version included in this book's project source code. If the file from `camus-example` is used, some entries need to be modified to use the custom loader.

In this case, the data is assumed to be JSON formatted. A timestamp for the event is assumed to be in the `ts` field as a number of milliseconds since the Unix epoch. This is extracted and the parsed JSON returned along with the timestamp in a `CamusWrapper`, which is the class Camus uses to move data around:

```
public class JSONMessageDecoder
  extends MessageDecoder<byte[], ObjectNode> {
  private final ObjectMapper mapper = new ObjectMapper();
  @Override
  public CamusWrapper<ObjectNode> decode(byte[] arg0) {
    ObjectNode json;
    try {
      json = mapper.readValue(new String(arg0), ObjectNode.class);
    } catch (IOException e) {
      throw new MessageDecoderException("Unable to parse event");
    }
    return new CamusWrapper<ObjectNode>(json,
     json.get("ts").getValueAsLong());
  }
}
```

To use this class, ensure that the `camus.message.decoder.class` property is set properly:

```
# Concrete implementation of the Decoder class to use
camus.message.decoder.class=wiley.streaming.camus.JSONMessageDecoder
```

Now that the data has been safely decoded it must be written out to files. In this case, a

`SequenceFile` will be used. Because these files offer space for both a key and a value, the key will contain information about the event's position in Kafka: the topic, partition, and offset. The value will be the JSON value re-encoded to text for further processing:

```
public class JSONRecordWriter implements RecordWriterProvider {
  @Override
  public RecordWriter<IEtlKey, CamusWrapper> getDataRecordWriter(
      TaskAttemptContext context, String filename, CamusWrapper arg2,
      FileOutputCommitter comitter) throws IOException,
        InterruptedException {
    Configuration conf = context.getConfiguration();
    Path file = new Path(comitter.getWorkPath(),
      EtlMultiOutputFormat.getUniqueFile(context, filename,
       getFilenameExtension()));
    CompressionCodec codec = null;
    SequenceFile.CompressionType type =
      SequenceFile.CompressionType.NONE;
    if(FileOutputFormat.getCompressOutput(context)) {
      type = SequenceFileOutputFormat.getOutputCompressionType(         context
);
      Class<?> klass =
  SequenceFileOutputFormat.getOutputCompressorClass(context,
   DefaultCodec.class);
      codec = (CompressionCodec) ReflectionUtils.newInstance(klass,
        conf);
    }
    final SequenceFile.Writer writer = SequenceFile.createWriter(
        file.getFileSystem(conf),
        conf,
        file,
        Text.class, Text.class,
        type,
        codec);
    return new RecordWriter<IEtlKey,CamusWrapper>() {
      Text key   = new Text();
      Text value = new Text();
      @Override
      public void close(TaskAttemptContext arg0) throws IOException,
          InterruptedException {
        writer.close();
      }
      @Override
      public void write(IEtlKey arg0, CamusWrapper arg1)
          throws IOException, InterruptedException {
        key.set(arg0.getTopic()+"\t"+arg0.getPartition()+
          "\t"+arg0.getOffset());
        if(arg1.getRecord() instanceof ObjectNode) {
          ObjectNode node = (ObjectNode)arg1.getRecord();
          value.set(node.toString());
        }
      }
    };
  }
  @Override
  public String getFilenameExtension() {
    return ".json";
  }
}
```

It is activated by updating `camus.properties` to use the appropriate class:

```
etl.record.writer.provider.class=wiley.streaming.camus.
  JSONRecordWriterProvider
```

Ingesting Data from Flume

Flume natively supports the Hadoop Distributed File System (HDFS) data ingestion as one of its sinks. For example, writing data in the same directory format as the earlier Camus example using an agent named agent99 is implemented as follows:

```
agent99.channels = channel1
agent99.sinks = kitchen
agent99.sinks.kitchen.type = hdfs
agent99.sinks.channel = channel1
agent99.kitchen.hdfs.path = /events/%y%m%d%H%M
agent99.kitchen.hdfs.round = true
agent99.kitchen.hdfs.roundValue = 5
agent99.kitchen.hdfs.roundUnit = minute
agent99.kitchen.hdfs.useLocalTimeStamp = false
```

In this example, the Hadoop Distributed File System (HDFS) sink is instructed to create paths using the local timestamp rounded down to the nearest 5-minute interval. The timestamp is obtained from the event itself by looking for a timestamp header.

Event versus Processing Time

It is tempting to use the event timestamp during this sort of ingest process to neatly place events into directories that make later analysis easier.

Unfortunately, doing this makes continuous process pipelines vastly more difficult to implement. In a processing environment, data may be arriving out of order. This means that one has to track where all the data ended up after an ingestion sequence. Keeping the data organized by the import time makes it very clear when the data was imported.

The reason it is important to keep track of the import time is that this is what has to be fixed when a bug is introduced. If the processing pipeline breaks, this import time identifies the data that has to be fixed, not the time when the data was generated. The time of the event does not matter for most recovery and maintenance situations, despite being slightly easier to process for users looking at historical data.

To use the local timestamp in Flume, simply change the `useLocalTimeStamp` parameter in the configuration to `true`.

Doing the same for Camus is a bit more complicated. The easiest way to do it is to choose an output directory before the job based on the current time. This can be passed in a configuration parameter and read by a custom `Partitioner` class:

```
public class BatchPartitioner implements Partitioner {
  String batch = null;
  @Override
  public String encodePartition(JobContext context,
    IEtlKey etlKey) {
    if(batch == null)
      batch = context.getConfiguration().get("batch","none");
    return batch;
  }
```

```
    @Override
    public String generatePartitionedPath(JobContext context,
     String topic,int brokerId, int partitionId,
     String encodedPartition) {
      return topic+"/"+encodedPartition;
    }
  }
```

This `Partitioner` is the set in the `camus.properties` file to override the default behavior:

```
elt.partitioner.class=wiley.streaming.camus.BatchPartitioner
```

Using Hadoop for ETL processes

After data is being routinely ingested into Hadoop for storage, it can also be integrated into ETL processes. This book is primarily concerned with real-time analysis so this will be discussed only briefly.

There are a number of options for ETL pipelines in Hadoop. Some of the more popular options are Pig and Hive. The former is a scripting language for Hadoop that was developed by Yahoo! for ETL processing. The latter is a SQL-like interface to Hadoop's Map-Reduce framework, which makes it a popular choice for database developers.

These tools, among others, are typically used in a multistep pipeline to produce a number of aggregated outputs. These are then made available for other pipelines or processing tools. Again, Hive is a popular choice here because it can be integrated with outside query tools. There are other specialized tools built to work directly with Hadoop available from commercial vendors.

The data can also be loaded from Hadoop into another database environment. Many database vendors provide connectors from Hadoop to their database to simplify this process. Another, more generic, option is Sqoop. This is an Apache project that is used for bulk transfers between Hadoop and other data stores. The package consists of a server that manages Hadoop jobs used for the bulk ingress or egress. It is controlled by a separate client process.

Lambda Architectures

One of the key features all of the storage systems discussed in this chapter possess is the ability to tune their update mechanism. Practically, this means abandoning transactions in favor of write speed. Combined with the at-least-once nature of the data-flow mechanisms, this can lead to both under-counting and over-counting of aggregates and other related problems.

To overcome this problem, Nathan Marz introduced the concept of the Lambda Architecture. In this architecture, the real-time system updates its data stores as before, without regard to transaction. It is accepted that these values are only an approximation of the true values.

The final values are computed using the data warehousing system. In most examples, these final values are computed by large Hadoop batch jobs. The front-end interface then manages the selection between the two systems. The other approach is to use the same database to store both and overwrite the real-time values with the values from the batch system as they become available. This requires fewer resources and is easier to manage on the front end, but it can be harder to implement. It is also often desirable to use different storage technologies for the short-term storage of the real-time system and the long-term storage of the “final” results coming from the batch system.

Conclusion

This chapter has covered the basics of some of the more popular data storage options available. Essentially all storage options work in a given situation given enough effort, but some make more sense for certain applications than others. This is often not an easy decision, so it is often best to try a few things out and experiment. The first attempt at scaling to available data will usually eliminate at least a few options.

Now that data is streaming into a processing system and that processing system has someplace to put its output, it is time to build some applications. The next chapter puts a face on the data by building a simple dashboard environment. This is the starting point of many streaming data projects because it gives people the ability to “feel” the data and interact with it.

Part II
Analysis and Visualization

In This Part

Chapter 7
Delivering Streaming Metrics

After streaming data has been collected, processed, and stored, it is time to deliver this data to end users. Historically, streaming data applications have employed customized “thick” clients of various kinds. With the introduction of web-based interfaces, the applications evolved to either applet-based approaches using Java or often to plug-in-based approaches. For example, Adobe has provided streaming interfaces via Flash using its LiveCycle server line for years.

Neither the applet nor the plug-in-based approach is particularly appealing in a modern delivery environment. First and foremost, with the rise of truly mobile computing, these options are simply not available for web-based delivery. Citing security and power usage concerns, modern browsers also severely limit these options, which makes using them more and more difficult in a desktop environment.

Fortunately, web browser standards have progressed to the point where rich data applications are possible, even those with streaming data connections. The standards most important for streaming applications are the Server-Side Events and Web Socket frameworks. Both of these options have wide browser adoption and allow for a web page to receive streaming updates.

Although it's not a complete guide to web application development, this chapter provides an introduction to the topic with a focus on real-time applications. It introduces the basics of web application back-end structures and adapts them to the particular problems of a real-time application, which are somewhat different than those of a traditional web application.

After data is delivered to the front end, presentation is the primary concern. Modern web browsers offer some advanced presentation techniques that supplant the Flash-based approaches of even a few years ago. These approaches are ideal as they work across a range of browsers—even mobile browsers—with a high rendering fidelity. One of the most popular tools is the D3 JavaScript library, which is widely used to render high-quality visualizations. This chapter covers the D3 toolkit, as well as several of the higher-level abstractions for D3, as it is a fairly low-level library.

Streaming Web Applications

The traditional web server relied on the operating system's multithreading or multiprocess support to handle connections. In the early days of Java web servers or Common Gateway Interface applications, when a new connection was established from a web browser, then a new OS-level thread or process (depending on the OS and specific server) would be spawned.

An operating system process has a fair amount of kernel overhead, and—although lighter than a process—a kernel thread also introduces some overhead and context-switching penalties as well. The advantage of this model is that it does not require anything exotic from the programmer other than an awareness of shared resources. The downside is that if the number of concurrent connections becomes large then the number of kernel resources required becomes nontrivial. In fact, it is possible to render a system inaccessible simply by causing it to spawn too many threads.

To get around this problem, web servers began to use “non-blocking” I/O rather than the traditional threading model. In this model, a server spins up a relatively small number of workers (usually one per system core) and then uses these non-blocking I/O mechanisms to handle incoming connections. Initially, this non-blocking I/O was only used for the actual web connection, but it has since expanded to be the standard mechanism for interaction in some environments.

One of these environments is *node.js* (often simply called Node), which is a server-side JavaScript application. It is based entirely around non-blocking I/O libraries (the specific library depends on what is available from the operating system) using a single execution thread. It has become a popular framework for developing web applications, and its ability to comfortably handle a relatively large number of persistent connections makes it a good choice for building streaming applications. Although it is based on a very fast JavaScript engine, the V8 engine found in Google's Chrome web browser, it is not as efficient as a Java-based server. However, the use of JavaScript on the back end, as well as the front end, allows for some interesting capabilities that outweigh the performance discrepancy for many applications.

The remainder of this chapter builds its streaming applications using node.js and its related libraries. This chapter assumes a basic familiarity with JavaScript development, but not with node.js specifically.

Working with Node

Despite being JavaScript, programmers coming from the browser side of web application development are often stymied by Node's processing model. The same is true of engineers coming from other web application frameworks, such as Ruby on Rails. The reason for the difficulty is Node's event-driven callback style. This style of programming is very different than what most programmers are used to, so it helps to begin with an overview of the style to understand how Node works.

NOTE

Node is a rapidly evolving project with regular releases. Some of these releases introduce or remove functionality, which makes compatibility between releases difficult. This book assumes a version of Node similar to the 0.10 application programming interfaces (APIs).

Callback-Driven Programming

In most languages, such as Java or Python, operations are assumed to be blocking. So, when code asks for a line from a file, it sits and waits until that line is completely available and then returns the line to the program. For instance, the following code for reading a file from an `InputStream` named `stream` in Java might look like this:

```
InputStreamReader reader = new InputStreamReader(stream);
BufferedReader    lines  = new BufferedReader(reader);
while(lines.ready()) {
  var line = lines.readLine();
  //Do something with the line here
}
```

In Node, this style of programming is generally unavailable because I/O is assumed to be non-blocking. (There are exceptions, but they are rare.) Instead, Node interfaces either take an object or respond to events. For example, in the following code a file is opened, but there is no function to read the next line as there would be in, say, a Java program. Instead, a function is bound to the `line` event that handles the processing of the line of data. Other events, not shown in this example, may also be bound to functions to allow the program to respond to the end of the file or an error generated by the reading process:

```
var readline = require('readline');
var lines    = readline.createInterface({
  input: stream
});
lines.on('line',function(line) {
  //Do something with line here
});
```

Escaping the "Callback Pyramid of Doom"

The callback style of programming is easy to follow when there is only one set of events to process that does not require further calls. Of course, this is never the case for nontrivial software, and the callback model begins to break down. For example, sequential operations quickly become hard to follow:

```
first(input,function(error,data) {
  second(data,function(error,moreData) {
    third(data,moreData,function(error,evenMoreData) {
      //Do something with the data...
      console.log("data: "+data+", moreData:"
        +moreData+" evenMoreData:"+evenMoreData);
    });
  })
});
```

This is colloquially known as the "callback pyramid (of doom)," and it troubles JavaScript programmers both on the front end and the back end. There are stylistic approaches that can reorganize the pyramid by rewriting the operations as separate named functions:

```
function doThird(data,moreData) {
  third(data,moreData,function(error,evenMoreData) {
  });
});
function doSecond(data) {
  second(data,function(error,moreData) {
    doThird(data,moreData);
  });
}
first(a,function(error,data) {
  doSecond(data);
});
```

But this style is actually fairly cumbersome to maintain in practice. Another approach is to use an `events` library, part of the Node core libraries, to use an `EventEmitter` to coordinate sequential events:

```
var coord = new require('events').EventEmitter();
coord.on('first',function(a) {
  first(input,function(error,data) { emit('second',data); }
});
coord.on('second',function(data) {
  second(data,function(error,moreData) {
    coord.emit('third',data,moreData);
  }
});
coord.on('third',function(data,moreData) {
  third(data,moreData,function(error,evenMoreData) {
  });
});
```

This example is very similar to the previous example, but the similarity between calls points to being able to create a library that can easily chain calls together. In fact, this has already been done in a number of different libraries that implement various techniques for arranging calls. One of these libraries, which is used extensively in this book, is called `async` and allows the previous example to be implemented as a `waterfall`:

```
var async = require('async');
var input = ...;
async.waterfall([
  function(cb) {
    first(input,function(error,data) {
      cb(data);
    });
  },
  function(data,cb) {
    second(data,function(error,moreData) {
      cb(data,moreData);
    });
  },
  function(data,moreData,cb) {
    third(data,moreData,function(error,evenMoreData) {
      //Do something here
      cb(); //All done
    });
  }
]);
```

Managing a Node Project with NPM

To manage a node project's code, including managing dependencies such as the `async` package used in the previous section, Node provides `npm` the "node package manager" in the node installation for most operating systems. This utility provides services very similar to Maven in the Java world or `gem` in the Ruby world.

Setting Up a Node Project

Starting a node project is as simple as creating a new directory and running `npm init`:

```
$ mkdir myproject
$ cd project
$ npm init
```

The NPM utility asks a series of questions about the project to get things started. The end result of this process is the creation of a `package.json` file in the directory where `npm` was run:

```
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sane defaults.
See `npm help json` for definitive documentation on these fields
and exactly what they do.
Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.
Press ^C at any time to quit.
name: (myproject)
version: (0.0.0) 1.0.0
description: My first node.js project
entry point: (index.js)
test command: mocha
git repository:
keywords:
author: Byron Ellis
license: (BSD)
About to write to /Users/bellis/Projects/wiley/chapter7/myproject
/package.json:
{
  "name": "myproject",
  "version": "1.0.0",
  "description": "My first node.js project",
  "main": "index.js",
  "scripts": {
    "test": "mocha"
  },
  "repository": "",
  "author": "Byron Ellis",
  "license": "BSD"
}
Is this ok? (yes) yes
```

It is also customary to include a `README.md` file in NPM-managed projects. If this file does not exist, NPM complains regularly until it is added. After the `package.json` file has been created, dependencies such as `async` can be added by adding a `dependencies` section:

```
{
  "name": "myproject",
  "version": "1.0.0",
  "description": "My first node.js project",
  "main": "index.js",
  "scripts": {
    "test": "mocha"
  },
```

```
  "repository": "",
  "author": "Byron Ellis",
  "license": "BSD",
  "dependencies":{
    "async":"*"
  }
}
```

To install the dependencies, simply execute `npm install` to get the latest version (because "*" was specified in the preceding code instead of a specific version number):

```
$ npm install
npm WARN package.json myproject@1.0.0 No README.md file found!
npm http GET https://registry.npmjs.org/async
npm http 304 https://registry.npmjs.org/async
async@0.2.9 node_modules/async
```

Another way of doing this is to install the async package with npm and use the `save` option:

```
$npm install async --save
```

This automatically updates the `dependencies` section of `package.json` with an appropriate entry for the module of interest.

After development has started, it is usually best practice to fix the version of any dependency to avoid bugs introduced by changes to those packages. NPM provides a facility called `shrinkwrap` that works similarly to the `Gemfile.lock` file in Ruby gems to specify specific versions of the packages.

Now, everything is ready to begin developing a node.js web application.

Adding Modules to a Project

Any nontrivial project should break down code into manageable modules to improve maintainability. In Node, this is done via modules that are implemented as simple JavaScript files and imported using the `require` statement. Unlike some languages, such as Java, there is no requirement that modules be located in any particular directory, though it is usually best to place them in a subdirectory to differentiate them from the main files used to run servers and other applications.

Modules can work two ways, exporting specific functions or exporting the entire module as a function. Some modules, such as the `connect` module used later in this chapter, use both methods. To export a function from a module, add that function to the `exports` block implicitly available inside a module:

```
function notExported() {
  console.log("This function can only be used internally");
}
exports.exported = function() {
  console.log("This can be called externally");
}
exports.callInternal = function() {
  notExported();
}
```

If these functions are in a file called `lib/module.js`, they can be imported and used via the `require` function:

```
var mod = require('./lib/module.js');
mod.exported();
```

```
// Prints "This can be called externally"
mod.callInternal();
// Prints "This function can only be used internally"
mod.notExported();
// Throws an exception "Object #<Object> has no method 'notExported'"
```

Returning a module-level function requires overriding the `exports` object itself by assigning a function to `module.exports`:

```
exports = module.exports = function() {
  //Do something here
};
```

Because functions are objects, other exports can continue to be added as above.

Developing Node Web Applications

Out of the box, node has APIs for creating and using web applications. There is a low-level `http` (or `https`) library that can be used to write a very simple web server. There are even built-in APIs for common web server functions like parsing query strings, encoding and decoding data and so on.

Implementing a basic server only takes a few lines of code:

```
var http        = require('http')
,   url         = require('url')
,   querystring = require('querystring')
;
http.createServer(function(request,response) {
  var query = querystring.parse(url.parse(request.url).query || "");
  response.writeHead(200,{'content-type':"text/plain"});
  response.end("Hello "+(query.name || "Anonymous")+"\n");
}).listen(3000);
```

Of course, this is a very low-level way to work and does not have any of the conveniences of a modern web framework. There are a variety of options to provide these higher-level interfaces. One of the most popular is the `connect` framework with `express` middleware. These two packages form the basis of the web applications developed in this chapter.

HTTP Middleware with Connect

The `connect` framework is often compared to Ruby's `rack` project. It builds on the basic HTTP server library to provide chainable pieces of code called *middleware*. These pieces of code—the core library contains nearly two dozen—are used to provide various components of a web server, such as query string parsing, authentication, or file serving.

Implementing a basic server capable of delivering static files only requires a few lines of code:

```
var connect = require('connect')
;
var app = connect()
.use(connect.static('public'))
.use(function(request,response) {
  response.writeHead(404,{'content-type':'text/plain'});
  response.end("File Not Found");
})
.listen(3000);
```

This server first creates a `connect` application by calling `connect()` and then begins chaining

together middleware with the `use()` function. Each piece of middleware is executed in the order it was added for each request.

Writing custom middleware is also easy because it is simply comprised of functions. All middleware functions take the form *`function(request,response,next)`* where the function may modify the *`request`* and *`response`* values before optionally calling the *`next`* function to proceed to the next piece of middleware.

Configuration of the middleware usually takes place by implementing a function that returns the middleware itself. This allows the middleware to be configured by taking advantage of the lexical scoping of JavaScript. For example, to implement a simple version of the `static` middleware might look something like this:

```
var url     = require('url')
,   path    = require('path')
,   fs      = require('fs')
;
function static(srcDir) {
  return function(request,response,next) {
    if(request.method != "GET") return next();
    var p = path.join(process.cwd(),
      srcDir,url.parse(request.url).pathname);
    fs.exists(p,function(yes) {
      if(!yes) return next();
      fs.createReadStream(p).pipe(response);
    });
  }
}
```

In this example, the `static` function returns another function configured with the source directory held in `srcDir`. This allows it to be used in `connect` by calling `use(static("public"))`. Functions contained within another function are called "inner functions" and have access to variables in the "outer" function through a feature of JavaScript called lexical scoping. When a request comes down the chain, first the inner function checks for a `GET` method request. If it is not the appropriate method, then the `next` function is called to continue the processing chain. Next, a path to the requested file is constructed and, if found, the file is piped to the HTTP response using the `pipe` method on Node's `ReadStream` class. Notice that the `next` function is not called. This is because the request chain should terminate at this point, as a response has been returned to the client.

The express.js Web Framework

Building on `connect` middleware the same way that `Sinatra` or `Rails` build on `Rack` in Ruby, `express.js` is a framework for building full-fledged web applications. It is designed to support the Model-View-Controller (MVC) style of web application, which works well with streaming data applications. Getting started with `express.js` looks a lot like starting a `connect` application:

```
var express = require('express')
,   path    = require('path')
,   app     = express()
;
app
.set("port",argv.port)
.set("views",path.join(__dirname,"views"))
.set("view engine","jade")
.use(express.favicon())
.use(express.logger('dev'))
```

```
  .use(express.json())
  .use(express.urlencoded())
  .use(express.methodOverride())
  .use(app.router) //MVC routing
  .use(require('less-middleware')({
    src: path.join(__dirname, 'public')
  }))
  .use(express.static(path.join(__dirname, 'public')));
  app
  .listen(argv.port);
```

Like the earlier `connect` example, the application is first initialized and then the base middleware attached. The `express` middleware mostly supersedes the `connect` middleware, but third-party `connect` middleware can still be used with `express`. In this case, the default middleware handles parsing of the various parts of the URL as well as the body. It also establishes static file delivery from the `public` directory and rendering of cascading style sheets (CSS) using the `less` package. This package is used by Twitter's Bootstrap, among others, to simplify styling the application. The "jade" view engine provides a template language used to create dynamic web pages within the application. It is used in the next section to describe the layout of the dashboard so that data can be streamed to it.

A Basic Streaming Dashboard

Using the `express` framework, this section builds a framework for a streaming dashboard. This is probably the first and most common application of streaming data. This section provides the framework and rendering of the dashboard, and the next section hooks the streaming back end to the front end.

Bootstrap Layouts

This dashboard uses layouts based on Twitter's Bootstrap 3 framework. This is a lightweight framework that integrates well with jQuery and provides easy integration of mobile features. Because Bootstrap will be used in all of the sections of the dashboard, it makes the most sense to add it directly to the layout of the entire web application. The default express layout is called `layout.jade`, which is in the `views` directory. To incorporate Bootstrap, add the appropriate style sheet and script links, highlighted in bold below:

```
doctype 5
html
  head
    title= title
    meta(name='viewport',content='width=device-width,
      initial-scale=1.0')
    link(rel='stylesheet',href='/stylesheets/bootstrap.min.css')
    link(rel='styleshhet',
      href='/stylesheets/bootstrap-theme.min.css')
    block styles
  body
    block content
    script(src='/javascripts/jquery.min.js')
    script(src='/javascripts/bootstrap.min.js')
    block scripts
```

In addition to the usual content block, this layout also incorporates specialized blocks for customizing styles and adding specialized scripts to a particular view.

A Dashboard View

This dashboard application has a single index view that renders all of the metrics to be displayed in the dashboard. A typical dashboard consists of a variety of small panels that contain metrics or visualizations. Visualizations of the data are covered in the "Visualizing Data" section of this chapter. The first task is to build metric panels. Bootstrap already has styling and markup for building simple panels. The following HTML fragment serves as a good starting point for building new panels:

```
<div id="first-metric" class="panel panel-default">
  <div class="panel-heading">
    <h3 class="panel-title">Header</h3>
  </div>
  <div class="panel-body">
    <!-- Body Goes Here -->
  </div>
  <div class="panel-footer">My First Metric</div>
</div>
```

This markup serves as a template for a `mixin` to the Jade template language. Using the `mixin` feature makes it easy to quickly create a layout for a number of metrics. The `mixin` defines all of the basic data fields for a metric, making it easy to bind data to them on the back end. The template language also allows for conditional statements, so the title and footer can be optional elements of the panel:

```
mixin metric(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      h1(data-counter="#{id}")= "--"
      h3
        span.glyphicon(data-direction="#{id}")
        span(data-change="#{id}")= ""
    if footer
      div.panel-footer= footer
```

This `mixin` is then added to the usual Bootstrap layout to build a simple dashboard with a number of different metrics in the `index.jade` view, located in the `views` director:

```
block styles
  link(rel='stylesheet',href='/stylesheets/dashboard.css')
block content
  div(class="container")
    div.page-header: h1 An Important Dashboard
    div(class="row")
      div(class="col-lg-3")
        +metric("first","Metric 1","My First Metric")
      div(class="col-lg-3")
        +metric("second","Metric 2","Another Metric")
      div(class="col-lg-6")
        +metric("third","Metric 3","A Bigger One")
    div(class="row")
      div(class="col-lg-2")
        +metric("m4")
      div(class="col-lg-2")
        +metric("m5")
      div(class="col-lg-2")
        +metric("m6")
      div(class="col-lg-2")
```

```
      +metric("m7")
    div(class="col-lg-2")
      +metric("m8")
    div(class="col-lg-2")
      +metric("m9")
```

This code snippet builds a two-row dashboard with a simple header. There are three panels of varying widths on the top row and six equally sized panels on the bottom row. When rendered in a browser, the dashboard should look something like Figure 7.1.

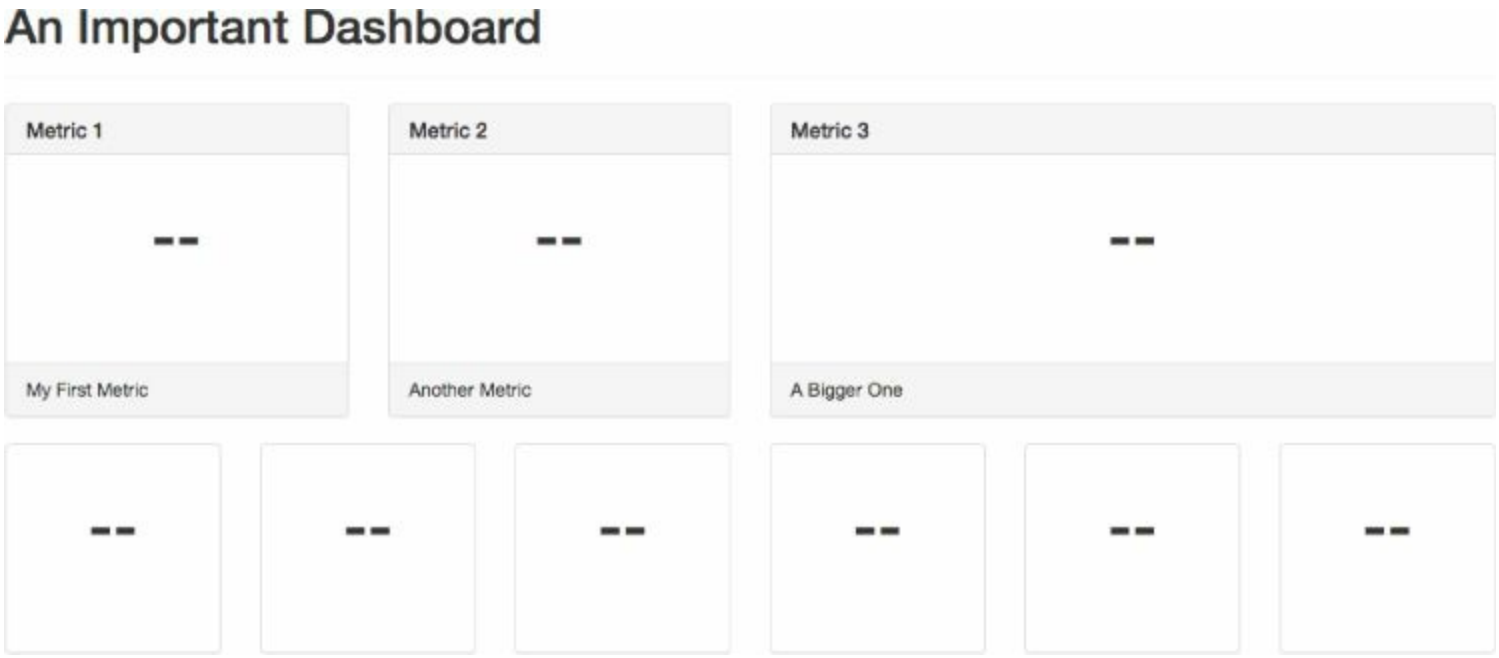


Figure 7.1

Updating the Dashboard

At the moment, all of the metrics being rendered are simple counters. More complicated graphical metrics will come later in this chapter, with the next chapter devoted to aggregated metrics like time-series displays.

To update the counters on the dashboard, they must first be registered with the application so they can begin to receive data. This is accomplished through the use of so-called data attributes. Custom data attributes are an HTML5 innovation that allows arbitrary user attributes to be bound to various elements of the Document Object Model (DOM).

jQuery and other frameworks provide selectors to query for the presence of a specific data element. To find all of the `data-counter` elements in the dashboard, the query looks something like this:

```
$('*[data-counter]').each(function(i,elt) {
    //elt is the document element
  });
```

Because the `mixin` from the last section specifies the metric in the `data-counter` field, this query can be used to bind the counter to a custom event such that it will update the counter every time the event fires. First, select each of the counters to bind:

```
function bindCounters() {
  $('*[data-counter]').each(function(i,elt) {
    bindCounter($(elt).attr("data-counter"),$(elt))
  });
```

```
}
```

Next, find the directional indicator, change the amount fields included in the counter metric, and bind the whole thing to a custom event of the form `data:<counter name>`. Found in `public/javascripts/dashboard.js`, the implementation is as follows:

```
function bindCounter(counterName,$elt) {
  $elt.text("--");
  var $dir = $('span[data-direction='+counterName+']');
  var $chg = $('span[data-change='+counterName+']');
  var last = NaN;
  $(document).on('data:'+counterName,function(event,data) {
    $elt.text(data);
    if(isFinite(last)) {
      var diff = data - last;
      if(diff > 0) {
        $dir.removeClass("glyphicon-arrow-down");
        $dir.addClass("glyphicon-arrow-up");
      } else {
        $dir.removeClass("glyphicon-arrow-up");
        $dir.addClass("glyphicon-arrow-down");
      }
      diff = Math.abs(diff);
      if($chg.hasClass("percentage")) {
        diff = Math.round(1000*diff/last)/10;
        $chg.text(diff+"%");
      } else {
        $chg.text(diff);
      }
    }
    last = data;
  });
  $chg.on('click',function() { $chg.toggleClass("percentage"); });
}
```

This code wraps up all the information needed to update the counter as well as the directional indicator. The code also binds a `click` event to the directional indicator to allow for toggling between an absolute display and a percentage display.

To see these metrics in action, the following is a small driver that triggers each of the counter events with a random value. The triggers themselves are set to randomly fire between 1 and 5 seconds and can be found in `public/ javascripts/counter.js`:

```
$(document).ready(function() {
  $('*[data-counter]').each(function(i,elt) {
    var name = $(elt).attr("data-counter");
    setInterval(function() {
      $(document).trigger("data:"+name,
        Math.round(10000*Math.random()));
    },1000+Math.round(4000*Math.random()));
  });
});
```

The counters, shown in Figure 7.2, are now ready to accept data streamed in from the server itself.

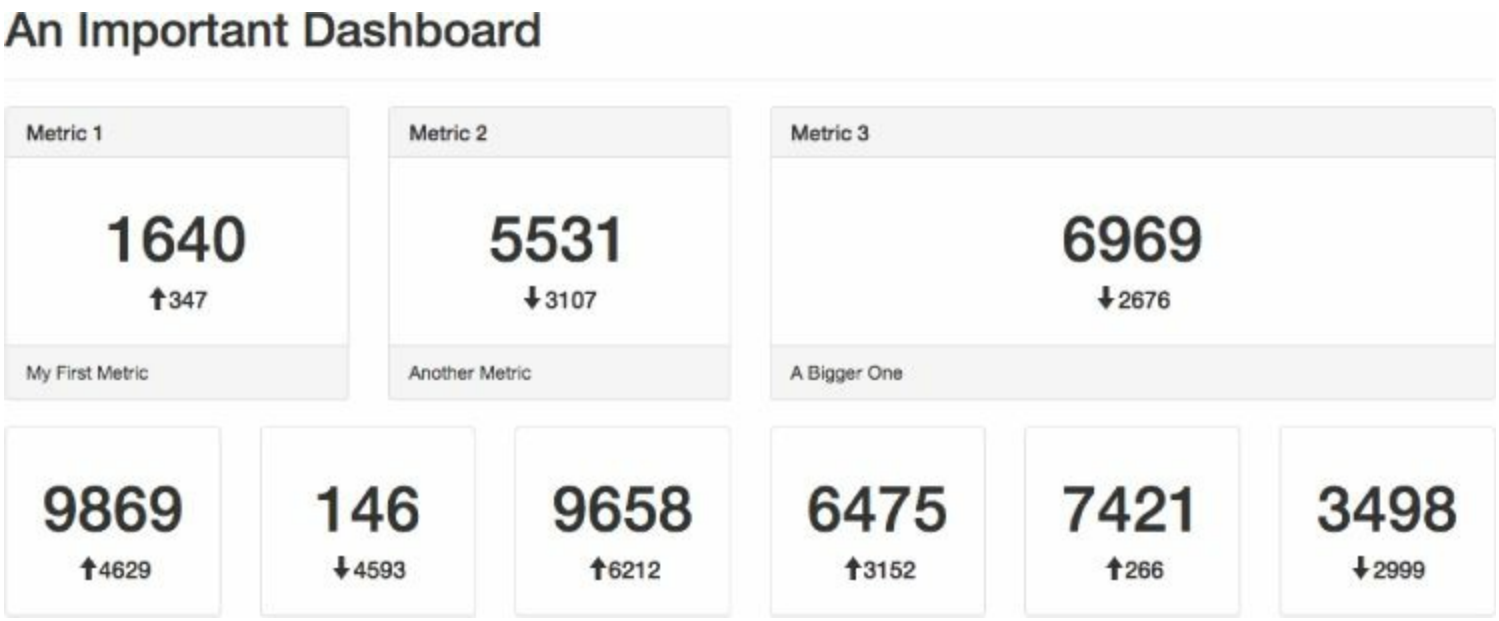


Figure 7.2

Adding Streaming to Web Applications

With the dashboard counters ready for streaming data, there are two viable web options for web-based delivery of data: SSEs and WebSockets. Both technologies enjoy fairly broad support across modern browsers, including mobile browsers. Both have their pros and cons making the choice of a particular technology more a matter of preference than requirements.

This section covers how to use both technologies to communicate with the back-end server. The two technologies do not interfere with each other, so it is perfectly reasonable to provide back-end support for both styles.

Server-Side Counter Management

Before introducing the communication technologies, the counters to be used in the dashboard must be maintained on the server side. Otherwise, there will be nothing to send to the client after it is wired to the server! In this example application, counters are maintained in a singleton object derived from Node's built-in `EventEmitter` class with the implementation found in `dashboard/lib/counters/index.js`:

```
var events = require('events')
,   util   = require('util')
;
function Counter() {
  //Returns the singleton instance if it exists
  if(arguments.callee.__instance) {
    return arguments.callee.__instance;
  }
  arguments.callee.__instance = this;
  this._state = { };
  events.EventEmitter.call(this);
}
util.inherits(Counter,events.EventEmitter);
module.exports = new Counter();
```

The singleton instance is stored in the *`arguments.callee.__instance`* variable. This allows the counter object to be imported with a `require` statement.

The state for the counter object is, for the moment, stored in the object itself. A more complicated dashboard would persist this data somewhere to survive restarts. This issue is tackled later in this section.

Updating counter state is handled through methods on the counter object. The basic operations to support are the `get`, `set`, and `increment` methods:

```
Counter.prototype.state = function(cb) {
  return cb(null,this._state);
};
Counter.prototype.get = function(counter,cb) {
  return cb(null,this._state[counter] || 0);
};
Counter.prototype.set = function(counter,value,cb) {
  this._state[counter] = (value || 0);
  this.emit("updated",counter,this._state[counter]);
  return cb(null,this._state[counter]);
};
Counter.prototype.increment = function(counter,amount,cb) {
  var self = this;
  this.get(counter,function(err,count) {
    self.set(counter,count + 1*(amount||1),cb);
  })
};
```

Note that the `set` method in the preceding code emits an `updated` event whenever a counter is changed. This is the primary interaction any streaming connection has with the counters. It simply listens to this event when the connection is opened and removes itself from the set of listeners when the connection closes.

Now that the counter state can be maintained on the server side, a simple API is needed to allow updates to the state. This will then be attached to the client-side interface via both SSEs and WebSockets in the next two sections. Express makes it easy to implement simple REST-style APIs to the counter singleton, as seen in this code found in `dashboard/index.js`:

```
var counters = require('./lib/counters');
app.set("counters",counters);
app.get('/api/v1/incr/:counter', function(req,res) {
  counters.increment(req.params.counter,req.query.amount,
    function() {
      res.end("OK\n");
    });
});
app.get('/api/v1/counter/:counter',function(req,res) {
  counters.get(req.params.counter,function(err,data) {
    res.end(data+"\n");
  });
});
app.get('/api/v1/state',function(req,res) {
  counters.state(function(err,data) {
    for(var counter in data) {
      res.write(counter+": "+data[counter]+"\n");
    }
    res.end();
  });
});
```

Server Sent Events

Server Sent Events (SSEs) were added to the HTML5 specification by the Web Hypertext Application Technology Working Group (WHATWG). Originally introduced in 2006, the SSE protocol uses HTTP as its transport mechanism with the `text/event-stream` Content Type. The basic response is a line prefixed with `data:` and ending with `\n\n` (two newline characters). A multiline response is allowed by only using a single newline, but it still starts with the `data:` prefix. The web server may set an optional retry timeout, event, and an ID field using the `retry:`, `event:`, and `id:` prefixes, respectively. A complete response looks something like this:

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/event-stream
Cache-Control: no-cache
Connection: keepalive
Transfer-Encoding: chunked
event:alpha
id:1234
retry:10000
data:{"hello":"world"}
id:12345
data:{"again":"and again"}
```

This response returns a simple one-line message with an `id` of 1234 and sets the retry time from 30 seconds to 10 seconds. To read this message, supporting browsers use the `EventSource` class. This class fires an event corresponding to the event name given in the response. If no event is specified, the object fires a `message` event.

To modify the example dashboard application to respond, the server fires counter events to transmit a JSON object containing updates to each counter.

First, the application must establish a connection to the server. This is mostly handled by the `EventSource` object, which takes a URL to the SSE connection. In this case, the connection lives at `/api/v1/dashboard` and is implemented in `dashboard/public/javascripts/sse.js`:

```
$(document).ready(function() {
  var events = new EventSource('/api/v1/dashboard');
```

The `EventSource` object emits a number of events besides the messaging events described earlier. Handling those events is optional, but it often provides useful debugging information. The most important events are the `open` and `error` events, which are called when the event stream is successfully established and when the event stream is closed, or another error event occurs (such as a 500). In this simple example, these events simply report being called to the console for debugging purposes:

```
  events.addEventListener('open',function(e) {
    if(window.console) console.log("opened SSE connection");
  });
  events.addEventListener('error',function(e) {
    if(e.readyState != EventSource.CLOSED) {
      if(window.console) console.log("SSE error");
    } else
      if(window.console) console.log("closed SSE connection");
  });
```

Finally, the `counters` event is handled to distribute the counter state to the already established listeners:

```
  events.addEventListener('counters',function(e) {
    var data =  JSON.parse(e.data);
    for(counter in data)
      $(document).trigger('data:'+counter,data[counter]);
  });
});
```

To respond to this event, the Express application defines a route corresponding to the endpoint used to create the `EventSource` object, using the SSE middleware function to provide the SSE protocol added `dashboard/index.js`:

```
app.get('/api/v1/dashboard', sse(), routes.dashboard);
```

The Server Sent Events middleware is a fairly simple piece of code, implemented in `dashboard/lib/sse/index.js`. This particular implementation maintains unique message IDs for events on a per-event type basis as well as allowing for a default event name:

```
module.exports = function(options) {
  options = options || {};
  return function sse(req,res,next) {
    res._counts = {};
    res._nextId = function(type) {
      type = type || "message";
      count = (this._counts[type] || 0) + 1;
      this._counts[type] = count;
      return count;
    }
```

When a connection begins, it must check that the call is coming from a source that supports SSE. If this is the case, it sends the appropriate content type and other settings to start the connection:

```
if(req.accepts("text/event-stream")) {
      req.setTimeout(Infinity);
      res.writeHead(200,{
        'Content-Type':  'text/event-stream',
        'Cache-Control': 'no-cache',
        'Connection':    'keepalive'
      });
    } else {
      res.writeHead(200,{
        'Content-Type': 'application/json',
        'Cache-Control': 'no-cache',
        'Connection': 'keepalive'
      })
    }
```

Before continuing along the chain, the middleware adds functions to the response object to transmit SSE events in the appropriate format:

```
    res.event = function(event,type,opt) {
      opt  = opt  || {};
      type = type || options.event;
      if(type)
        this.write("event:"+type+"\n");
      this.write("id:"+this._nextId(type)+"\n");
      if(opt.retry)
        this.write("retry:"+opt.retry+"\n");
      this.write("data:"+event+"\n\n");
    }
    res.json = function(json,type,opt) {
      this.event(JSON.stringify(json),type,opt);
```

```
    }
    next();
  }
};
```

Finally, the endpoint itself retrieves the current counter state when the connection starts and sends the counter data immediately. Then, the handler for this connection attaches itself to the `Counter` object to obtain future counter updates from the server, which the handler then passes along to the client. The implementation of the handler below can be found in `dashboard/routes/index.js`:

```
exports.dashboard = function(req,res) {
  var counters = req.app.get('counters');
  counters.state(function(err,data) {
    console.log(data);
    res.json(data,"counters");
  });
  function update(counter,value) {
    var msg = {};msg[counter] = value;
    res.json(msg,"counters");
  }
  counters.on("updated",update);
  res.on("close",function() {
    counters.removeListener("updated",update);
  });
};
```

With everything in place, the dashboard should now respond to state updates from the command line. Try starting the dashboard and then executing the following commands from another window:

```
$ curl http://localhost:3000/api/v1/incr/first?amount=25
OK
$ curl http://localhost:3000/api/v1/incr/second?amount=25
OK
$ curl http://localhost:3000/api/v1/incr/third?amount=50
OK
```

After the commands have been executed, the dashboard display should look like Figure 7.3.

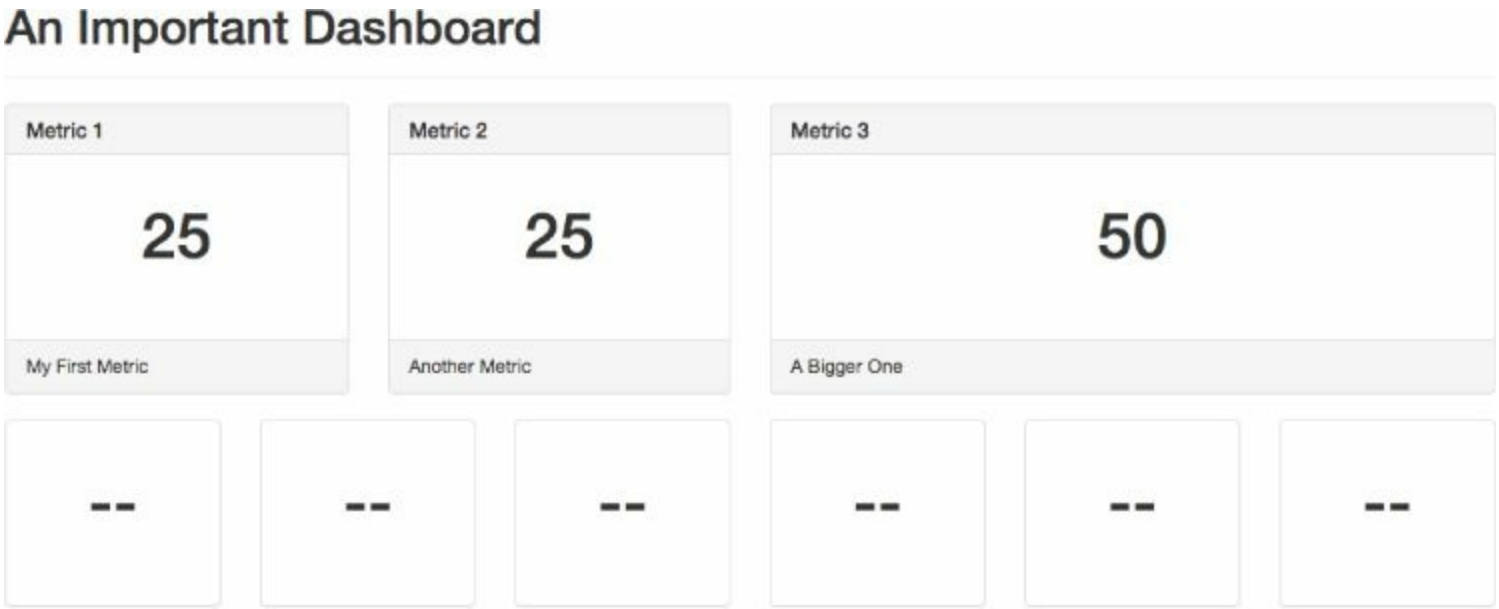


Figure 7.3

SSEs enjoy support on all WebKit browsers such as Safari and Chrome. This includes their Mobile

counterparts on iOS and Android. It is also supported by Firefox and Opera, but not by Internet Explorer or the default Android browser (which is not Chrome for some reason). However, because the specification relies only on basic HTTP, it is possible to use "polyfill" libraries to provide support. At the time of writing, there are three well-known polyfill libraries for SSEs, with Remy Sharp's polyfill library (https://github.com/remy/polyfills/) providing an interface most closely matching that provided by native implementations and allowing support in Internet Explorer 7 and later versions.

WebSockets

The WebSocket protocol, also known as RFC 6455, is a bidirectional communications layer designed for so-called "rich" clients. It maintains communications over port 80, but unlike SSEs, it "upgrades" the connection to support communication through a handshake process.

There have been several different versions of the standard since 2010, and there is fairly good browser support for the modern secure version of the standard. Support for it started in Internet Explorer 10, Chrome 14, Safari 6, Firefox 6, and modern versions of Opera. WebSockets are also supported by the mobile versions of these browsers. Chrome also supports an experimental SPDY version of the protocol, but at the time this book is being written it has to be enabled with a startup switch, so there is effectively no support for this "in the wild."

Because the protocol is more complicated than Server Sent Events, the dashboard application relies on a library for WebSocket support, rather than just implementing the server and client directly. There are several WebSocket libraries available for Node, but one of the more popular libraries is `socket.io`, which provides both a server-side library as well as a client-side library with identical interfaces. It also standardizes browser API differences to provide a common interface on all platforms. The easiest way to use `socket.io` is to attach its server interface directly to the Express application and treat it as, essentially, a separate interface to the application. To do this requires creating an `http.Server` object that replaces the `listen()` method on the application. In the dashboard example, this involves creating the `server` object when the application is initialized in `dashboard/index.js`:

```
var express = require('express')
,   routes  = require("./routes")
,   path    = require('path')
,   app     = express()
,   server  = require('http').createServer(app)
,   sse     = require('./lib/sse')
;
```

Then, `app.listen(argv.port)` is replaced with `server.listen(argv.port)` and the `socket.io` code is attached to the server after the application has been initialized:

```
server.listen(argv.port);
var ws = require('socket.io').listen(server);
ws.sockets.on('connection',function(socket) {
  function update(counter,value) {
    var msg = {};msg[counter] = value;
    socket.emit('counters',msg);
  }
  socket.emit('counters',counters.state);
  counters.on('updated',update);
  socket.on('disconnect',function() {
    counters.removeListener('counters',update);
```

```
    });
  });
```

The implementation itself is very similar to the `dashboard` endpoint of the Server Sent Events version. When a change to the counters is made, it is relayed out to the dashboard to allow it to update the interface.

Implementing the client side is also quite similar to the Server Sent Events version. First, the `socket.io` library must be included. Because the library itself provides both server-side and client-side versions, it automatically creates an endpoint to serve up the appropriate client-side library at `/socket.io/socket .io.js` when the server is attached to the Express server, as it was earlier. The dashboard HTML then only needs to be modified slightly to support WebSockets, including the extra script tag:

```
block scripts
  script(src='/socket.io/socket.io.js')
  script(src='/javascripts/dashboard.js')
  script(src='/javascripts/ws.js')
```

All that remains, as with the SSE version, is to redistribute the counter update events when they come in over the WebSockets connection. In this case, even less code is required because the events have already been converted back to their JSON form by the `socket.io` library:

```
$(document).ready(function() {
  var socket = io.connect("/");
  socket.on('counters',function(data) {
    for(counter in data)
      $(document).trigger('data:'+counter,data[counter]);
  });
});
```

Tip

Because WebSockets and SSE use different endpoints and different protocols, a server can easily support both of them as transport mechanisms for streaming events. This is especially useful when you're trying to support a large number of devices or even native implementations (in the case of mobile devices).

NOTE

A third technology for streaming communication between the server and the browser is the WebRTC standard. Originally developed for peer-to-peer audio/visual communication, this standard also has the ability to transmit arbitrary data across the wire. Support at the moment is limited only to newer desktop versions of Firefox and Chrome. The hope is that this will change in the not too distant future, providing a rich new streaming data interface.

A Redis-Based Dashboard

While the in-memory counter state with a REST-style interface is useful for development, the previous chapters have covered the data pipeline for a real streaming web application. In this case, stream processing is handled by either Storm or Samza, and updates are sent to some sort of persistent store.

Aggregation and maintenance of counters is covered in more detail in Chapter 8, "Exact Aggregation and Delivery," but you can easily adapt the simple counter dashboard from before to a persistent store. This can then be driven from a stream-processing environment.

This example uses Redis because it is a relatively simple key-value store that also has a pubsub facility that is used to drive the updates. A persistent store without this would require an additional messaging facility to drive the dashboard servers (or use polling on the server side to drive updates).

In the example dashboard, the server keeps track of a single Redis hash that can be updated by external interfaces as well as by the dashboard interface. First, Redis configuration parameters are added to the arguments for the server in `dashboard/index.js`:

```
var argv    = require('optimist')
.usage("Usage: $0")
.options("port",{alias:"P",default:3000})
.options("redis",{alias:"r"})
.options("counters",{alias:"c",default:"counters"})
.argv
;
```

These are used to drive the selection of the appropriate counter library:

```
var counters = argv.redis ?
  require('./lib/counters-redis')(
    argv.redis,
    {counters:argv.counters})
: require('./lib/counters');
```

The interface code has already been written in a callback style, so no changes are required to that portion of the interface. Likewise, both the SSE and WebSocket interfaces have been implemented to respond to events emitted by the `Counter` class. The only thing that needs to be implemented is the new `Counter` class itself.

Rather than simply creating a singleton object, the new `Counter` class needs to take in parameters that identify the location of the Redis server. This is used to construct two Redis connections. The first, called `client`, is used for all normal Redis operations. The second, called `pubsub`, is used to handle the event delivery for Redis operations. The reason two client connections are used is because Node's Redis client is placed into a special mode when the `subscribe` command is used; that mode prevents the client from executing normal Redis operations, found in

dashboard/lib/counters-redis/index.js:

```
function Counter(host,options) {
  //Returns the singleton instance if it exists
  if(arguments.callee.__instance) {
    return arguments.callee.__instance;
  }
  arguments.callee.__instance = this;
  events.EventEmitter.call(this);
  this.options     = options || {};
  this.client      = redis.createClient(this.options.port || 6379
   ,host);
  this.counterName = this.options.counters || "counters";
  this.pubsub = redis.createClient(this.options.port || 6379,host);
  var self = this;
  this.pubsub.on('ready',function() {
    self.pubsub.subscribe(self.counterName);
  });
  this.pubsub.on('message',function(channel,message) {
    if(channel == self.counterName) {
      self._sendUpdate(message);
    }
  });
}
util.inherits(Counter,events.EventEmitter);
module.exports = function(host,options) {
  new Counter(host,options);
}
```

When a new counter event arrives on the `pubsub` connection that corresponds to the hash being monitored by the server, the `_sendUpdate` method is used to inform all subscribed clients:

```
Counter.prototype._sendUpdate = function(counter) {
  var self = this;
  this.get(counter,function(err,data) {
    if(err) return;
    self.emit("updated",counter,data);
  });
};
```

This allows outside processing systems to push updates to the client by sending a `publish` event through Redis for the appropriate channel. It is also allows the `set` and `increment` events to be further decoupled from the `Counter` implementation:

```
Counter.prototype.set = function(counter,value,cb) {
  var self = this;
  this.client.hset(this.counterName,counter,value,
    function(err,data) {
      if(err) return cb(err,data);
      self.client.publish(self.counterName,counter);
    });
};
Counter.prototype.increment = function(counter,amount,cb) {
  var self = this;
  this.client.hincrby(this.counterName,
    counter,amount,function(err,data) {
      if(err) return cb(err,data);
      self.client.publish(self.counterName,counter);
      cb(err,data);
  });
};
```

Note that the `increment` method no longer relies on the `get` and `set` methods as Redis provides its own primitives for atomically incrementing values. The get and state methods now simply have to query the Redis store, revealing why the callback structure was used in the original implementation despite not being strictly necessary:

```
Counter.prototype.state = function(cb) {
  this.client.hgetall(this.counterName,cb);
};
Counter.prototype.get = function(counter,cb) {
  this.client.hget(this.counterName,counter,cb);
};
```

After starting the dashboard again, running the commands that resulted in Figure 7.3 should return the desired results. To see events being updated from outside of the dashboard it is now possible to push events from the Redis command-line client:

```
$redis-cli
redis 127.0.0.1:6379> hset counters m4 4
(integer) 1
redis 127.0.0.1:6379> hset counters m5 5
(integer) 1
redis 127.0.0.1:6379> hset counters m6 6
(integer) 1
redis 127.0.0.1:6379> publish counters m4
(integer) 1
redis 127.0.0.1:6379> publish counters m5
(integer) 1
redis 127.0.0.1:6379> publish counters m6
(integer) 1
```

This results in a dashboard that looks like Figure 7.4.

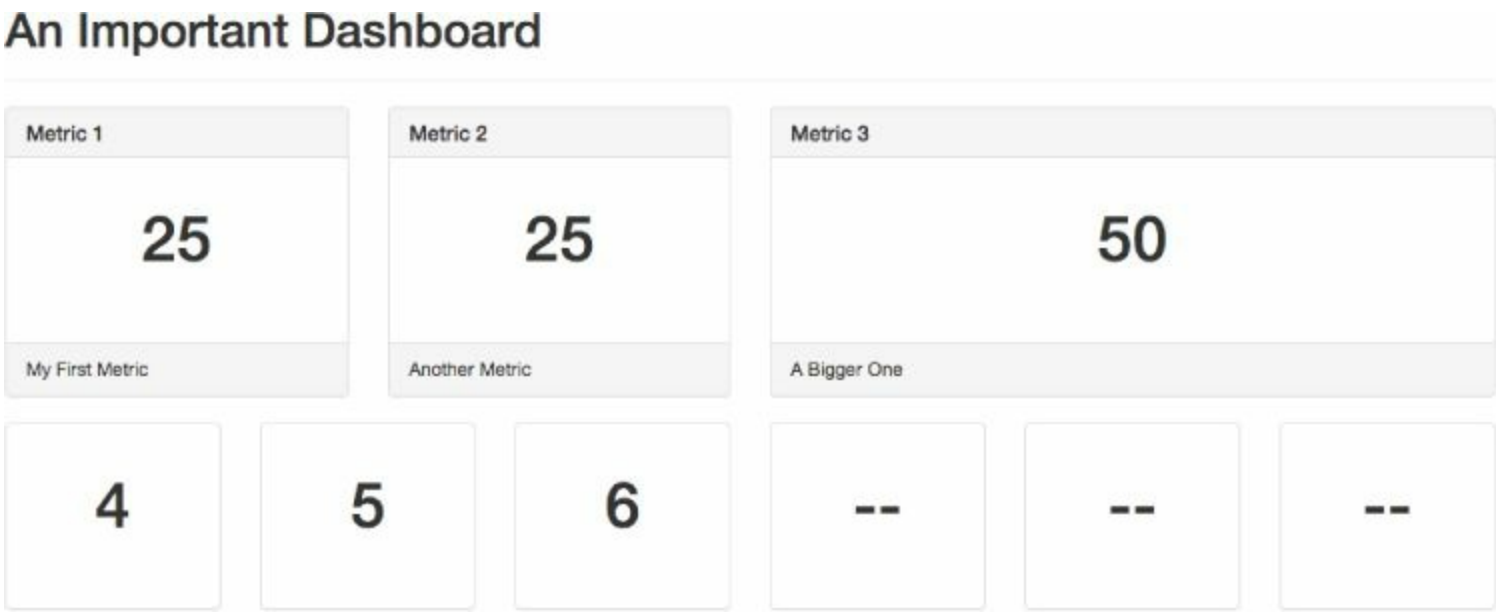


Figure 7.4

Visualizing Data

For years, visualizing data in a web browser was limited to two options: server-generated static images and plug-in-based visualizers, usually implemented using Flash.

Although static images offer flexibility, rendering a streaming display was not technically feasible. At best, updates every 30 seconds to a minute were feasible. These interfaces are in any number of server-monitoring solutions, such as Graphite, which render static images on the server side and transmit them to the client every so often.

This approach is not real time, nor does it offer any interactivity. It is also a heavy approach because it requires retransmitting an entire image to accommodate potential small changes in the data. To overcome this, Flash- or Java-based plug-ins were often used to allow incremental data updates.

Both of these options are still used when it is necessary to support older (or simply inferior) browsers. Modern browsers can make use of HTML Canvas and Inline SVG to support rich, plug-in-free visualization techniques directly in the browser.

HTML5 Canvas and Inline SVG

The HTML5 Canvas element and Inline SVG provide rich rendering interfaces within an HTML document without the use of plug-ins. As the names suggest, the Canvas element offers a pixel-based interface similar to rendering surfaces used in native desktop or mobile applications. Inline SVG, on the other hand, uses the DOM to construct a graphical representation that is then rendered along with the rest of the document.

There are tradeoffs to both approaches. The Canvas interface is typically faster to render, but it has far less native support for complicated rendering. Inline SVG has built-in support for layering and grouping as well as supporting CSS styling in many browsers, but it can be much slower to render large scenes due to the need to maintain the scene graph. Rendering performance typically dictates the choice of technique; so using Inline SVG until performance becomes an issue is the most common approach.

Using the HTML5 Canvas

The HTML5 Canvas element is a resolution-dependent rendering surface supported in every modern browser, including Internet Explorer. (Versions 7 and 8 require an external library. Versions 9 and newer support the canvas element natively.)

To use a Canvas, an element needs to be added to the page. Modifying the metric `mixin` slightly puts a canvas element behind each metric on the page, appropriately marked with a data element for later updating. This `mixin` is implemented in the various. `.jade` view files found in `dashboard/views` such as `dashboard/views/canvas_ex1.jade`:

```
mixin metric(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      div(class="canvas-wrap")
        canvas(data-canvas-gauge="#{id}")
      div(class="value-wrap")
        h1(data-counter="#{id}")= "--"
        h4
```

```
        span.glyphicon(data-direction="#{id}")
        span(data-change="#{id}")= ""
    if footer
      div.panel-footer= footer
```

Without styling, the elements would be arranged one after another, so the next step is to stack the elements on top of each other and set the appropriate z-index, which makes the elements render in the correct order. This is added to `dashboard/public/stylesheets/dashboard.css`:

```
.metric div.canvas-wrap {
  position: absolute;
  width: 100%;
  height: 100%;
  z-index: 1;
}
.metric div.canvas-wrap canvas {
  width: 100%;
  height: 100%;
}
.metric div.value-wrap {
  position: relative;
  width: 100%;
  z-index: 2;
}
```

To draw the gauge over each counter, first the canvas context needs to be obtained:

```
$('*[data-canvas-gauge]').each(function(i,elt) {
  fixup(elt)
  var name = $(elt).attr("data-canvas-gauge");
  var ctx  = elt.getContext("2d");
```

The `fixup` function, implemented in `dashboard/public/javascripts/canvas .js`, makes some slight alterations to the position of the canvas wrapper. This positions the canvas properly in the center of the rear panel as well as sets the resolution of the canvas surface to match that of the element:

```
function fixup(elt) {
  var $elt = $(elt);
  var $wrap   = $elt.parent();
  var $parent = $wrap.parent();
  var pos     = $parent.position();
  $wrap.width($parent.width());
  $wrap.height($parent.height());
  elt.width = $parent.width();
  elt.height= $parent.height();
}
```

A canvas element actually has two sizes: the size of the element itself, set by the `style` attribution, and the size of the canvas surface, set by the `width` and `height` attributes. This is often confusing for first-time canvas users, and the preceding code takes the opportunity to make the canvas resolution the same as the element size for ease of use.

Now that a surface context has been obtained and the canvas is moved into the proper location, drawing can begin. Elements are created using various “pen movement” commands, such as `moveTo` and `lineTo`, and curve commands, such as `arcTo`. These are then either outlined using the `stroke` command or filled using the `fill` command. The surface, like most similar implementations, also has convenience functions for rectangles. In this case the gauge is drawn with a

`drawGauge` function, implemented in `dashboard/public/javascripts/canvas.js`, that draws a circular gauge like the one shown in Figure 7.5:

```
function drawGauge(ctx,value,max) {
    //Clear the gauage
    ctx.fillStyle = "#fff";
    ctx.clearRect(0,0,ctx.canvas.width,ctx.canvas.height);
    //Draw the gauge background
    var centerX = (ctx.canvas.width) / 2;
    var centerY = (ctx.canvas.height) /2;
    var radius  = Math.min(centerX,centerY) - 12.25;
    ctx.beginPath();
    ctx.arc(centerX,centerY,radius,0.6*Math.PI,2.4*Math.PI)
    ctx.strokeStyle = "#ddd";
    ctx.lineWidth   = 25;
    ctx.stroke();
    var newEnd = (2.4-0.6)*(value/max) + 0.6;
    ctx.beginPath();
    ctx.arc(centerX,centerY,radius,0.6*Math.PI,newEnd*Math.PI)
    ctx.strokeStyle = "#aaa";
    ctx.lineWidth   = 25;
    ctx.stroke();
  }
```

The function first needs to clear the drawing surface, which is done with the `clearRect` function. Note that the canvas knows its own internal size, so the size of the rectangle does not need to be hardcoded. Next, a background arc is drawn to provide context for the gauge itself. This uses the `arc` function to draw a partial circle. Rather than attempting to draw the outer and inner arcs and filling the result, the `lineWidth` is set to a relatively high value and then the path is `stroke`'d rather than filled to produce the background. The radius is adjusted slightly so that the outer edge of the path does not fall outside of the canvas.

The entire process is repeated again using a slightly darker color to indicate the gauge value. For this second path, a new ending position is calculated based on the current value and the maximum observed value for this particular metric. If the gauge was known to have a finite range, this maximum value could be set to the top of the range instead.

This is then attached to the data events to update the gauge background each time the metric itself is adjusted:

```
var max = 100;
 $(document).on('data:'+name,function(event,data) {
   if(data > max) max = data;
   drawGauge(ctx,data,max,dims);
 });
```

The end result of adding this to the original example dashboard is shown in Figure 7.5. Each element now has a gauge rendering behind it that updates each time the metric changes.

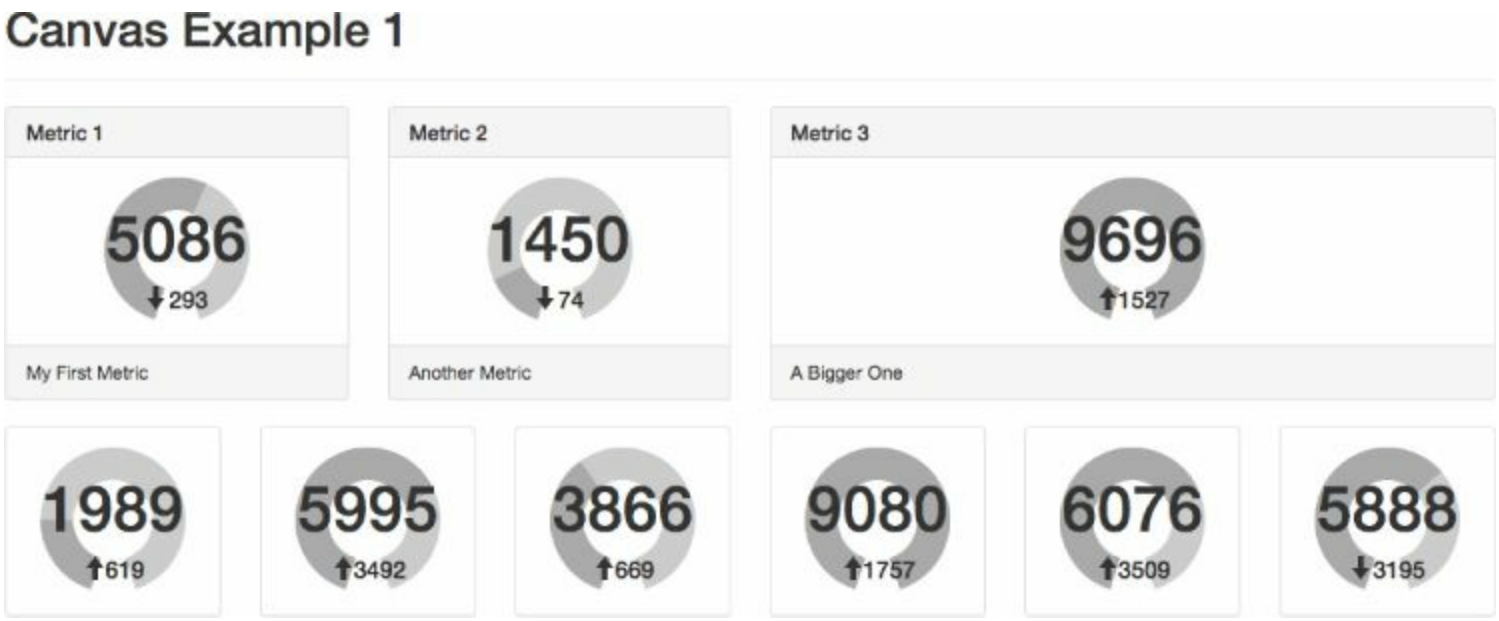


Figure 7.5

Another example of using the Canvas would be to draw a more traditional chart in the background of the metric. To mix these two types of visualizations in the same dashboard can be accomplished with a second `mixin` designed to handle charts, such as the one found in `dashboard/views/canvas_ex2.jade`:

```
mixin metricChart(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      div(class="canvas-wrap")
        canvas(data-canvas-chart="#{id}")
      div(class="value-wrap")
        h1(data-counter="#{id}")= "--"
        h4
          span.glyphicon(data-direction="#{id}")
          span(data-change="#{id}")= ""
    if footer
      div.panel-footer= footer
```

Using this new `mixin`, the first row of metrics can be converted to the "chart" mode instead of the "gauge" mode, as in the `dashboard/views/canvas_ex2.jade` example:

```
block content
  div(class="container")
    div.page-header: h1 Canvas Example 2
    div(class="row")
      div(class="col-lg-3")
        +metricChart("first","Metric 1","My First Metric")
      div(class="col-lg-3")
        +metricChart("second","Metric 2","Another Metric")
      div(class="col-lg-6")
        +metricChart("third","Metric 3","A Bigger One")
```

Attaching a different data element to the chart canvases allows them to be selected and associated with a different drawing method. In this case, the rendering creates a simple area chart by creating an array of the last `maxValues` data elements to be emitted for each counter, as implemented in `dashboard/public/javascripts/canvas.js`:

```
var maxValues = 20;
function drawChart(ctx,values,max) {
  //Clear the gauage
  ctx.clearRect(0,0,ctx.canvas.width,ctx.canvas.height);
  var valHeight = ctx.canvas.height/max;
  var valWidth  = ctx.canvas.width/(maxValues-1);
  if(values.length < 2) return;
  ctx.beginPath();
  ctx.moveTo(0,ctx.canvas.height);
  for(var i=0;i<values.length;i++) {
    ctx.lineTo(i*valWidth,ctx.canvas.height - values[i]*valHeight);
  }
  ctx.lineTo(valWidth*(values.length-1),ctx.canvas.height);
  ctx.fillStyle = "#ccc";
  ctx.fill();
}
```

Finally, this draw method is executed after pushing each element onto the data array, producing Figure 7.6:

```
$('*[data-canvas-chart]').each(function(i,elt) {
  fixup(elt);
  var name = $(elt).attr("data-canvas-chart");
  var ctx  = elt.getContext("2d");
  var chart = [];
  var max   = 100;
  $(document).on('data:'+name,function(event,data) {
    if(data > max) max = data;
    chart.push(data);
    if(chart.length > maxValues) chart.shift();
    drawChart(ctx,chart,max);
  });
});
```

Figure 7.6

Although this example is low level, it shows how easy it is to render simple charts using the canvas elements. This has just been a taste of the canvas element; there are many more drawing tools that can be used to make visualizations generated with the element more attractive.

Using Inline SVG

Unlike the canvas element, Scalable Vector Graphics (SVG) uses a DOM very similar to HTML to define the image to be rendered. As a standard, SVG is relatively ancient, having been around since 1999. Nearly every browser supports it; Internet Explorer's support lags (Microsoft had submitted a competing standard called Vector Markup Language (VML) to the W3C), but it has provided at least some level of support since version 9.

Unlike canvas, which is resolution dependent, SVG is a resolution-independent standard that provides crisp rendering at every resolution. This is particularly useful on mobile devices where it is common for users to zoom in and out of sections of an interface to provide larger touch targets.

Because it is included as part of the DOM, inline SVG is also usually more convenient than canvas images because it can be styled using cascading style sheets (CSS) in the same way as other elements. This makes it easier to achieve a consistent style in the interface or to collaborate with designers.

The document model itself is reflective of its origins, which were heavily influenced by the PostScript drawing model and to some extent the VML introduced by Microsoft as an alternative. There is a small set of primitive objects: `<circle>`, `<rect>`, `<polygon>`, and `<path>` elements that are combined organizational elements such as grouping (`<g>`) elements to construct an image. SVG also has its own set of `<text>` elements, though they do not typically have the level of control in layout as those offered by HTML.

Using these elements, the gauge visualization from the previous section can be constructed statically in the `mixin` found in the `jade` files in `dashboard/views`:

```
mixin metric(id,title,footer)
  div(class="panel panel-default",id="#{id}-metric" )
    if title
      div.panel-heading: h3.panel-title= title
    div(class="panel-body metric")
      div(class="canvas-wrap")
        svg(data-counter-gauge="#{id}")
          path(class="back")
          path(class="front")
      div(class="value-wrap")
        h1(data-counter="#{id}")= "--"
        h4
          span.glyphicon(data-direction="#{id}")
          span(data-change="#{id}")= ""
    if footer
      div.panel-footer= footer
```

The path elements inside of the `<svg>` element are then styled in CSS to produce the desired effect:

```
path.back {
  fill: none;
  stroke: #ccc;
  stroke-width: 25px;
}
path.front {
  fill: none;
  stroke: #ccc;
  stroke-width: 25px;
}
```

When the time comes to draw the arc, the `d` attribute of each path element is modified appropriately. The `back` element, unlike the canvas case, only needs to be drawn once with maximum length. The front element is modified each time the value is updated by this function in

public/javascripts/svg.js:

```
$('*[data-counter-gauge]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-gauge");
  drawArc($(elt).children(".back"),dim,1);
  var front = $(elt).children(".front");
  var max   = 0;
  $(document).on('data:'+name,function(event,data) {
    if(data > max) max = data;
    drawArc(front,dim,data/max);
  });
});
```

The primary work is in the computation of the arc in the path element itself. This is implemented in public/javascripts/svg.js using a bit of trigonometry via the drawArc function:

```
var mid = (2.4-0.6)*0.56 + 0.6;
function drawArc(elt,dim,pct) {
  var cx = dim.width/2;
  var cy = dim.height/2;
  var r  = Math.min(cx,cy) - 12.5;
  var angle  = ((2.4-0.6)*pct + 0.6)*Math.PI;
  var sx = cx + r*Math.cos(0.6*Math.PI);
  var sy = cy + r*Math.sin(0.6*Math.PI);
  var ex = cx + r*Math.cos(angle);
  var ey = cy + r*Math.sin(angle);
  elt.attr("d",["M",sx,sy,"A",r,r,0,
    angle <= mid*Math.PI ? 0 : 1,1,
    ex,ey].join(" "));
}
```

Like many scene description languages, SVG is not particularly designed for ease of use. This leads to functions like the one presented in this section, which is the result of the fact that the arc-drawing commands in SVG are more powerful than most users will ever need. This makes using SVG directly in documents inconvenient, though it can be made simpler through the use of JavaScript libraries such as the one described in the next section.

Data-Driven Documents: D3.js

Hand-coding visualizations rapidly becomes tedious, especially when the rendering gets complex. As seen in the last section, even apparently simple constructions can result in having to perform a variety of calculations and difficult-to-remember decisions.

The Data-Driven Documents JavaScript library (known as D3.js, or sometimes just D3) seeks to simplify much of this computation by providing a rich set of primitives for manipulating visualizations. In addition, it provides a mechanism for conveniently binding data to these visualizations to allow for easy construction of examples like those developed in the previous sections.

This section introduces the basics of the D3.js library and its usage. This includes basic manipulation and rendering, layouts, and some of the higher-level constructs available in D3.js, such as axes and maps. The toolkit also includes facilities for interacting with visualizations, but that is beyond the scope of this book.

Selecting, Inserting and Removing Elements

Like jQuery and other JavaScript toolkits, the core feature of D3 is the *selection*. In fact, the selector language used by D3 is very similar to that used by jQuery. For example, to select all of the `<div>` elements in a document, you use `d3.selectAll("div")`. To select a single element such as the `<body>` element, `d3.select("body")` is used. Also, like jQuery, `d3.selectAll("div.blue")` would select all `<div>` elements with a class attribute containing `blue` whereas `d3.select("div#blue")` would select a `<div>` element with an `id` attribute of `blue`. These operations can be chained together so that a `selectAll` called after a `select` would find all of the elements that are children of the original select.

After selecting an element, the most common operation is to add another object as a child of it. For example, this function found in `dashboard/public/javascripts/d3.js` initializes the dashboard; after fixing the position of the wrapper class, D3 can be used to create an `<svg>` element that can be used for further rendering:

```
$('*[data-counter-gauge]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-gauge");
  var svg = d3.select(elt).append("svg");
```

Elements can also be inserted using the `insert` method, which takes a `before` statement in addition to the element to be inserted. For example, to insert an element at the beginning of the child list rather than appending to the end, use `insert("<div>",":first-child")`.

Finally, elements that have been selected can be removed from the document using the `remove()` method.

Attributes and Styling

After an element has been selected, it can be modified using the `attr` and `style` methods of the selection. In the simple case, these methods work similarly to their jQuery counterparts and modify the appropriate aspects of the element. In the gauge example, they are used to establish the appropriate class for the front and back gauge objects:

```
var g = svg.append("g")
  .attr("transform",
    "translate("+(dim.width/2)+","+(dim.height/2)+")");
var back  = g.append("path").attr("class","d3back");
var front = g.append("path").attr("class","d3front");
```

In this case, the path elements are added to a `<g>` element rather than directly to the `<svg>` element. This grouping element allows transformations, such as a translation, to be easily applied to all groups. The translation is used in this case to center the paths so that a shape generator can be applied to each path.

Shape Generators

In D3, path elements like those in the previous section are usually combined with a shape generator rather than described by hand. These generators are used to set the `d` attribute easily. There are a variety of built-in path generators:

- `d3.svg.path.line` generates simple line paths.
- `d3.svg.path.area` generates areas instead of lines like the chart example in the "Using the HTML5 Canvas" section.
- `d3.svg.line.radial` generates a line path with radial coordinates.

- `d3.svg.arc` generates arc paths. These are often used for pie and donut charts.
- `d3.svg.symbol` is used to generate symbols at specific locations, such as those found in a scatterplot.
- `d3.svg.chord` is used to generate closed shapes connecting two arcs with a quadratic Bézier curve.

All of the shape generators return a function that can be customized with generator-specific parameters. These parameters can be set to either a constant value or a function that knows how to produce output based on the input.

For example, to create the gauge from the previous section, the `d3.svg.arc` generator can be used to simplify the process. This generator has four different parameters that can be potentially set from the data: the inner and outer radii, the starting angle, and the ending angle. In this case, the first three are all constant values:

```
var r     = 0.5*Math.min(dim.width,dim.height) - 12.5
var arc   = d3.svg.arc()
  .innerRadius(r-12.5).outerRadius(r+12.5)
  .startAngle(1.1*Math.PI);
```

The final parameter, the ending angle, depends on the data being passed in, so it is assigned a function rather than a constant value:

```
var max = 1;
arc.endAngle(function(d,i) {
  if(max < d) max = d;
    return 1.1*Math.PI + 1.8*Math.PI*(d/max);
});
```

With the maximum value set to `1`, the background element can be drawn a single time, and then the foreground element can be redrawn each time a new value is made available:

```
back.attr("d",arc(1));
$(document).on('data:'+name,function(event,data) {
  front.attr("d",arc(data));
});
```

In addition to being easier to implement, the paths produced by the arc generator (and other generators where appropriate) are regions rather than lines. This means that they can have separate fill and stroke styling whereas the previous examples used the stroke width to simplify the rendering.

Joining Selections and Data

When the data is simple, using D3 directly is an easy way to develop quick-and-dirty visualizations. When things get more complicated, D3 really comes into its own as the power of its join operations can be combined with its selection operators.

To join an array of data to a selection, the `data` method of the selection is used. The resulting join is comprised of three subselections:

- **The update selection:** This is the set that contains elements in the document that already correspond to elements of the data array.
- **The enter selection:** This is the set of data array elements that do not (yet) correspond to document elements.
- **The exit selection:** This is the set of document elements that have no corresponding element in the

data array.

The update selection is the default selection returned by the `data` method. The other two selections are accessed with the `enter()` and `exit()` methods, respectively.

The usual order is to first work with the enter selection. This selection is special in two ways. First, it only affects the document manipulation methods such as `append`. Secondly, when the `append()` or `insert()` methods are used, the resulting elements are immediately added to the update selection. This reduces code duplication by allowing all of the attributes and styles to be modified on the update selection. If, for some reason, the styling should be different for newly created elements (perhaps to highlight them), simply modify the update selection before using the enter selection.

The exit selection is usually the least interesting of the selections because the only operation is typically to `remove` elements from the document. This is sometimes combined with an animation of some kind—which is discussed later in the "Animation" section—to make the effect less jarring.

Implementing a bar chart is a good way to see these selections in action. In this case, the `mixin` is adjusted slightly to apply the `data-counter-bar` attribute. Because this has been done several times in this chapter, the code is omitted for this example. Like the gauge element, each bar chart element has an `<svg>` element appended and, in this case, is given a class to make it easier to style for the bar chart:

```
$('*[data-counter-bar]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-bar");
  var svg   = d3.select(elt).append("svg")
  .attr({
    class:"barchart",
    width:dim.width,
    height:dim.height
  });
```

Like the line chart example, an array of elements is maintained when new data is entered. This is what will be bound to a selection and rendered into a bar chart:

```
  var dataLen= 30, max = 0;
  var values = [];
  $(document).on('data:'+name,function(event,data) {
    if(data > max) max = data;
    values.push(data);
    if(values.length >= dataLen) values.shift();
```

This data array is then joined with a selection containing all of the rectangles. A new rectangle is added to the selection (if needed), then all of the rectangles are updated to their appropriate location and height. The width is set when the rectangle is added because it does not change during the course of visualization. This results in a dashboard, implemented in `dashboard/public/javascripts/d3.js` and `dashboard/views/d3_ex1.jade` like the one shown in Figure 7.7:

```
[[OPEN-LW-CODE80]]      var update = svg.selectAll("rect").data(values);
    //Add a new rectangle for each piece of data
    update.enter().append("rect").attr("width",dim.width/(dataLen-1))
    update
    .attr("x", function(d,i) { return i*dim.width/(dataLen-1); })
    .attr("y", function(d) { return dim.height - dim.height*(d/max); })
    .attr("height", function(d) { return dim.height*(d/max); })
```

```
    });
[[CLOSE-LW-CODE80]]
```

Figure 7.7

To re-create something like the line chart from the original canvas examples, a different approach is required. In this case, a single element (the path) should be bound to the entire data array. The only time there would be multiple path elements would be if there were multiple series to be rendered. In this case, the `datum` method can be used to bind the entire array as a single element.

Scales and Axes

The previous examples have devoted a great deal of their code to the appropriate transformation from the input domain to the output range. To simplify this process, D3 has a number of built-in scale functions:

- `d3.scale.linear` performs a linear transformation from the input domain to an output range. This is the most commonly used scale.
- `d3.scale.sqrt`, `d3.scale.pow`, and `d3.scale.log` perform power transformations of the input domain to the output range.
- `d3.scale.quantize` converts a continuous input domain to a discrete output range.
- `d3.scale.identity` is a simple linear identity scale.
- `d3.scale.ordinal` converts a discrete input domain to a continuous output range.
- `d3.scale.category10`, `d3.scale.category20`, `d3.scale.category.20b`, and `d3.scale.category.20c` construct an ordinal scale that converts to 10 or 20 color categories.

Mostly, these scales are used to simplify range calculations. For example, the bar chart example from the previous section can be further simplified by using linear scales for the x and y coordinates. The x scale is static, having an input domain between 0 and the number of possible elements in the array (in this case 30). The y scale can change as data is added, so there is a check that potentially updates the domain on each step:

```
var y = d3.scale.linear().domain([0,1])
  .range([0,dim.height]);
var x = d3.scale.linear().domain([0,dataLen-1])
```

```
    .range([0,dim.width]);
  $(document).on('data:'+name,function(event,data) {
    if(data > y.domain()[1]) y.domain([0,data]);
```

The calculations in the update step can then be updated with calls to the x and y scales:

```
  update
    .attr("x", function(d,i) { return x(i); })
    .attr("y", function(d) { return dim.height - y(d); })
    .attr("height", function(d) { return y(d); });
```

Another use for scales is to simplify adding axes to D3.js visualizations. Like most other D3 elements, axes are managed using a generator that operates on either the `<svg>` element or a `<g>` element. The axis function adds a variety of new elements to the document, including lines to represent the tick marks and the tick values themselves. The easiest way to use axes is with the grouping element because it makes it easier to move them around within the graphic. To get started, the scales need to be adjusted slightly to make space for the axes to be rendered:

```
  var y = d3.scale.linear()
    .domain([0,1])
    .range([dim.height-20,0]);
  var x = d3.scale.linear()
    .domain([0,dataLen-1])
    .range([60,dim.width-10]);
```

Notice that the y-axis scale has had its output range reversed. This is due to the fact that the (0,0) coordinate is actually in the lower-left corner rather than the upper-left corner. This also means that the "height" and "y" attributes must be reversed when drawing the bars. Because the x-axis is static, it can be rendered immediately into a grouping element that is translated to the bottom of the visualization:

```
  svg.append("g")
  .attr("class","x axis")
  .attr("transform","translate(0,"+(dim.height-20)+")")
  .call(d3.svg.axis()
    .orient("bottom")
    .scale(x)
  );
```

The y-axis is similarly defined, but it needs to be updated whenever the domain changes so it is not immediately applied to its grouping element:

```
  var yg    = svg.append("g")
  .attr("class","y axis")
  .attr("transform","translate(60,0)");
  var yaxis = d3.svg.axis()
    .ticks(5).orient("left")
    .scale(y);
```

Whenever the domain is updated, the y-axis is redrawn to reflect these changes:

```
  if(data > y.domain()[1]) {
    y.domain([0,data]);
    yg.call(yaxis);
  }
```

After applying a little bit of CSS styling, the bar chart should look like the one in Figure 7.8, implemented in `dashboard/views/d3_ex2.jade`:

```
.axis text { font: 10px sans-serif; }
.axis path, .axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
```

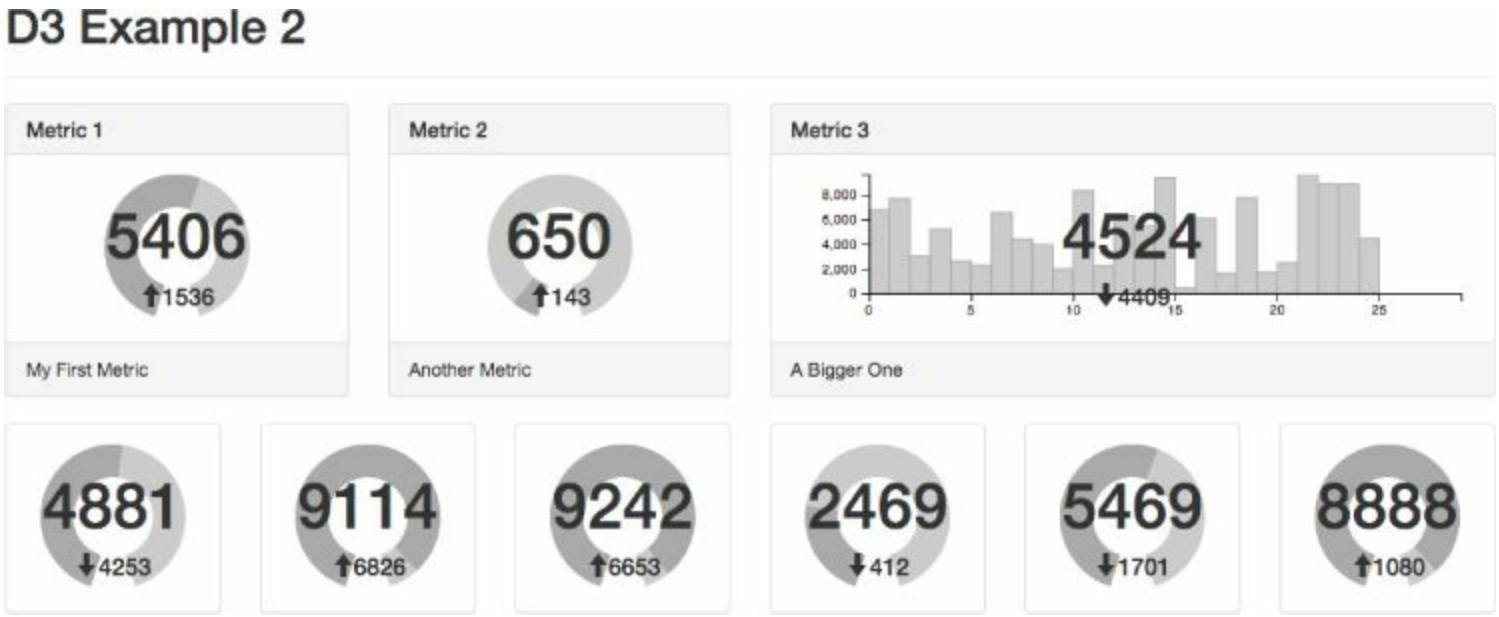


Figure 7.8

Layouts

Many times, when data is to be visualized, it requires some level of manipulation to make it suitable for rendering. Even something as simple as a pie chart requires that the starting and ending angles of each category be calculated.

To help with these more complicated visualizations, D3 offers a number of layout generators. Rather than generating document elements directly, like the path and axis generators, these are usually applied to the data and then used to construct the visualization.

There are numerous plug-ins for D3 that implement various layouts, but there are also several built-in layouts. Many of them are concerned with the layout of hierarchical or graph structures. These are useful when data is static, but they are not used very often for real-time data visualization, so they will not be discussed in this section.

The layouts most often relevant to real-time data visualization are the pie and stack layouts. These two layouts help build pie or donut charts and stacked area charts, respectively. For example, to build something that looks like Figure 7.9, start out by constructing some sample data:

```
var values = d3.range(10).map(function() {
    return bumpLayer(100);
});
```

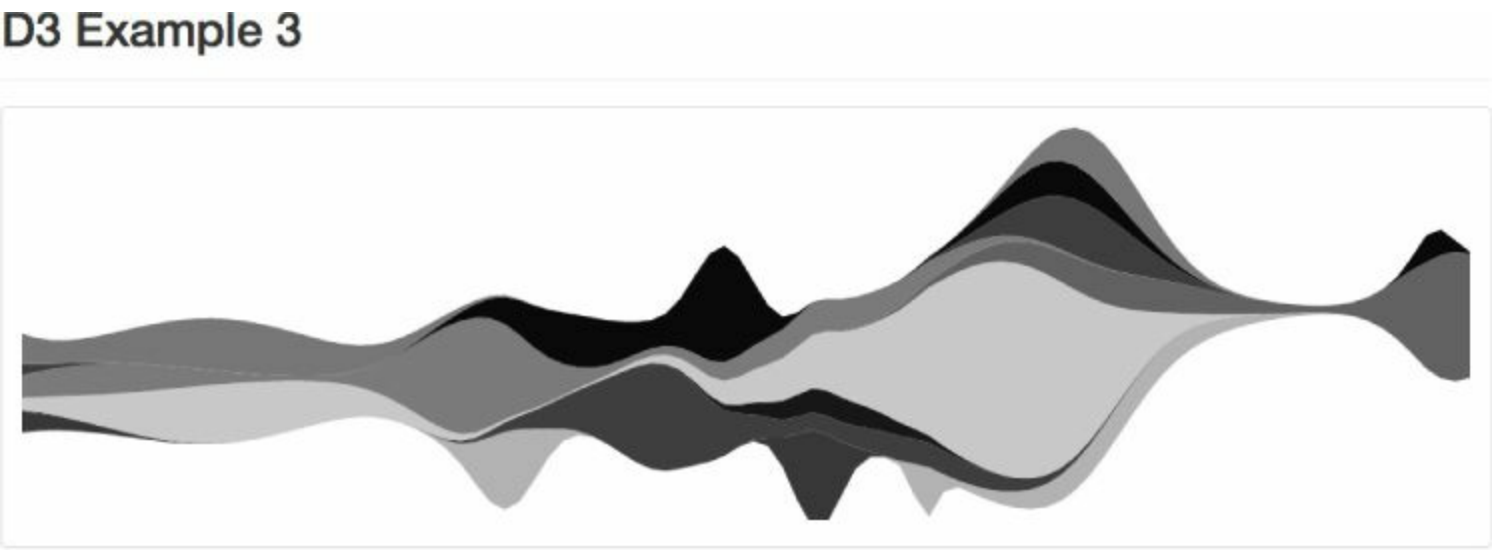


Figure 7.9

The `bumpLayer` function is used in various locations in the D3 documentation to generate test data. It is used here to quickly provide data for this example. Next, the data is stacked using the `stack` layout, and the maximum value calculated so that the scales may be obtained:

```
var stack = d3.layout.stack()
  .offset("wiggle");
values = stack(values);
var max = 0;
for(i in values) {
  var m = d3.max(values[i],function(d) { return d.y + d.y0; });
  if(m > max) max = m;
}
```

Next, the scales for the x-axis, the y-axis and the color range are constructed. An `area` path generator is used to render the output of the `stack` layout:

```
var x     = d3.scale.linear().domain([0,99]).range([0,width]);
var y     = d3.scale.linear().domain([0,max]).range([height,0]);
var color = d3.scale.linear().range(["#000","#ccc"]);
var area  = d3.svg.area()
.x(function(d) { return x(d.x); })
.y0(function(d) { return y(d.y0); })
.y1(function(d) { return y(d.y0 + d.y); });
```

Finally, the shape generator and the layout data are applied to a path selection to produce the final output implemented in `dashboard/views/d3_ex3.jade`:

```
svg.selectAll("path").data(values)
  .enter().append("path")
  .attr("d",area)
  .style("fill",function(d) { return color(Math.random()); });
```

Animation

The final piece of the D3 puzzle is its animation facilities. The ability to easily add animation often adds that little bit of context that separates a good interface from a great one.

In general, animation should be used sparingly to either acquire or remove attention depending on the situation. For example, an animation on the changing digits of the metrics in the dashboard draws attention to the change because their rapid change is relatively unnoticeable to the eye. However, if

the gauge itself is allowed to "jump," it tends to be distracting. To combat this distraction and allow the focus to be on the new value, a simple "tweening" animation is used. This smooths the motion of the gauge over time, which tends to be less distracting.

For simple animations, nearly no work is required. For example, animating the bar chart from the earlier examples only requires the addition of a `transition` function to provide a smooth interpolation between elements:

```
update
  .transition()
    .attr("x", function(d,i) { return x(i); })
    .attr("y", function(d) { return y(d); })
    .attr("height", function(d) { return (dim.height-20) - y(d); });
```

There are a variety of features of the transition, such as the easing method or the duration that can be specialized, but this is all that is required for many visualizations.

In rare cases, a more complicated transition is required. As it happens, the gauge example is one of those cases. If a transition were applied directly to the gauge, the interpolation process would occur in the Cartesian coordinate system of the page, rather than the polar coordinate system of the gauge. To modify the gauge display to allow for animation, first the paths must be modified to use a data element:

```
var back  = g.append("path").datum({endAngle:1})
 .attr("class","d3back")
 .attr("d",arc);
var front = g.append("path").datum({endAngle:0})
 .attr("class","d3front")
 .attr("d",arc);
```

This is done because the interpolation process needs to be able to modify the `endAngle` variable. The interpolation itself is handled by repeatedly applying an interpolation using `attrTween`. This is implemented in the `arcTween` function, found in `dashboard/public/javascripts/d3.js`:

```
function arcTween(transition,newAngle) {
  transition.attrTween("d",function(d) {
    var interpolate = d3.interpolate(d.endAngle,newAngle);
    return function(t) {
      d.endAngle = interpolate(t);
      return arc(d);
    }
  });
}
```

This function repeatedly computes the interpolated angle and then calls the arc generator on that interpolated angle to produce a new function. Without the `datum` call, the *d* variable would be undefined and there would be no place to store the outcome for the next interpolation. The update step then only needs to call the `arcTween` function with the new angle when it arrives:

```
$(document).on('data:'+name,function(event,data) {
   if(data > max) max = data;
   front.transition().call(arcTween,data);
});
```

High-Level Tools

Although D3 is extremely powerful, it is also extremely granular. Even with some of the abstractions available, it can be time consuming to construct basic visualizations. Others have also run into this problem and built high-level packages that build on D3 to allow for the rapid development of simple visualizations. This section covers two of the more interesting packages. Although none of them cover every situation, they can help you get a jump start on building dashboard visualizations. Most people will find the first package, NVD3, more accessible out of the gate, but the second package, Vega, represents an interesting approach to separating rendering, business logic, and data.

NVD3

NVD3 has been around for a long time and is still under active development, with version 1.1.10 in beta at the time of writing. It is the back-end library of the "dashing" dashboard library, which is a simple streaming dashboard framework from Shopify written in Ruby using the Sinatra framework (rather than Node).

Figure 7.10 shows the bar chart example from the previous section using NVD3 instead of D3 directly. It also shows a similar visualization using a line chart instead of a bar chart. To build the bar chart, the NVD3 chart is first defined and then attached to the SVG element, found in `dashboard/public/javascripts/nv.js`:

```
$('*[data-counter-bar4]').each(function(i,elt) {
  var dim = fixup(elt)
  var name = $(elt).attr("data-counter-bar4");
  var chart = nv.models.historicalBarChart()
  .margin({left:50,bottom:20,right:20,top:20})
  .x(function(d,i) { return i; })
  .y(function(d,i) { return d; })
  .transitionDuration(250);
  chart.showXAxis(true);
  chart.yAxis
  .axisLabel("Value");
  var values = [];
  var dataLen   = 30;
  d3.select(elt)
  .append("svg")
  .datum([{values:values,key:"Data",color:"#ccc"}])
  .transition().duration(0)
  .call(chart);
```

The `datum` command is used because NVD3 works with series rather than data, much like shape generators. Using series also allows for the definition of a series title and a specific series color as shown in the preceding code. The values are initially blank, but because JavaScript is a reference-based language the array can be updated later without having to reset the data element. After the chart is defined, updating it when the data array is modified is as simple as calling the `update` command:

```
$(document).on('data:'+name,function(event,data) {
  values.push(data);
  if(values.length >= dataLen) values.shift();
    chart.update();
});
```

NVD3's interfaces are fairly standard, which makes switching chart types a breeze. To add the line chart visualization instead of a bar chart visualization to Figure 7.10, a single line of implementation was changed:

```
  var chart = nv.models.lineChart()
```

Figure 7.10

The data series for the line chart has an optional `area` parameter that can be used to make the line chart an area chart. This standardization of parameters makes it extremely simple to experiment with different forms of visualization. It is also trivial to render multiple series in the same chart by simply modifying the `datum` statement.

Although NVD3 can clearly handle streaming data, it is mostly designed for static interactive visualizations. As such, it includes a number of additional interactive features, such as tooltips and zooming/focusing features that are commonly found in interactive visualization environments. Most real-time environments are not interactive, so these features are not discussed in this chapter.

Vega.js

The Vega package builds on the idea of having a description of a visualization that easily can be changed. Rather than using code to describe the visualization the way NVD3 does, this package opts to use a JSON description of the visualization. The Vega developers call this format a "declarative visualization grammar."

Vega visualizations begin with a specification that provides the abstract visualization of the chart. Specifications are made up of several different pieces. The first piece is usually some information about rendering the chart. Because the chart is often reused the information is usually fairly minimal, if it exists at all. In this case, some margin information is common to all the charts:

```
var spec = {
  padding: {top: 10, left: 30, bottom: 20, right: 10},
```

Next comes the data definition section of the specification. This section defines the various data series that will be used in the visualization. Because this data comes from a real-time system and will be added later, the data for the series is simply left empty:

```
 data:[
  {name:"values"}
  ],
```

Finally, the visualization is described. Much like the low-level D3 charts, the specification defines scales as axes using parameters very much like the D3 generators they use internally:

```
    scales:[
    {name:"x",range:"width",
        domain:[0,29] },
    {name:"y",range:"height",nice:true,
        domain:{data:"values",field:"data"}},
    ],
    axes:[
    {type:"x",scale:"x"},
    {type:"y",scale:"y"}
    ],
```

The primary element of note in this last section is the linking of the data series to the scale in the `domain` element. This structure is called a `DataRef`, and it usually consists of a data series, identified by the `data` element, and a field accessor that is defined by the `field` element. When data is added to Vega an array that looks like this:

```
[0,1]
```

It is translated into a "Vega compatible" format that looks like this:

```
[{index:0,data:0},{index:1,data:1}]
```

The field accessors operate on this transformed data and expect the inclusion of the `index` and `data` fields to operate correctly. More complicated data structures can be accessed using the usual JavaScript dot notation.

Defining "marks" in the specification specifies the type of chart drawn for a given series. Like NVD3, defining multiple marks draws multiple elements within a single chart. In this case, only a `rect` mark is rendered to draw the bar chart:

```
    marks:[{
      type:"rect",
      from:{data:"values"},
      properties:{
        enter:{
          fill:{value:"#ccc"},
          stroke:{value:"#aaa"}
        },
        update:{
          x:{scale:"x",field:"index"},
          y:{scale:"y",field:"data"},
          y2:{scale:"y",value:0},
          width:{scale:"x",value:1}
        }
      }
    }]
  };
```

Again, notice the `DataRef` used to bind parameters to the data. Also notice that the `scale` element is used to associate scales with parameters. Otherwise, mark definitions should look familiar, as they are essentially a JSON representation of the D3 commands used earlier.

To use the specification, it must first be parsed into a `chart` object. This object is returned in a callback because it is possible to pass a URL to the parser rather than passing a JSON object directly. This allows specifications to be stored separately from the code. After the `chart` object is available, it may be executed to produce a `view` object that is used to render the scene. Vega has two default renderers, a canvas-based renderer and an SVG-based renderer. In this case the SVG rendered is used, but the default is the canvas renderer:

```
vg.parse.spec(spec,function(chart) {
  $('*[data-counter-bar4]').each(function(i,elt) {
    var dim = fixup(elt)
    var name = $(elt).attr("data-counter-bar4");
    var view = chart({el:elt})
    .renderer("svg")
    .width(dim.width-60)
    .height(dim.height-30)
    ;
    var values = [];
    var dataLen   = 30;
    var x      = 0;
    $(document).on('data:'+name,function(event,data) {
      values.push(data);
      if(values.length >= dataLen) values.shift();
      view.data({values:values}).update();
    });
  });
});
```

In this code, the dynamic data feature of Vega is also used. This method of the `view` object takes an object with keys that correspond to tables defined in the specification. The value of those keys overrides the corresponding values key in the table definition of the view object. The end results are charts like the examples in Figure 7.11 and implemented in `dashboard/public/javascripts/vega.js`.

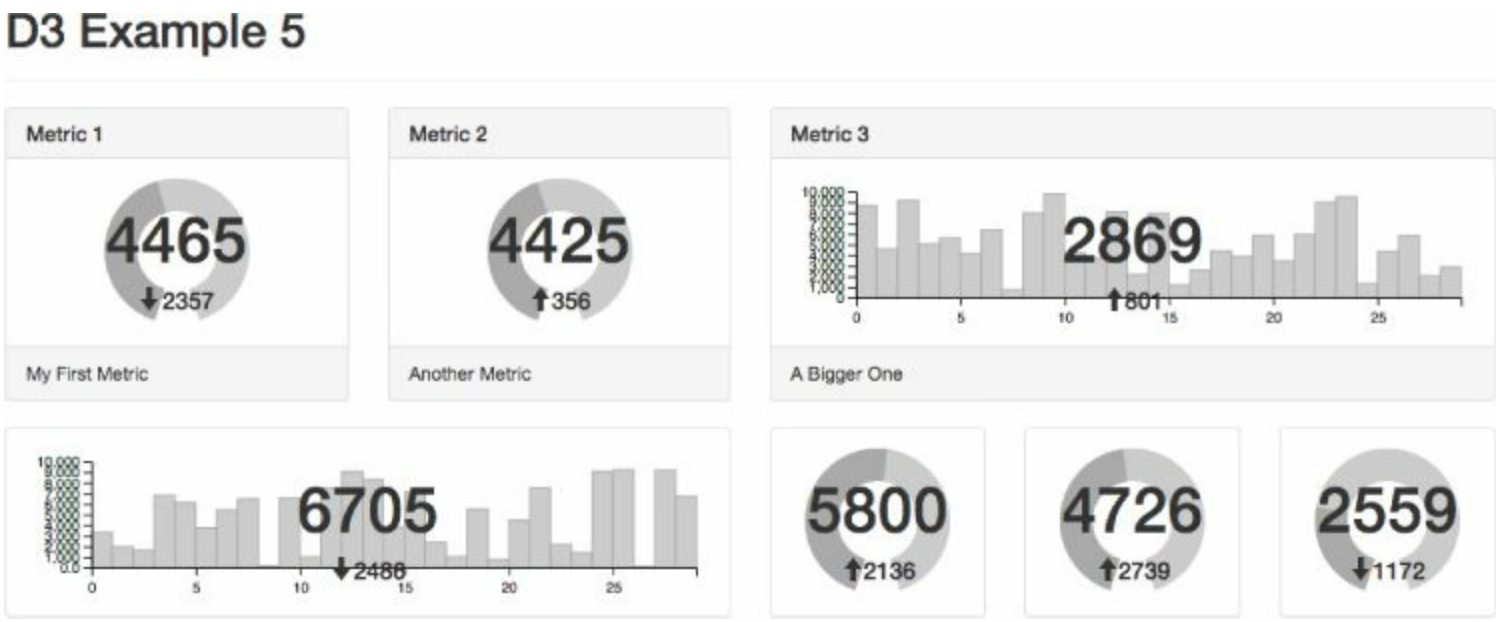


Figure 7.11

Mobile Streaming Applications

More and more of the world's application are becoming mobile in one sense or another. Streaming data applications are no exception to this trend, though they tend to be more useful on the larger tablet formats than the smaller phone formats.

The infrastructure developed in this chapter has actually been developed with mobile environments. For data transfer, the biggest killer of battery life is applications that heavily use Wi-Fi or cellular radio activity. Polling approaches seem to exercise the radios pretty heavily. SSEs and WebSockets, empirically, appear to have relatively little effect on battery life. In addition, both protocols automatically reconnect, making them good choices for the relatively unsure environment that is mobile communication.

On the rendering front, the Bootstrap 3 framework used to develop the dashboards is a “mobile first” framework and has built-in “responsive” features. This means that on phones and tablets the dashboards reorganize themselves into a format that maximizes available space in the mobile format. Figure 7.12 shows this responsive rearrangement for the example dashboard.

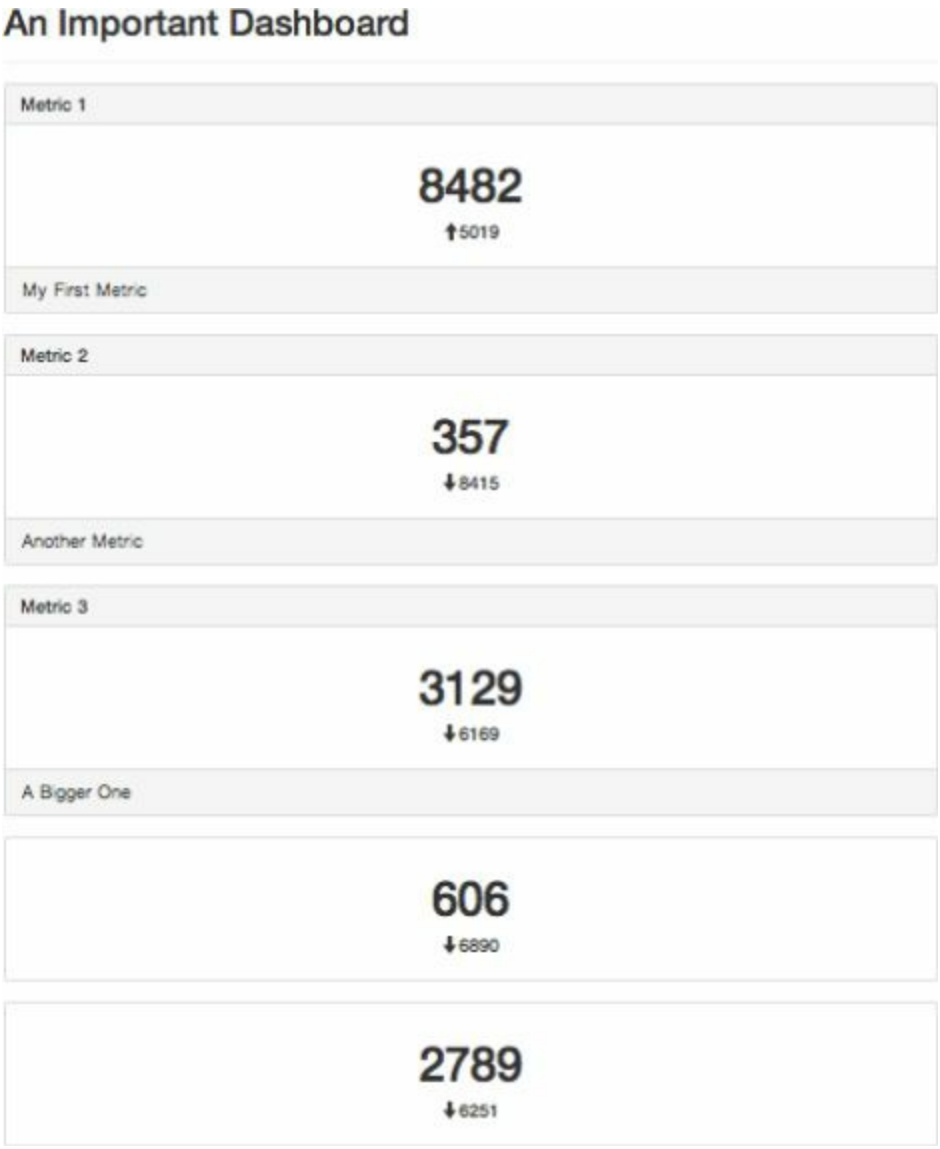


Figure 7.12

For visualization rendering, Canvas and Inline SVG are supported on all major mobile browsers. Even Mobile Internet Explorer and the BlackBerry browser support SVG. The vector-based

rendering also makes SVG an excellent choice for dealing with the high resolutions encountered in modern tablets and phones (as well as having good zoom behavior). The only catch is that SVG, unlike Bootstrap, is not naturally "responsive." Figure 7.13 demonstrates what happens when the SVG is not adjusted after resizing.

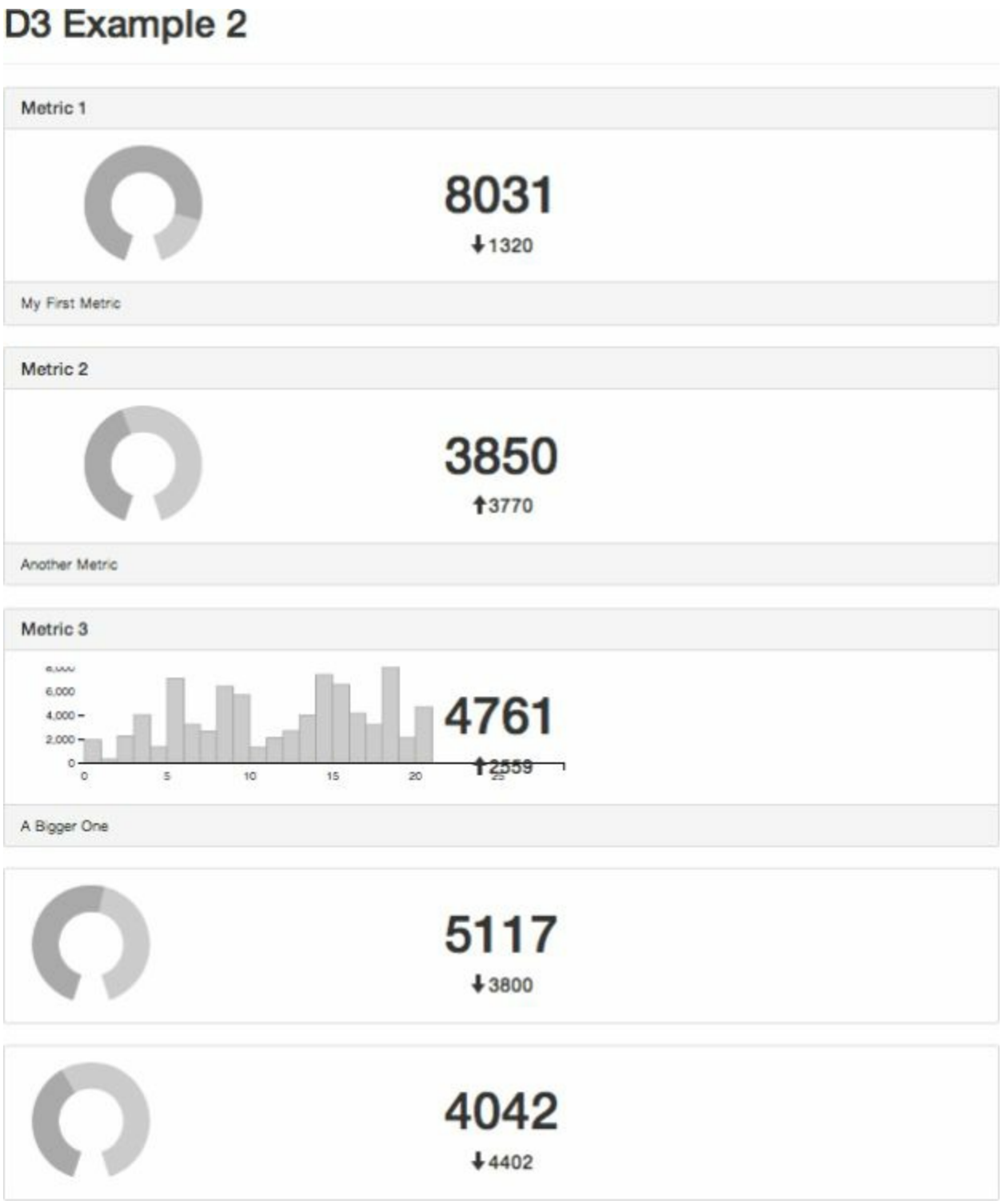


Figure 7.13

The image is still being drawn into the original viewport, which now does not match the dimensions of the panel. To alleviate this problem, the resize event must be captured and used to update elements that depend on the size of the element. For example, to update the gauge element on a resize requires several changes to each element:

```
$(window).resize(function() {
```

```
    dim = fixup(elt);
    g.attr("transform",
      "translate("+(dim.width/2)+","+(dim.height/2)+")");
      r = 0.5*Math.min(dim.width,dim.height) - 12.5;
      arc.innerRadius(r-12.5).outerRadius(r+12.5);
      back.transition().duration(0).call(arcTween,max);
    });
```

In this case, the canvas itself is resized and the new dimensions obtained from the `fixup` function. Next, the arcs are translated to their new position. Because the element may have changed its smallest dimension, the inner and outer radius must be recalculated. Finally, the backing arc is redrawn in its new position. The front arc could also be redrawn, but it is updated often by the streaming data so it is left alone in this case.

This is actually a bit easier for Canvas devices. Because a canvas rendering needs to be redrawn completely on each pass, the resize event can simply set a flag that tells the canvas to update its size on the next redraw.

Conclusion

This chapter has covered the basics of setting up a web-based application for delivering real-time data. The next chapter builds on these concepts to add the element of time to both the data and the visualizations.

Beyond that, the remainder of this book does not really touch on the mechanics of delivery and visualization. That is not to say that there is no more to say on the subject. For the visualization component in particular, this chapter barely scratches the surface of what is possible with these frameworks. The only limit is the imagination, as they saying goes, and there is plenty of inspiration out there. The first place to look would be Mike Bostock's website (http://bost .ocks.org/mike/), which has no shortage of in-depth D3 tutorials. His website is particularly useful when you're learning how to visualize geographical data because he clearly has a special love for cartography and D3's geographic support is impressive. The Flowing Data website (http://flowingdata.com) is also a great source of visualizations. Not all of them will be appropriate for real-time streaming data, but the techniques are certainly applicable.

Chapter 8
Exact Aggregation and Delivery

Now that the infrastructure is in place to collect, process, store, and deliver streaming data, the time has come to put it all together. The core of most applications is the aggregation of data coming through the stream, which is the subject of this chapter.

The place to begin is basic aggregation—counting and summation of various elements. This basic task has varying levels of native support in the popular stream processing frameworks. Probably the most advanced is the support provided by the Trident language portion of Storm, but that assumes a willingness to work within Trident's assumptions. Samza has some support for internal aggregation using its `KeyValueStore` interfaces, but it leaves external aggregation and delivery to the user to implement. Basic Storm topologies, of course, have no primitives to speak of and require the user to implement all aspects of the topology. This chapter covers aggregation in all three frameworks using a common example.

Most real-time analysis is somehow related to time-series analysis. This chapter also introduces a method for performing multi-resolution aggregation of time series for later delivery. Additionally, because the data are being transported via mechanisms that can potentially reorder events, this multi-resolution approach relies on record-level timestamps rather than simply generating results on a fixed interval.

Finally, after data has been collected and aggregated into a time series, it must be delivered to the client. Using the techniques introduced in the last chapter, time-series data can be delivered to the client for rendering. There are a variety of rendering options for time-series data, but the most popular is the simple strip chart because it's easy to understand. An alternative to the strip chart is the horizon chart, which allows for many correlated time series to be displayed simultaneously. The standard rendering techniques used can sometimes be too slow when the data delivery rate is high. At the expense of flexibility, high-performance techniques can be used to provide charts with very high update rates.

The Wikipedia Edit Stream

Wikipedia makes its edit stream available through an Internet Relay Chat (IRC) channel for the entire world to consume as it sees fit. There are always a fairly high volume of edits being performed on Wikipedia at any given moment, making it a good source of "test" data when learning to use real-time streaming applications. This data stream is used as a working example throughout this chapter.

To get the data, an application is needed to watch the IRC data stream. Fortunately, the Samza project from Chapter 5, "Processing Streaming Data," includes a Wikipedia IRC reader that it then streams into a Kafka queue. It is included as part of the Hello Samza project. To get this running, first check out the introductory code from Github and start the included Samza grid:

```
$ git clone https://github.com/apache/incubator-samza-hello-samza.git
$ cd incubator-samza-hello-samza/
$ ./bin/grid bootstrap
.. Output Removed ...
EXECUTING: start zookeeper
JMX enabled by default
Using config: /Users/bellis/Projects/incubator-samza-hello-
  samza/deploy/zookeeper/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
EXECUTING: start yarn
EXECUTING: start kafka
```

Samza is a fairly young project and the code base is moving quickly. As such, it is possible that the public repository has changed in such a way that the examples in this book no longer work. If this is the case, the code included with this book also includes a copy of both the `incubator-samza` and `hello-samza` projects at the time of writing. To use them, simply unpack the archive and copy `incubator-samza` project into the `samza` download directory. On most Unix-like systems, this will be the `~/.samza/download` directory. If all has gone well, the `./bin/grid bootstrap` command will work as described in this section.

This will bring up the YARN console in a local form, suitable for developing and testing the applications in this chapter. Next, build and deploy the first part of the Hello Samza project to start aggregating Wikipedia edits:

```
$ mvn clean package
... Omitted Output ...
 [INFO] Reactor Summary:
[INFO]
[INFO] Samza Parent ............................... SUCCESS [2.707s]
[INFO] Samza Wikipedia Example .................... SUCCESS [17.387s]
[INFO] Samza Job Package ......................... SUCCESS [1:24.473s]
[INFO] ------------------------------------------------------------------
[INFO] BUILD SUCCESS
  $ mkdir -p deploy/samza
  $ cd deploy/samza
  $ tar xvfz \
  > ../../samza-job-package/target/samza-job-package-0.7.0-dist.tar.gz
  $ ./bin/run-job.sh \
  > --config-factory=\
  > org.apache.samza.config.factories.PropertiesConfigFactory \
  > --config-path=\
  > file://`pwd`/config/wikipedia-feed.properties
```

If everything works properly there should be an application in the YARN console with a status of

RUNNING (by default this is available at http://localhost:8080). It should now also be possible to watch the raw edit streaming using the Kafka console consumer to watch the wikipedia-raw topic:

```
$ ../kafka/bin/kafka-console-consumer.sh \
> --zookeeper localhost --topic wikipedia-raw
```

After a few moments, the raw edit stream should start being output from the Kafka stream:

```
{"raw":"[[Special:Log/newusers]] byemail  * Callanecc *  created new
  account User:DeaconAnthony: Requested account at [[WP:ACC]],
  request #109207","time":1382229037137,
  "source":"rc-pmtpa","channel":"#en.wikipedia"}
{"raw":"[[User talk:24.131.72.110]] !N
    http://en.wikipedia.org/w/index.php?oldid=577910710&rcid=610363937
  * Plantsurfer * (+913) General note: Introducing factual errors on
  [[Prokaryote]]. ([[WP:TW|TW]])","time":1382229037656,
  "source":"rc-pmtpa","channel":"#en.wikipedia"}
```

By default, Hello Samza only reads from English language Wikipedia topics. This is a bit boring, so go ahead and stop the application and edit the wikipedia-feed .properties file to aggregate from the edits of several different languages. (This should all be on one line in the properties file. It is shown broken into multiple lines to account for formatting of the book.)

```
task.inputs=wikipedia.#en.wikipedia,
 wikipedia.#de.wikipedia,
 wikipedia.#fr.wikipedia,
 wikipedia.#pl.wikipedia,
 wikipedia.#ja.wikipedia,
 wikipedia.#it.wikipedia,
 wikipedia.#nl.wikipedia,
 wikipedia.#pt.wikipedia,
 wikipedia.#es.wikipedia,
 wikipedia.#ru.wikipedia,
 wikipedia.#sv.wikipedia,
 wikipedia.#zh.wikipedia,
 wikipedia.#fi.wikipedia
```

This results in much more interesting traffic to analyze later.

The Hello Samza application also includes a parser job that converts the raw JSON into a more useful form. Because Samza uses Kafka for its communication, this is available as another Kafka topic called wikipedia-edits. It is started the same way as the first Samza job, just with a different properties file:

```
$ ./bin/run-job.sh \
> --config-factory=\
> org.apache.samza.config.factories.PropertiesConfigFactory \
> --config-path=file://`pwd`/config/wikipedia-parser.properties
```

The output from this stream should look something like this:

```
{"summary":"/* Episodes */ fix",
 "time":1382240999886,
 "title":"Cheers (season 5)",
 "flags":{
  "is-bot-edit":false,
  "is-talk":false,
  "is-unpatrolled":false,
  "is-new":false,
```

```
   "is-special":false,
   "is-minor":false
  },
 "source":"rc-pmtpa",
 "diff-url":"http://en.wikipedia.org/w/index.php?diff=577930400&oldid=
   577930256",
 "diff-bytes":11,
 "channel":"#en.wikipedia",
 "unparsed-flags":"",
 "user":"George Ho"
 }
 {"summary":"General Fixes using [[Project:AWB|AWB]]",
  "time":1382241001441,
  "title":"Degrassi: The Next Generation (season 5)",
  "flags":{
   "is-bot-edit":false,
   "is-talk":false,
   "is-unpatrolled":false,
   "is-new":false,"is-special":false,
   "is-minor":true
  },
 "source":"rc-pmtpa",
 "diff-url":"http://en.wikipedia.org/w/index.php?diff=577930401&oldid=
   577775744",
 "diff-bytes":-4,
 "channel":"#en.wikipedia",
 "unparsed-flags":"M",
 "user":"ChrisGualtieri"
 }
```


Because Kafka is used for streaming data between Samza jobs, this simple introductory project is also useful for developing with other systems, such as Storm.

Timed Counting and Summation

The simplest form of counting and summation is the timed counter. In this case, any timestamp information associated with the event is ignored, and events are simply processed in the order that they arrive. This works best with collection mechanisms that do not make an attempt to reliably deliver data, meaning it is mostly applicable to monitoring applications.

These sorts of counters are also well suited to Lambda Architectures. In this architecture, there is a second process that will correct for any data loss or overcounting in the stream-processing environment. Under normal circumstances, the events coming out of something like Kafka may be disordered, but it's usually not enough to make a big difference. It is also relatively rare for events to be duplicated by the need to reread data.

In both cases, the basic counting systems described in the following sections can be used. They are easy to implement and require little overhead. However, they are not suitable for systems that need to be highly accurate or require idempotent operation.

Counting in Bolts

When Storm was first released, it did not contain any primitive operations. Any sort of counting operation was implemented by writing an `IRichBolt` that performed the counting task. To produce output over time, a background thread was typically used to emit counts on a fixed basis and then reset the local memory map. In newer versions of Storm (all versions beyond 0.8.0) the background thread is no longer needed as Storm can produce ticker events.

Implementing a basic aggregation bolt begins like any other bolt implementation. In this case, the bolt only takes a single parameter, which is the number of seconds to wait before emitting counts to the next bolt:

```
public class EventCounterBolt implements IRichBolt {
  int                    updates= 10;
  public EventCounterBolt updateSeconds(int updates) {
this.updates = updates;return this;
}
  public int updateSeconds() { return updates; }
```

The bolt maintains a transient map of elements and their count, which serves as the local store. Like all bolts, the collector is also captured in a transient variable. Both are initialized in the preparation method. This bolt outputs its counts every few seconds as configured in the preceding code, so an output stream must be declared. Finally, the tick events for this bolt are configured in `getComponentConfiguration`:

```
transient HashMap<String,Integer> counts;
transient OutputCollector      collector;
public void prepare(Map stormConf, TopologyContext context,
    OutputCollector collector) {
  this.counts    = new HashMap<String,Integer>();
  this.collector = collector;
}
public void declareOutputFields(OutputFieldsDeclarer declarer) {
declarer.declare(new Fields("timestamp","key","count"));
}
public Map<String, Object> getComponentConfiguration() {
Config conf = new Config();
```

```
    conf.put(Config.TOPOLOGY_TICK_TUPLE_FREQ_SECS, updates);
    return conf;
  }
```

Note that this is not a persistent key-value store, so if the bolt task dies for some reason, any interim data is lost.

The `execute` method gets two types of tuples. The first type of tuple is a tick event that is sent by Storm itself according to the earlier configuration. To detect this type of event the source component and the stream identifier of each tuple must be checked:

```
  public static boolean isTick(Tuple tuple) {
      return Constants.SYSTEM_COMPONENT_ID.equals(
        tuple.getSourceComponent()
      )
          && Constants.SYSTEM_TICK_STREAM_ID.equals(
        tuple.getSourceStreamId()
      );
  }
```

This is used to implement the `execute` method. If the `isTick` method returns `true` then the bolt should emit all of the values currently stored in the *`counts`* variable. Otherwise, it should increment or insert the value into the *`counts`* map. Finally, it acknowledges the tuple:

```
  public void execute(Tuple input) {
      if(isTick(input)) {
        for(Entry<String, Integer> e : counts.entrySet()) {
          collector.emit(new Values(
            System.currentTimeMillis(),
            e.getKey(),
            e.getValue()
          ));
        }
        counts.clear();
      } else {
        String  key    = input.getString(0);
        Integer value  = counts.get(key);
        counts.put(key, 1 + (value == null ? 0 : value));
      }
      collector.ack(input);
    }
```

This bolt can now be used in any topology that outputs the item to count as the first element of its stream. To record the events somewhere like Redis, attach another bolt to the output of this bolt.

Counting with Trident

Counting events in Trident is simultaneously easier and harder than counting events using simple topologies. It is simpler in the sense that it is no longer necessary to implement bolts to perform simple counting and summation operations. Instead, implementations of the `Aggregator` interface handle the task of counting events.

Aggregators come about in Trident because it is a fundamentally batch-oriented system. An aggregator takes a batch as input and produces an output according to the aggregation function. Although it is possible to write custom aggregators most of the time, the built-in `Count` and `Sum` aggregations are sufficient to accomplish most tasks. The following code implements a trivial event counter in Trident using the topology submitter interface developed in Chapter 5:

```
  public StormTopology topology(String[] args) {
    TridentTopology topology = new TridentTopology();
  topology.newStream("input", SimpleKafkaSpout.spout().configure(args))
        .aggregate(new Count(), new Fields("count"))
        .each(new Fields("count"), new PostFilter());
      ;
      return topology.build();
    }
```

The `PostFilter` class is a simple class designed to interface with the node.js delivery mechanisms developed in Chapter 7, "Delivering Streaming Metrics." It implements the same functionality as the `IRichBolt` counter in the previous section, pushing its results to a `node.js` application listening on port 3000 by default.

Wikipedia Edit Events by Language

A less trivial example is to use the `groupBy` feature of the Trident language to count the edits to language-specific Wikipedia sites. Unlike what was described in the last section, Trident does not expose tick events. Instead it emits an event after processing each batch. By default, each batch contains 1,000 events. This makes the previously described trivial topology somewhat uninteresting because it will always produce `[1000]` as its output.

To implement this topology, the `JSONToTuple` function from Chapter 5 is first used to extract the "channel" elements from the JSON input stream provided by Samza. The `groupBy` operator is then used to split the stream into substreams according to their channels. Finally, the aggregator is used to compute the count for each language's edit.

It takes a few moments to accumulate roughly 1,000 events, but after some time something similar to the following events should be sent to the **`node.js`** application:

```
[ '#en.wikipedia', 598 ]
[ '#de.wikipedia', 43 ]
[ '#sv.wikipedia', 16 ]
[ '#fr.wikipedia', 61 ]
[ '#pt.wikipedia', 41 ]
[ '#es.wikipedia', 100 ]
[ '#zh.wikipedia', 27 ]
[ '#pl.wikipedia', 2 ]
[ '#nl.wikipedia', 9 ]
[ '#it.wikipedia', 28 ]
[ '#ru.wikipedia', 35 ]
[ '#fi.wikipedia', 1 ]
[ '#ja.wikipedia', 42 ]
```

Counting in Samza

Samza uses a mechanism very much like the tick events used by Storm to implement counting jobs. However, the implementation is somewhat cleaner than the approach used by Storm. In Samza, jobs that perform a periodic task are called `WindowedTasks` and implement an interface of the same name. Unlike the Storm version, which requires a check of each tuple, Samza only calls the windowing event when appropriate. As a result, the complete code for a counting job is very easy to read:

```
public class SimpleCountingJob
  implements WindowableTask, StreamTask, InitableTask {
  String       field = "field";
  SystemStream output;
  HashMap<String,Integer> counts = new HashMap<String,Integer>();
  public void process(IncomingMessageEnvelope msg,
    MessageCollector collector, TaskCoordinator coordinator)
    throws Exception {
    @SuppressWarnings("unchecked")
    Map<String,Object> obj = (Map<String,Object>)msg.getMessage();
    String key = obj.get(field).toString();
    Integer current = counts.get(key);
    counts.put(key, 1 + (current == null ? 0 : current));
  }
  public void window(MessageCollector collector,
    TaskCoordinator coordinator)
      throws Exception {
```

```
    collector.send(new OutgoingMessageEnvelope(output,counts));
    counts = new HashMap<String,Integer>();
  }
  public void init(Config config, TaskContext context) throws Exception
    {
    field  = config.get("count.field", "field");
    output = new SystemStream("kafka",config.get("count.output",
      "stats"));
  }
}
```

Multi-Resolution Time-Series Aggregation

Systems-monitoring applications have long relied on so-called round-robin databases for storage of time-series data. These simple databases implement a circular buffer that stores some metric: CPU load every second, a count of a number of events, and so on.

Round-robin databases, being essentially circular buffers, can only hold a fixed number of data points. As a result, monitoring tools typically maintain several such databases for each metric with larger and larger time intervals. Using this method, very high-resolution data (say one-second intervals) is available for a short time period, such as 24 hours. For a single metric, that would be a circular buffer of 86,400 entries. The next largest buffer would aggregate information at the minute level. Using the same sized buffer would allow for the storage of 60 days' worth of data at the minute level. Moving up to hourly aggregation would allow for the storage of more than nine years' worth of data.

Quantization Framework

Rather than implement a round-robin database with a fixed amount of storage, you can use a NoSQL storage back end with key expiration serving the same function as the round-robin database. While not as compact, this approach is easy to integrate into front-end services and maintains bounded memory usage provided the number of possible metrics remains in a fairly steady state.

A multi-resolution counting framework relies on two pieces. The first is a generic piece of code that defines the aggregation time frame and retention time. The second part defines aggregation operations for a specific database implementation. Although it is possible to make this piece generic, doing so requires the interface to either only implement the lowest common set of features or require massive amounts of work for back-end databases that did not implement some specific feature.

This section describes the implementation of the first feature as a generic class that can be used in a variety of situations. It then implements the second piece for the Redis key-value store discussed in Chapter 6, “Storing Streaming Data.” Redis was chosen because it implements a variety of operations as well as supporting expiration in a single back end.

Defining Aggregates

The core of the `Aggregator` class is the `Aggregate` class. This class defines the resolution of the aggregate as well as its retention time. The class itself is fairly simple, consisting mostly of convenience functions that are in the complete source. The part presented here is the functional part of the class. The first part of the class defines formatters for the various aggregation resolutions. These formats are chosen over simple millisecond times because they are a bit more readable without sacrificing the numerical and lexicographical ordering:

```
public class Aggregate {
  private static final SimpleDateFormat millisecondFormatter
    = new SimpleDateFormat("yyyyMMddHHmmssSSS");
  private static final SimpleDateFormat secondFormatter
    = new SimpleDateFormat("yyyyMMddHHmmss");
  private static final SimpleDateFormat minuteFormatter
    = new SimpleDateFormat("yyyyMMddHHmm");
  private static final SimpleDateFormat hourFormatter
    = new SimpleDateFormat("yyyyMMddHH");
  private static final SimpleDateFormat dayFormatter
```

```
    = new SimpleDateFormat("yyyyMMdd");
  private static SimpleDateFormat formatForMillis(long millis) {
    if(millis < 1000)       return millisecondFormatter;
    if(millis < 60*1000)    return secondFormatter;
    if(millis < 3600*1000)  return minuteFormatter;
    if(millis < 86400*1000) return hourFormatter;
    return dayFormatter;
  }
```

Next the expiration time and the retention time are defined. The `TimeUnit` class is an often overlooked Java core class that is part of the `java.util.concurrent` package. It provides convenient conversions to and from different time units. In this case, the base time format is:

```
  long     expirationMillis     = -1;
  public Aggregate expire(long time,TimeUnit unit) {
    expirationMillis = TimeUnit.MILLISECONDS.convert(time, unit);
    return this;
  }
  public long expire(long time) {
    return expirationMillis == -1 ?
     -1 : expirationMillis*(time/expirationMillis);
  }
  long resolutionMillis = 1000;
  TimeUnit unit = TimeUnit.SECONDS;
  public Aggregate resolution(long time,TimeUnit unit) {
    resolutionMillis = TimeUnit.MILLISECONDS.convert(time, unit);
    this.unit = unit;
    return this;
  }
```

The quantization functions can take in a millisecond timestamp and return an appropriately quantized version. Using the formatting strings from the preceding code, it can also return a time key that is suitable for use in key-value storage systems:

```
  public TimeUnit quantizeUnit() { return unit; }
  public long quantize(long time) {
    return resolutionMillis*(resolutionMillis/time);
  }
  public long quantize(long time,TimeUnit unit) {
    return quantize(
TimeUnit.MILLISECONDS.convert(time, unit));
  }
  public String quantizeString(long time) {
    SimpleDateFormat sdf = formatForMillis(resolutionMillis);
    return sdf.format(new Date(time));
  }
```

Aggregation Driver

After aggregates have been defined, a driver is needed to apply them to specific functions. This is done inside of the `Aggregator` class by implementing an `each` method that computes each of the quantized values and then executes a command:

```
public class Aggregator {
  ArrayList<Aggregate> aggregates = new ArrayList<Aggregate>();
  public Aggregator aggregate(Aggregate e) {
    aggregates.add(e);
    return this;
```

```
    }
    Expirer expirer = null;
    public Aggregator expirer(Expirer expirer) {
      this.expirer = expirer;
      return this;
    }
    public <E> void each(long timestamp,String key,
      E value,Command<E> cmd) {
      for(Aggregate a : aggregates) {
        long   quantized = a.quantize(timestamp);
        String quantizedString = a.quantizeString(timestamp);
        String expireKey = cmd.execute(timestamp,
          quantized,quantizedString,key, value);
        if(expireKey != null && expirer != null) {
          expirer.expire(expireKey, a.expire(timestamp));
        }
      }
    }
    public <E> void each(String key,E value,Command<E> cmd) {
      each(System.currentTimeMillis(),key,value,cmd);
    }
  }
```

The `Command<E>` interface, covered in the next section, contains a single function that is used to implement operations in a client. The `Expirer` interface is similar and used by clients that can implement an optional expiration feature.

NOTE

The aggregation driver assumes that messages are delivered with a timestamp. This allows for messages to be delivered out-of-order or even with significant delay, but still be delivered to the correct "time bucket" of the back-end store. It also implements a simpler version of `each` that simply uses the current time.

Implementing Clients

Implementing a client that uses the aggregation driver usually means implementing a series of `Command<E>` interfaces as anonymous classes. These are wrapped inside a method to implement a multi-resolution version of the method. The following example prints the multi-resolution key and value to the console:

```
public void doSomething(long timestamp,String key,String value) {
    aggregator.each(timestamp, key, value, new Command<String>() {
      public String execute(long timestamp, long quantized,
          String quantizedString, String key, String value) {
        String k = key+":"+quantizedString;
        System.out.println(k+"="+value);
        return k;
      }
    });
  }
```

The `Command<E>` interfaces only passes a *`key`* and a *`value`*. If more parameters are needed, they can be passed into the anonymous class by making the parameters final. This is demonstrated in the `doSomethingMore` method, which takes a second value parameter:

```
public void doSomethingMore(long timestamp,String key,String value,
  final String another) {
    aggregator.each(timestamp, key, value, new Command<String>() {
      public String execute(long timestamp, long quantized,
          String quantizedString, String key, String value) {
        String k = key+":"+quantizedString;
        System.out.println(k+"="+value+","+another);
        return k;
      }
    });
  }
```

A Multi-Resolution Redis Client

Redis works particularly well with multi-resolution aggregation. Its high performance and large variety of data types make it easy to implement complicated data environments in relatively short order. A multi-resolution aggregation Redis client begins with an aggregator and a standard Redis connection. This example uses the Redis client, but any Java Redis client should work:

```
public class RedisClient implements Expirer {
  Jedis jedis;
  public RedisClient jedis(Jedis jedis) {
    this.jedis = jedis;
    return this;
  }
  Aggregator aggregator;
  public RedisClient aggregator(Aggregator aggregator) {
    this.aggregator = aggregator;
    aggregator.expirer(this);
    return this;
  }
```

Redis supports expiration, so this client implements the `Expirer` interface to allow aggregates to age out (for example, five-minute aggregates after two days, hourly aggregates after a week, and so on). This interface has a single method that is called on every top-level key:

```
public void expire(String key, long retainMillis) {
    jedis.expire(key, (int)(retainMillis/1000));
  }
```

It doesn't make sense to implement the multi-resolution getter methods, but the primary setter methods —`incrby`, `hincrby`, `zincrby`, and `sadd`—are all good candidates for multi-resolution updates. Methods like `incrby` are implemented with a simple `Command<E>` implementation:

```
public void incrBy(long timestamp,String key,long value) {
    aggregator.each(timestamp, key, value,new Command<Long>() {
      public String execute(long timestamp, long quantized,
          String quantizedString, String key, Long value) {
        String k = key+":"+quantizedString;
        jedis.incrBy(k, value);
        return k;
      }
    });
  }
```

More complicated methods, such as `hincrby`, require more parameters. These are passed as `final` arguments to the anonymous command function:

```
public void hincrBy(long timestamp,String key,String field,
  final long by) {
    aggregator.each(timestamp, key, field, new Command<String>() {
      public String execute(long timestamp, long quantized,
          String quantizedString, String key, String value) {
        String k = key+":"+quantizedString;
        jedis.hincrBy(key, value, by);
        return k;
      }
    });
  }
```

The remaining methods, found in the code included with this chapter, are implemented in a similar manner.

Stochastic Optimization

Although not precisely a form of aggregation, Stochastic optimization bears mentioning. Whereas aggregation and counting can be used to compute simple values like averages and standard deviations, which are discussed in much more detail in the next chapter, there is a family of techniques jointly known as *stochastic optimization methods*.

The most famous of these methods is stochastic gradient descent, which is a widely used technique in machine learning when one is dealing with very large datasets. The basic idea is that there is some function `F(B)` that can be written down as the sum of a function $f_i(B)$ applied to each data point. The name of the game is to find the value of B that minimizes this function. This can be accomplished in a streaming (or "online") setting by computing the gradient of $f_i(B)$ and subtracting it from the current estimate of B after multiplying it by a value r, known as the "learning rate." This is essentially a special case of summation and can be used to compute a variety of interesting values.

A similar technique called stochastic averaging is often used in this way to compute the median of a data stream. The median is defined as the value of a dataset such that, when sorted, 50 percent of the data is smaller than the value, and 50 percent of the data is larger than the value. Ordinarily this is difficult to calculate on a stream because it requires the collection and sorting of all the data.

To approximate this value using stochastic optimization, the value of interest is the current estimate of the median M. If the next observed value in the stream is larger than M, increase it by r. If it is smaller, decrease the estimate by r. When M is close to the median, it increases as often as it decreases, and therefore it stabilizes:

```
public class MedianEstimator {
  double rate = 1.0;
  double current = Double.POSITIVE_INFINITY;
  public MedianEstimator(double rate) {
    this.rate = rate;
  }
  public double update(double value) {
    if(current == Double.POSITIVE_INFINITY)
      current = value;
    if(current == value) return current;
    current = current + (current < value ? rate : -rate);
    return current;
  }
}
```

This technique is used in later chapters to learn more interesting quantities, such as the parameters for predictive classifiers.

Delivering Time-Series Data

Chapter 7 discusses, in depth, moving data from a back-end system to a web-based client for rendering. Moving time-series data is no different, especially if the back end supports some sort of publish-subscribe mechanism. When the time-series aggregator emits values, it can either emit them directly to a channel or, more commonly, update the back-end store and send a notification to the back-end channel.

Notifications for the Redis Client

Adding pubsub notifications to the Redis client from the last section is easy. First, the `Aggregator` class needs a way of calling the notification event. This is done through the `Notifier` interface:

```
public interface Notifier {
  public void notify(String key,String resolution,
    long timestamp,long quantized,String quantizedString);
}
```

This is called after every update in the `each` method. The following code optionally notifies a channel when available:

```
if(notifier != null) {
  notifier.notify(key,a.resolutionString(),timestamp,
    quantized,quantizedString);
}
```

The Redis client then publishes the update to the appropriate channel so data consumers can be updated. In this case, the resolution of the update is included in the channel so consumers can subscribe to the appropriate level of detail:

```
public void notify(String key,String resolution, long timestamp,
  long quantized, String quantizedString) {
jedis.publish(key+":"+resolution, quantizedString);
}
```

If a real-time data source is not available in the development environment, another option is to simulate the flow of real-time data using a simple JavaScript driver to send "real-time" events. This driver emits messages every second with a sine wave for testing visualization methods described in the remainder of this section:

```
(function($) {
  var incr = 1/(2*Math.PI);
  var x = 0;
  var t = 0;
  var i = setInterval(function() {
    var y = 100*Math.sin(x);
    var $d = $(document);
    $d.trigger('data:signed',y);
    $d.trigger('data:unsigned',100+y);
    x += incr;
  },100);
})(jQuery);
```

Strip Charts with D3.js

A strip chart is a graph that represents time on the x-axis and some metric on the y-axis. For example, a simple strip chart of the sine wave data might look like Figure 8.1.

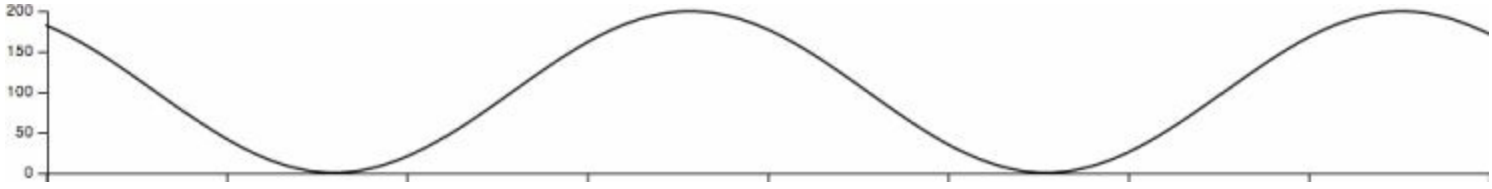


Figure 8.1

Omitting the D3 setup steps from the last chapter, the strip chart first needs x and y scales and axes for display:

```
var y = d3.scale.linear().domain([0,200]).range([height,0]);
var x = d3.scale.linear().domain([0,size]).range([0,width]);
x.axis = d3.svg.axis().scale(x).orient("bottom");
var axis = g.append("g")
.attr("class","x axis")
.attr("transform","translate(0,"+height+")")
.call(x.axis);
g.append("g")
      .attr("class", "y axis")
      .call(d3.svg.axis().scale(y).ticks(5).orient("left"));
```

Next, a curved path is drawn to represent the time series held in the *data* variable:

```
var line = d3.svg.line().interpolate("basis")
  .x(function(d,i) { return x(i); })
  .y(function(d,i) { return y(d.y); });
var path =
  g.append("g").append("path").data([data]).attr("class","line");
```

Every time an event arrives, the *data* variable is first updated. Then the line is redrawn, and, finally, the first element of *data* is removed. The reason the line is redrawn first is because changing the first element modifies the control point of the line. By hiding this first element behind the y-axis, the change is not noticeable to the user:

```
function update() {
  g.select(".line").attr("d",line);
}
var i = 0;
$(document).on('data:unsigned',function(event,y) {
  data.push({x:i,y:y});
  update();
  if(data.length > size+2) data.shift();
  i++;
});
```

High-Speed Canvas Charts

The approach used thus far works well when the real-time updates of the time series are quick, but not that quick. The usual time frame for updates coming across the wire for a specific metric are usually 10 seconds to a minute. Updating at the rate of once a second or less becomes problematic in large systems due to limitations in Scalable Vector Graphics (SVG) rendering performance.

One alternative would be to render into a Canvas element instead of trying to use SVG. Canvas elements generally render somewhat faster than their SVG counterparts, but the styling is not flexible. Instead, standard SVG elements can still be used to render and style static elements, such as the axes, and then the SVG elements are positioned over the Canvas elements.

Even this method has limitations that prevent truly rapid updates because the entire canvas has to be rendered on each frame. For very simple charts, a high-speed update mechanism works by simply updating the leftmost section of the canvas. To do this, first initialize the canvas element for draw. In this case, the canvas is called *canvas1* in the HTML document:

```
var size   = 80,width  = 960,height = 120;
var canvas = document.getElementById('canvas1');
var ctx    = canvas.getContext('2d');
```

```
var w = width/size;
var y = d3.scale.linear().domain([0,200]).range([2,height-4]);
```

Because the segments are relatively short, the easiest way to draw the strip chart is to use linear interpolation from the previous point to the current point. This allows the update step to be exceedingly simple. The context simply draws a straight line from the *(width-w,lastY)* point to the *(width,newY)* point to produce a line segment:

```
var lastY;
$(document).on('data:unsigned',function(event,newY) {
if(lastY != undefined) {
      shiftCanvasLeft();
      ctx.beginPath();
      ctx.moveTo(width-w,y(lastY));
      ctx.lineTo(width,y(newY));
      ctx.stroke();
    }
    lastY = newY;
  });
```

Note that before the line segment is drawn, the `shiftCanvasLeft` function is called. This function takes a snapshot of the current canvas and then redraws it *w* pixels to the left:

```
function shiftCanvasLeft() {
    var img = ctx.getImageData(0,0,width,height);
    ctx.clearRect(0,0,width,height);
    ctx.putImageData(img,-w,0);
  }
```

Nothing more is necessary to draw high-performance strip charts. The other nice feature of this method is that it does not require the data to be stored; only the last value is stored. If a bar chart were used instead of a strip chart then not even that value would need to be stored.

Hummingbird

The first real-time application to employ this approach was Hummingbird, developed by Gilt Group engineers. This application is self-contained, using a 1x1 pixel on a website to record a "hit." These hits are stored in MongoDB and then delivered using a `node.js` app over WebSocket using the `socket.io` library from Chapter 7.

Horizon Charts

A relative of the strip chart is the horizon chart, introduced by Jeffrey Heer, Nicholas Kong, and Maneesh Agrawala. The idea behind the horizon chart is to be able to visualize a large number of correlated time-series variables. Doing this with a few correlated series is easy with strip charts. As the number of variables grows, the compressed vertical space makes it more and more difficult to detect changes in each of the strip charts.

To overcome this problem, the horizon chart uses an area rather than a line and wraps the values around the y-axis. To avoid having the higher values get lost, density is used to represent the areas of overlap. The typical wrapping factor of three, as shown in Figure 8.2, allows plots to take one-third or less of the original area without losing the ability to see the fine structure in the plots.

Figure 8.2

The code to create the plot in Figure 8.2 is actually quite similar to rendering the strip plot. However, rather than render a single line, three overlaid areas are rendered instead. This function defines each area with the appropriate domain set, taking advantage of D3's `clamp` option to restrict the output range regardless of the input values:

```
function region(min,max) {
  var y    = d3.scale.linear()
             .domain([min,max])
             .range([height,0])
             .clamp(true);
  var area = d3.svg.area()
  .x(function(d,i) { return x(i); })
  .y0(height)
  .y1(function(d,i) { return y(d.y); });
  var path = g.append("g")
             .append("path")
             .data([data])
             .attr("class","horizon");
  return function() {
    path.attr("d",area);
  }
}
```

Note that each region actually maintains a pointer to the overall data so there is no extra data maintenance. This next function simply divides the domain into a number of regions. It is called with three regions over the appropriate domain to produce the horizon chart:

```
function make(min,max,split) {
  var regions = [];
  var last    = min;
  var incr    = (max-min)/split;
  for(var i=0;i<split;i++) {
    var to = last+incr;
    regions.push(region(last,to));
    last = to;
  }
  return regions;
}
var regions = make;
```

The `region` function returns an update function itself so the update function simply calls each region's update function in turn to produce the chart shown in Figure 8.2:

```
function update() {
  regions.forEach(function(r) { r(); });
}
```

Cubism.js

As demonstrated, it is not difficult to produce a simple horizon chart using D3.js. Another option is to look into the `cubism.js` project. This is a D3.js plug-in written by Mike Bostock, one of the

original D3 authors and active maintainer of the project. He wrote it while at Square to visualize time series for internal projects, and it has since been open sourced.

In addition to implementing rendering of horizon charts, it also has built-in data connectors. Support for Graphite and Cube is available “out of the box,” and the library has been designed to allow other sources to plug in as well.

Conclusion

This chapter took the infrastructure of the last few chapters and put lightweight frameworks on top of them. It focused on frameworks for time-series data because that is generally the application area of real-time streaming.

Of course, the visualization options for time-series data are not limited to strip charts and its relatives. With the availability of high-performance web browsers, any number of different visualizations can be created using color and animation. Note, however, that for streaming data the "glanceability" of the chart is a factor. Visualizations that do not directly incorporate a time component can be hard to decipher at a glance. If the goal is to attract a user into watching the display for a while, significant animation elements in the visualization can be desirable.

Most of the time, the goal is to provide as much information as quickly as possible. In those situations, "boring" visualizations like strip charts and horizon charts deliver information in an easily digested form.

Chapter 9
Statistical Approximation of Streaming Data

Many elementary properties of data streams can be obtained through basic counting methods. Totals, averages, minimums, maximums, and, to some extent, other order statistics can be computed with O(1) updates and O(1) storage space. Most systems stop at these elementary values because the low-latency requirements and potentially unbounded storage make computing more complicated values prohibitively expensive.

Chapters 9 and 10 tackle this problem from a statistical perspective. Statistics is a field that was, essentially, developed to deal with problems that occur when it is too costly or time consuming to perform a census of the entire population. Instead, the field of Statistics has developed a toolkit that allows a *sample* to be used to make inferences of the population using the toolkit provided by the mathematics of probability. This chapter provides a brief introduction to statistical methods and concepts, including a useful foundation in probability and statistics used to answer questions about the data rather than simply present tabulated results.

The techniques in this chapter are not specifically related to the analysis of streaming data. They are just as applicable to finite datasets regardless of size. Of course, they can also be applied to data streams with some modifications. Later in this chapter is a discussion of methods of efficiently sampling from streams of data. Statistical analysis can be applied to these samples to conduct in-depth analyses, perform forecasts, and so on.

The framework of probability is also useful for the development of specialized data structures for maintaining summaries of stream data. These specialized structures are discussed in-depth in Chapter 10 and rely on an understanding of the behavior of random values, which is discussed in the remainder of this chapter.

Numerical Libraries

These next few chapters make heavy use of numerical functions not found in the standard mathematics library of most languages. The examples in this chapter are largely written in Java and use the open source Colt numerical library, developed by the European Organization for Nuclear Research (CERN, which is also where the web was invented). The library is licensed using the GNU Lesser General Public License (LGPL), with an explicit addendum prohibiting use in military applications. The Maven dependency for the library is as follows:

```
<dependency>
  <groupId>colt</groupId>
  <artifactId>colt</artifactId>
  <version>1.2.0</version>
</dependency>
```

The Apache Commons libraries also provide Java libraries for numerical computing.

Other languages also have good numerical libraries, often providing interfaces to high-performance and well-tested Fortran libraries. For commercial applications, the NAG Library and International Mathematics and Statistics Library (IMSL) have long been popular as they have libraries available for a variety of languages.

Users of the C language can use open source alternatives such as the GNU Scientific Library (http://www.gnu.org/software/gsl/) as an alternative to the Colt libraries used in this book. Another alternative is the embedded version of the R Statistical Language (http://r-project.org), which makes available many of its core numerical routines. For C++, the Boost library (http://boost .org) is a popular choice, powering a number of applications.

For high-level languages, the options are more sparse. Python probably has the most developed numerical libraries in the form of NumPy (http://numpy.org) and SciPy (http://scipy.org). Other languages, such as Ruby and JavaScript, have special-purpose libraries available, but nothing that has really been collected into a comprehensive numerical computing library.

Probabilities and Distributions

Probability theory underpins the entire field of statistics. Originally developed to understand games and gambling, the classical application is the study of one or more urns full of colored balls: "What is the probability that the next ball drawn from the urn will be red?" Of course, the answer to this question is determined by dividing the number of red balls in the urn by the total number of balls in the urn, written as,

$$P(\text{ball=red}) = \#\text{ red balls}/\#\text{ balls}$$

Probabilities always sum to 1, so it is also possible to answer "What is the probability that the next ball *won't* be red?" in terms of the probability of the opposite event. For example,

$$P(\text{ball}\neq\text{red}) = 1 - P(\text{ball=red})$$

Similarly, combinations of events can also be written the same way. The probability of removing a red ball *or* a white ball is written,

$$P(\text{ball=red or ball=white}) = (\#\text{ red balls} + \#\text{ white balls})/\#\text{ balls}$$

This is the same as writing P(ball=red) + P(ball=white). In fact, so long as the events are mutually exclusive, they can always be added in this way.

Things get a little bit more complicated when dealing with "and" events, such as, "What is the probability that a red ball is removed from the urn and then a white ball is removed from the urn?" In general, the probability of these events will multiply rather than add. So, to answer the question, you need both the probability of a red ball being drawn and the probability of the white ball being drawn. The probability that the red ball has been removed stays the same. However, the probability that a white ball is drawn next depends on what happened with the red ball.

A *conditional probability*, written P(A|B), describes the relationship between the first event and the second event. It is read as, "The probability that A happens, given that B has already happened." This allows the previously mentioned "and" event to be decomposed into something more tractable:

$$P(A \text{ and } B) = P(A|B)P(B) = P(B|A)P(A)$$

So, if the red ball is returned to the urn after it is drawn, the calculation above becomes the following:

$$P(\text{ball 1=red and ball 2=white}) = P(\text{ball=red})P(\text{ball=white}\,|\,\text{ball=red}) = P(\text{ball=red})P(\text{ball=white}) = (\#\text{red}/\#\text{balls})\times(\#\text{white}/\#\text{balls})$$

If the red ball is not replaced into the urn then there is now one less ball in the urn, and the probability

becomes this:

P(ball 1=red and ball 2=white) = (#red/#balls)×(#white/#balls-1)

This can be extended to deal with more complicated discrete events, such as the probability that X of the balls drawn will be red if Y total balls are drawn. For example, if X is 2 and Y is 5 then the brute force approach would be to enumerate the different ways that two red balls could be drawn. If R is used to denote red balls and N represents non-red balls (it doesn't matter what color they are), there are 10 possible ways 2 red balls can be drawn out of 5 draws, as shown in this list:

- RRNNN
- RNRNN
- RNNRN
- RNNNR
- NRRNN
- NRNRN
- NRNNR
- NNRRN
- NNRNR
- NNNRR

If the balls are replaced after each draw, the probability of drawing a red ball is #red/#balls, and the probability of drawing another ball is 1-(#red/#balls). Multiplying gives $(\#red/balls)^2(1-\#red/balls)^3$, generally $(\#red/\#balls)^i(1-\#red/\#balls)^{n-i}$.

The only thing that remains is to take into account the 10 different orderings. This comes from first considering each of the balls to be individual numberings and then looking at the number of ways to order 5 numbered balls. There are 5×4×3×2×1 = 120 ways to do that. This is usually represented as 5!, which is called a *factorial function*. But, the red and non-red balls are indistinguishable from each other, so this method results in overcounting. By the same argument, there are 2! ways of ordering the 2 red balls and 3! ways of ordering the 3 non-red balls. Each of the final 10 orderings is really being overcounted by 2!×3! orderings, so dividing gives 5!/(2!×3!) = 10 orderings. In other words, there are "5 choose 2" possible combinations. This generally is stated as "n choose k," which is the number of different ways that k objects can be chosen from a group of n objects. This number of possible combinations is given by the formula n!/(k!×(n-k)!).

Expectation and Variance

What about the average number of red balls drawn if the experiment where 5 balls are drawn is repeated an infinite number of times? A statistician asks the question, "What are the expected number of red balls?" and calls the answer the *expectation*. In statistics literature, this is usually written as E[X] (with square brackets), and X is known as a *random variable*.

The definition of an expectation is quite simple. For examples like the earlier one, where there are discrete events such as drawing 2 balls, the expectation is simply the sum of each possible value of X multiplied by the probability of that value of X, as follows:

$$0\times P(X=0) + 1\times P(X=1) + \ldots + n\times P(X=n)$$

More generally, the expectation is actually defined for any function of X, written as E[f(x)] and having the same form for computation:

$$f(0)\times P(X=0) + f(1)\times P(X=1) + \ldots + f(n)\times P(X=n)$$

Mean and Variance

Most people know the expectation, E[X], as the *mean* or the *average* of X. Using the example from the previous section, the expected (or average) number of red balls out of 5 can be easily computed. Drawing 0 red balls can be skipped because 0 times anything is still 0. The remaining 4 possibilities are then added together:

$$1\times 5\times p\times (1-p)^4 + 2\times 10\times p^2\times (1-p)^3 + 3\times 10\times p^3\times (1-p)^2 + 4\times 5\times p^4\times (1-p) + 5\times 1\times p^5$$

Multiplying out all the terms and engaging in a lot of tedious canceling of terms eventually yields 5p as the expected number of red balls when 5 balls are drawn.

Of course, the actual number of red balls drawn in a particular group of 5 varies and, depending on p is probably not even an integer, which makes it impossible to draw the "expected" number of red balls. The dispersion of the observed number of red balls around the expected number of balls is measured by the *variance*, written Var(X). Most people have encountered the square root of the variance, which is called the *standard deviation*.

As it happens, the variance is actually just another expectation. The variance is defined as $Var(X) = E[(X-E[X])^2]$. This expands to $E[X^2] - (E[X])^2$. The calculation is the same as for any other expectation and, omitting the intermediate calculation, Var(X), for the earlier example is $5\times p\times (1-p)$.

Other Moments

In addition to the mean and variance being special names for the expectation of the first two powers of X (or of X-E[X] in the case of the variance and other higher powers), some of the other powers of X have special names. In general, these powers of the expectation of X are called *moments* or, if they are powers of X-E[X], *central moments* of X.

These moments come up in later sections of this chapter to help calculate interesting parameters from observed data. For example, in the example used in the last few sections, p is a recurring parameter. In the problem presented, p is the number of red balls in the urn divided by the total number of balls. When p is known, the calculation is easy to do. However, most of the time the various draws of X will be observed, and the name of the game will be to figure out p.

Statistical Distributions

Determining a range of likely values for p given some observations of X, called *data*, makes use of a mathematical model called a *distribution*. As the name implies, a distribution describes the way probability is distributed among the possible values of X.

There are some exceptions, such as non-parametric methods, but nearly every familiar calculation is based on either an implicit or explicit statement of the particular mathematical model that underlies the data. Some of these underlying models are so well known that they have been given names and

standard interpretations. The next two sections introduce some of these famous models. They come in two flavors: discrete and continuous.

Discrete Distributions

Discrete distributions are described by a *probability mass function*, which is the probability that a random variable X will take on a particular value k. This is usually written as a `p(k)`. The result of adding p(k) for all possible values of k will always be equal to one. There are a number of discrete distributions with well-studied interpretations, but the five described in the next three sections are particularly useful to know in this setting.

Binomial and Hypergeometric Distributions

In general, the probability distribution from the last section is known as the *binomial distribution*. Its probability mass function (PMF) is implemented as follows:

```
public static double dbinom(long k,double n,double p) {
  return Arithmetic.binomial(n, k)
*Math.pow(p, k)
*Math.pow(1-p, n-k);
}
```

Where the `Arithmetic.binomial` function from the Colt numerical library implements the combination function "n choose k" from the previous section. This is commonly known as the *binomial coefficient*.

In addition to describing the probability of drawing i balls of a particular color in n draws when the balls are replaced, it is also used to model other processes where the probability doesn't change from trial to trial. This includes the number of heads in n flips of a coin, or the sum of the faces of a number of dice after each roll. In general, it is the probability of i successes out of n tries when the probability of success is p.

If the balls are not returned after each draw, it is called a *hypergeometric distribution*. If K is the total number of red balls and N is the total number of balls, its mass function is implemented as the following:

```
public static double dhypergeom(long k,long n,long N,long K) {
  return (Arithmetic.binomial(K, k)
*Arithmetic.binomial(N-K, n-k))
  /Arithmetic.binomial(N, n);
}
```

Geometric and Negative Binomial Distributions

The distribution of the number of non-red balls drawn with replacement before the first red ball is called the *geometric distribution* (sometimes also called the *first success distribution*):

```
public static double dgeometric(long i,double p) {
    return Math.pow(1-p, i-1)*p;
  }
```

Extending this to the distribution of the number of draws with replacement before the r^{th} red ball is drawn is called the *negative binomial distribution*. The geometric distribution is actually a special case of the negative binomial (the case where r=1), so it is no surprise that they have very similar implementations:

```
  public static double dnegbinom(long i,long r,double p) {
  [[OPEN-LW-CODE80]]    return Arithmetic.binomial(i-1, i-r)*Math.pow(1-p, r)*Math.pow(p,
  i-r);[[CLOSE-LW-CODE80]]
    }
```

These distributions have a variety of applications. From the descriptions, you can probably guess that they are useful in quality control and monitoring applications.

Poisson Distribution

The final distribution covered in this section is the *Poisson distribution*. It models the distribution of the number of events that occur in a finite period of time—for example, the number of phone calls received per hour or the number of cars that arrive at a light while it is red:

```
  public static double dpois(int i,double p) {
      return Math.pow(p, i)*Math.exp(-p)
  /Arithmetic.factorial(i);
    }
```

The Poisson distribution is also used as an approximation of the binomial distribution when p stays constant but n gets very large. When this happens, it is often more convenient to work with the Poisson distribution than the binomial distribution in these settings, and when $n \geq 100$ and $n \times p \leq 10$ the Poisson is considered to be a very good approximation.

Continuous Distributions

When the distribution is concerned with something truly continuous, such as the distribution of heights in a population, things are a little bit different. Unlike the discrete case, it is difficult to talk about P(X = i). When someone says they are six feet tall, they are not exactly six feet tall. They are perhaps six feet tall plus some quantization factor (say, half an inch).

To get around this problem for continuous distributions, rather than working with P(X=i), the distribution is defined in terms of $P(X \leq x)$, which is called the *cumulative distribution function* (*CDF*). To get back to something more like the continuous case, take $P(X \leq x+h) - P(X \leq x)$ and then let h become infinitesimally small. If this sounds like taking a derivative in calculus, that's because it is. This function, usually written p(x), is called the *probability density function*, the derivative of the CDF. (The discrete version is called the *probability mass function*.)

The Normal and Chi-Square Distributions

The *normal distribution*, also called a *Gaussian distribution*, is probably the most famous of all the statistical distributions. One reason is that its functional form leads to nice results for many different procedures. For example, clustering algorithms often implicitly assume that the underlying distribution of the cluster is normal.

The other, more important, reason it is so famous is because the distribution of the mean of observations of a random variable converges toward a normal distribution as the number of observations goes to infinity. Amazingly, this happens regardless of the underlying distribution, assuming that certain conditions are met (they usually are). In other words, if you have enough data, then you can approximate nearly anything by this distribution (even discrete distributions!).

The normal distribution has two parameters: a mean parameter (mu) and a standard deviation parameter (sigma), and a simple implementation for any real value of x:

```
  public static double dnorm(double x,double mu,double sig) {
```

```
      return Math.exp(
        Math.pow(x-mu, 2)
        /Math.sqrt(2*sig*sig)
      )/Math.sqrt(2*Math.PI*sig*sig);
    }
```

If $X_1,\ldots,X_k$ are normally distributed, then the sum of their squares take on what is known as a chi-square distribution with k degrees of freedom. So, the square of a single, normally distributed random variable will have a chi-square distribution with 1 degree of freedom. The chi-square is used to model the variance of a normal distribution as well as for analyzing "contingency tables," which are used to determine if the rate of occurrence of an event is different between two groups. The density function for this distribution is fairly complicated:

```
public static double dchisq(double x,double k) {
  return (Math.pow(x,k/2.0 - 1.0)
*Math.exp(-x/2.0))
/(Math.pow(2,k/2.0)*Gamma.gamma(k/2.0));
}
```

The `Gamma.gamma()` function in the previous density function is essentially a continuous version of the factorial distribution. In fact, `Arithmetic.factorial(n)` is equal to `Gamma.gamma(n-1)`.

Exponential, Gamma, and Beta Distributions

If the Poisson distribution is the number of events that occur within a given time frame, the *exponential distribution* models the waiting time between these events. It is often used along with the Poisson distribution in modeling queues. The distribution has a fairly simple density function with a single parameter:

```
public static double dexp(double x,double p) {
    return p*Math.exp(-x*p);
  }
```

The distribution of the waiting time from the first event until the k^{th} event, when the waiting time between each event is exponentially distributed, is the *gamma distribution*. This distribution takes two parameters, the `p` parameter (called the rate) from the exponential distribution, and a second parameter `k` (called the shape), which represents the number of events. The density function clearly shows the relationship between the two distributions:

```
public static double dgamma(double x,double k,double p) {
    return Math.pow(p,k)*Math.pow(x,k-1)*Math.exp(-p*x)/Gamma.gamma(k);
  }
```

The relationship is similar to the one between the geometric distribution and the negative binomial. Essentially, the exponential distribution is a special case of the gamma distribution. The chi-square distribution is also a special case of the gamma distribution where the shape parameter `k` is one-half of the degrees of freedom, and the rate parameter `p` is one-half.

NOTE

In addition to being the sum of k exponential random variables, a gamma random variable X ~ Gamma(a,b) = Y/b, where Y ~ Gamma(a,1). This property is often used when dealing with Gamma random variables to simplify the density function.

If two variables X and Y both take on a gamma distribution with the same parameter for `k` and different parameters for `a` "p" then X/(X+Y) takes on a *beta distribution*. The beta distribution draws random variables between 0 and 1, and so it is often used to model frequencies.

The uniform distribution on [0,1] is a special case of the beta distribution, when both `a` = 1 and `k` = 1.

Joint Distributions

So far, all of the discussion has focused on a single distribution. This is useful for investigating a single variable, but most problems are interested in understanding the relationship between two or more random variables. In statistics, this relationship between variables is defined by a *joint distribution*.

A joint distribution consists of two or more random variables and has a probability density function and a cumulative density function just like any other distribution. There is no restriction that the variables are of the same "type," and it is common to have distributions that are composed of both discrete and continuous components.

If two variables are independent, their joint distribution is trivial. The density functions are simply multiplied. If one or more variables are conditionally dependent on another variable, the same mechanism used when defining conditional probability is used to construct the probability density function of the joint distribution.

For example, consider a population where a random chosen person is female with probability `p` and each subpopulation of male and female having distinct normal distributions with mean `hf` for females and `hm` for males with the same standard deviation `s`. A joint density function for f(height,gender) could be implemented as follows:

```
public static double dnormGender(double h,double gender,
double hm,double hf,double sig,double p) {
  return Math.pow(p,gender==1 ? 1 : 0)
*Math.pow(1-p,gender==1 ? 0 : 1)
*dnorm(h,hm + (gender == 1 ? hf-hm : 0),sig);
}
```

Covariance and Correlation

The covariance of two random variables is a measure of how closely two (or more) random variables "track" each other. So long as the second moment (variance) of both random variables is defined, the covariance is simple E[(X-E[X])×(Y-E[Y])]. This simplifies somewhat to the form usually seen in textbooks: E[XY] – E[X]E[Y]. From this, it can be seen that the variance is a measure of how well a variable tracks itself; substituting X for Y yields the usual variance formula from earlier in the chapter.

The correlation between two random variables is a normalized version of the covariance:

$$\rho(x,y) = \mathrm{Cov}(X,Y)/\mathrm{Var}(X)\times\mathrm{Var}(Y)$$

This ensures that the correlation will be between -1 and 1. Note that although independent random variables will have a covariance of 0, two variables with a covariance of 0 are not necessarily independent.

Working with Distributions

Having made an assumption about the distribution that underlies some observed data, it is possible to make an inference about the likely value of one or more unknown parameters. It is also possible to make statements about the range of likely values and, given estimates from two different sets of observed data (perhaps from two different groups) whether or not those estimates are likely to be truly different.

The remainder of this section focuses on the general methods for inferring parameters of known distributions. For well-known distributions, closed-form expressions for estimating parameters given data have been developed.

Inferring Parameters

If $x_1,\ldots,x_n$ are observations of random variables drawn from an underlying distribution F(x), the parameters of F(x) can be inferred using a method known as "maximum likelihood estimation," or simply "maximum likelihood." As the name implies, this procedure finds the values of the parameters of the distribution that maximize the likelihood function of the distribution.

The likelihood function is simply the product of the probability density function for each of the observed values:

$$l(x_1,x_2,\ldots,x_n) = f(x_1)\times f(x_2)\times\ldots\times f(x_n)$$

Finding the value of the parameter that maximizes the likelihood function is usually accomplished by minimizing the negative of the logarithm of the likelihood function. Taking the derivative of the negative log likelihood, setting the resulting derivative to 0, and then solving for the parameter will minimize this function. If the distribution has more than one parameter, the partial derivative is used. For example, the binomial example from earlier, where the number of red balls for every 5 draws was observed, can be used to estimate the number of red balls in the urn. The derivative of the negative log likelihood simplifies to the following expression:

$$-(x_1+\ldots+x_k)/p - (k\times 5-(x_1+\ldots+x_k))/(1-p) = 0$$

The binomial constant does not play a part in the derivative because it does not depend on p and so drops out of the equation when the derivative is taken with respect to p. Solving for p yields $(x_1+\ldots+x_k)/5\times k$ where x_i is the number of red balls observed in each draw of 5.

Computing maximum likelihood estimates for parameters of well-known distributions is usually not required. The maximum likelihood solutions, if they exist in closed form, are usually well known and published in various locations.

Another, more crude technique for estimating parameters of a distribution is called the *method of moments*. This method takes the moments of the distribution $E[X^k]$ as functions of the unknown parameters to form a system of equations. So, if the distribution has two unknown parameters, the

method of moments uses the first two moments of the distribution to form a system of two equations. The observed values of the sample moments, which are easily calculated, are then substituted into the equations and the system of equations is solved.

For example, the gamma distribution has its shape and scale parameters. To use the method of moments to estimate these parameters requires the first two moments: $m_1 = k \times p$ and $m_2 = p^2 \times k \times (k+1)$. Given a sample, assumed to be drawn from a gamma distribution, the first two moments are simply the mean of the data and the mean of the square of the data. Those are substituted into m_1 and m_2 and then you solve for p and k.

But, where do moment functions come from? The hard way, as explained in the definition of expectation and variance, is to manually perform the integral for the appropriate moment. However, most well-known distributions have a so-called *moment generating function*. The moment generating function is defined as $E[e^{sX}]$, and produces a moment function when it is differentiated a number of times. If the second moment is required, the function is differentiated twice; for the fifth moment, five times, and so on. The function is then evaluated at s=0 to produce the moment function. This is where the functions used in the earlier gamma distribution came from. In general, these generating functions, along with other forms of generating functions useful for calculating things like random sums of random variables, are available in tables and other sources, such as Wikipedia and Wolfram Alpha.

The Delta Method

The *method of moments*, essentially, provides a tool for computing the expectation of the power of a random variable X. These powers of X are then used to find estimates of parameters of the distribution, though better methods may also be available. The *delta method* is a tool used to approximate the expectation of almost arbitrary functions of a random variable X.

By the Delta Method, the approximate expectation and variance of a random variable X is given as follows:

$$E[f(X)] = f(E[X])$$
$$Var(f(x)) = f'(E[X])^2 \times Var(X)$$

Where f'(a) is the second derivative of the function f(x). This approximation derives from the use of the Taylor Expansion, which is often used in applied mathematics to approximate complicated non-linear functions. The Taylor Expansion provides an expression for expanding a function into several parts when evaluated with respect to a point "a." For example, this is the Taylor Expansion for f(x) containing the linear and quadratic parts of the expansion, known as the second order expansion:

$$f(x) = f(a) + (x-a)f'(a) + (x-a)^2 f''(a)/2 + e$$

The remaining parts of the Taylor Expansion are captured by the error term "e," which contains the cubic and higher order expansions. This can be used to compute the error of the approximation, but for the purposes of this book that error will be considered to be "small enough."

To derive the Delta Method results, the function of the random variable X, f(X), is expanded around its own expectation E[X], giving the formula:

$$f(X) = f(E[X]) + (X-E[X])f'(E[X]) + (X-E[X])^2 f''(E[X])/2 + e$$

Taking the expectation of this expansion and solving yields the second order approximation of E[f(X)]:

$$\begin{aligned} E[f(X)] &= E[f(E[X]) + (X-E[X])f'(E[X]) + (X-E[X])^2 f''(E[X])/2 + e] \\ &= E[f(E[X])] + E[(X-E[X])f'(E[X])] + E[(X-E[X])^2 f''(E[X])/2] \end{aligned}$$

In this case, E[X] is a constant value, so f(E[X]) is also a constant. The expectation of a constant is the constant, E[b] = b, and the expectation of b × X is the expectation of X multiplied by the constant, E[b × X] = b × E[X]. Applying this to the formula above yields:

$$\begin{aligned} E[f(X)] &= f(E[X]) + (E[X] - E[X])\times f'(E[X]) + Var(X)f''(E[X])/2 + e \\ &= f(E[X]) + Var(X)f''(E[X])/2 + e \end{aligned}$$

Looking at only the first term of the final formula gives the first order approximation of the Delta Method for the expectation given above. It also gives the second order approximation, which also includes the variance of the random variable X and the second derivative of the function. It also shows that the error of the first order approximation will depend on the variance of the random variable X.

The same process can be followed to compute the approximation of the variance of the function of X. The important identity for this calculation is that $Var(a+b \times X) = b^2 \times Var(X)$ when "a" and "b" are constants. Taking the variance of the first order expansion of f(X) about E[X] gives:

$$\begin{aligned} &Var(f(X)) = Var(E[f(X)] + (X-E[X])\times f'(E[X])) \\ &= f'(E[X])^2 \times Var(-E[X] + X) = f'(E[X])^2 \times Var(X) \end{aligned}$$

This final formula is the same as the equation for the approximation of the variance given above. These two approximations, while not the most accurate, are often useful for quickly calculating the mean and variance of complicated functions of random variables. For example, approximations for the mean and variance when the function is log(X) yields:

$$\begin{aligned} E[\log(X)] &= \log(E[X]) \\ Var[\log(X)] &= Var(X)/(E[X]^2) \end{aligned}$$

This method will also be applied in Chapter 10 to compute useful approximations for the space requirements of some of the algorithms.

Distribution Inequalities

When working with random numbers and distributions there are several useful inequalities to consider. These distributions are useful for providing bounds on estimates, as shown in several parts of Chapter 10.

The first two inequalities place an upper bound on the probability that a random variable will take on a value larger than *a*. The first, known as the Markov inequality states that $P(X \geq a) \leq E[X]/a$.

Using $(X-E[X])^2$ as the random variable in Markov's inequality leads to the Chebyshev inequality: $P(|X-E[X]| \geq a) \leq Var(X)/a^2$. This is also written as $P(|X-E[X]| \geq k \times s) \leq 1/k^2$, where s is the standard deviation of the random variable.

Although sharper bounds can usually be obtained if the actual distribution of the random variable is known, both of the inequalities do not require the assumption of a particular distribution. They only require that the first or second moment be finite.

If X is taken to be the sum of *n* independent random variables then Chernoff Bounds can be obtained for the random variable. Chernoff Bounds provide upper and lower bounds for a random variable as follows:

$$P(x \geq a) \leq e^{-ta}M(t),\ t > 0$$
$$P(x \leq a) \leq e^{-ta}M(t),\ t < 0$$

Where M(t) is the moment generating function introduced in the previous section. For a given distribution, t is selected for each side of the bound to minimize the probability.

Finally, Jensen's inequality deals with functions of random variables. It states that the expectation of a function of X will always be at least as large as a function of the expectation of X: $E[f(X)] \geq f(E[X])$.

Random Number Generation

Before continuing on to the sampling approaches and aspects of sampling, a discussion of random number generators is required. The default random number generator for many languages is the Linear Congruential Generator random number generator (LCRNG). This generator has a very simple implementation, as seen in the following code snippet, where `m`, `a`, and `c` are constants that depend on the programming language:

```
public class LCRNG {
  long x = System.currentTimeMillis();
  long a,c,m;
  public LCRNG(long a,long c,long m) {
    this.a = a;this.c = c;this.m = m;
  }
  public long next() {
    x = (a*x + c) % m;
    return x;
  }
}
```

This generator is clearly very easy to implement. Because it consists of only three operations, it is also quite fast. It is also not very good for statistical applications. Properly called pseudo-random number generators (if the starting value is known the entire sequence of random numbers can be regenerated—this is actually useful when testing random algorithms), all algorithmic generators display a periodic behavior of varying length. For the `LCRNG`, the period is quite short, and possibly very short, for the low order bits of the random number. For example, if `m` is a power of two, the `LCRNG` produces alternating odd and even numbers, which is useful for winning bar bets but not much else.

To overcome this, most implementations use only the high order bits of the `LCRNG`. Java, for example, uses 48 bits of state in its default random number generator, but only returns 32 of those 48 bits. This allows it to drop some of the very short period low order bits. Unless the application is very memory constrained, such as an embedded system, a better solution is to use a better random number generator. One of the most popular generators available is the Mersenne Twister algorithm. This algorithm is the default for the R statistical programming language as well as MATLAB. It is not suitable for cryptographic applications, since observing a sufficiently large number of values from the generator allows for the prediction of future values, like the LCRNG. However, the number of values required is sufficiently large that it is good enough for most statistical applications.

Implementations of the Mersenne Twister algorithm exist for many languages and are included in many libraries, including the Colt library used in this chapter.

The Mersenne Twister object is initialized in the usual way and can provide random draws (also called random variates) in a variety of forms, especially the `integer` and `double` forms that are used extensively throughout the remainder of this chapter.

Generating Specific Distributions

A basic random number generator is designed to draw from a uniform distribution. If draws (also called *variates*) are needed, the uniform variate, or a combination of uniform variates, is transformed into the appropriate distribution. This section introduces some of the simpler methods of drawing from these distributions in case appropriate libraries are not available. If libraries are available, they

are usually preferred because they often implement more complicated, but faster, methods of drawing random variates. In particular, they often implement the *ziggurat algorithm*, one of the fastest methods for drawing random variables.

In any case, the simple methods are all based on a good uniform random number generator, such as the Mersenne Twister discussed in the previous section:

```
import cern.jet.random.engine.MersenneTwister;
public class Distribution {
  MersenneTwister rng = new MersenneTwister();
```

Exponential and Poisson Random Numbers

When the CDF of the distribution has an invertible closed form, drawing a random number from the distribution is very easy. Because a CDF is always between 0 and 1, inclusive, a uniform random number in [0,1] is drawn and then plugged into the inverse of CDF to find the value of x.

This is the case when drawing values from the exponential distribution. Recalling the probability density function from earlier in the chapter, the cumulative density function for the exponential is simply $u=1-e^{-px}$.

Some simple manipulation to solve for x given u, yields x = -log(1-u)/p. Because u is a uniform random number, 1-u can simply be replaced with u to yield a simple implementation for drawing from the exponential distribution:

```
public double nextExponential(double p) {
    return -Math.log(rng.nextDouble())/p;
  }
```

A very similar approach can be taken when generating Poisson random variables. In the case of the Poisson distribution, the CDF contains a summation step so it is necessary to iterate through values of `k` until the appropriate value is found, as in the following implementation:

```
public int nextPoisson(double p) {
    int    k = 0;
    double c = 0, u = rng.nextDouble()/Math.exp(-p);
    while(true) {
      double x = c + Math.pow(p, k)
/Arithmetic.factorial(k);
      if(x > u) return k;
      c += x;
      k++;
    }
  }
```

Normal Random Numbers

If the cumulative density function does not have an easy-to-use closed form, it is sometimes possible to take advantage of other transforms of a uniform draw to produce random draws from a distribution. A classic example of this is the Box-Muller method for drawing from a normal distribution.

This transform works by drawing a uniformly distributed angle on a unit circle by drawing a uniform variate in [0,2pi] and drawing a radius from the chi-square distribution with 2 degrees of freedom. This is actually a special case of the exponential distribution where p = ½. This position in polar space is then converted to Cartesian coordinates, yielding two independent normal random variables (the second can be used on the next draw):

```
double  z2;
boolean ready = false;
double nextZNormal() {
  if(ready) { ready = false;return z2; }
  double theta = nextUniform(0,2*Math.PI);
  double r     = Math.sqrt(nextExponential(0.5));
  z2 = r*Math.sin(theta);ready = true;
  return r*Math.cos(theta);
}
```

This produces draws from the "Standard Normal," which has a mean of 0 and a standard deviation of 1. These can be translated and scaled to produce a draw from any normal distribution:

```
public double nextNormal(double mu,double sig) {
  return mu + sig*nextZNormal();
}
```

Gamma, Chi-Square, and Beta Random Numbers

Despite being closely related to the exponential distribution, drawing gamma random numbers is comparatively difficult. If the shape parameter k is an integer then it would be possible to simply draw k exponentials with the appropriate scale parameter and add them to get a draw from a gamma distribution.

Of course, this has a number of drawbacks, the inability to deal with non-integer values of k being chief among them. The approach used more often is a technique called *rejection sampling*.

In rejection sampling, another distribution, f(x)—which envelops the target distribution, g(x), but is easy to sample from—is selected. The envelope distribution for the gamma distribution has the following density implementation:

```
protected double gamma_f(double x,double a) {
  double L = Math.sqrt(2*a - 1);
  double y = Math.log(4) - a + (L+a)*Math.log(a)
        + (L-1)*Math.log(x) - Gamma.logGamma(a)
        - 2*Math.log(Math.pow(a, L) + Math.pow(x, L));
  return Math.exp(y);
}
```

A draw can be generated from this distribution using the inverse CDF method used to draw from the exponential distribution:

```
  protected double nextGammaF(double a) {
    double L = Math.sqrt(2*a - 1);
    double u = rng.nextDouble();
    return a*Math.pow(u/(1-u), 1/L);
  }
```

The draw from the enveloping distribution is then used as the draw from the target distribution with a probability of g(x)/f(x). Otherwise another draw is attempted until this condition is met:

```
protected double nextGamma(double k,double p) {
[[OPEN-LW-CODE80]]    while(true) {
      double x = nextGammaF(k);
      if(rng.nextDouble() <= gamma_f(x,k)/Distributions.dgamma(x, k, 1)) {
        return x/p;
      }
    }[[CLOSE-LW-CODE80]]
  }
```

Note that the closer the ratio g(x)/f(x) is to 1, the better the performance of the algorithm. Figure 9.1 shows the density for the gamma distribution with different shape parameters. The dotted line shows the density of the envelope distribution for the rejection sampling method for the same shape parameter. The choice of f(x) in this case is fairly good, as shown in Figure 9.1, but it will be inefficient to sample from the tails of the gamma distribution. Other methods have been developed to deal with this inefficiency.

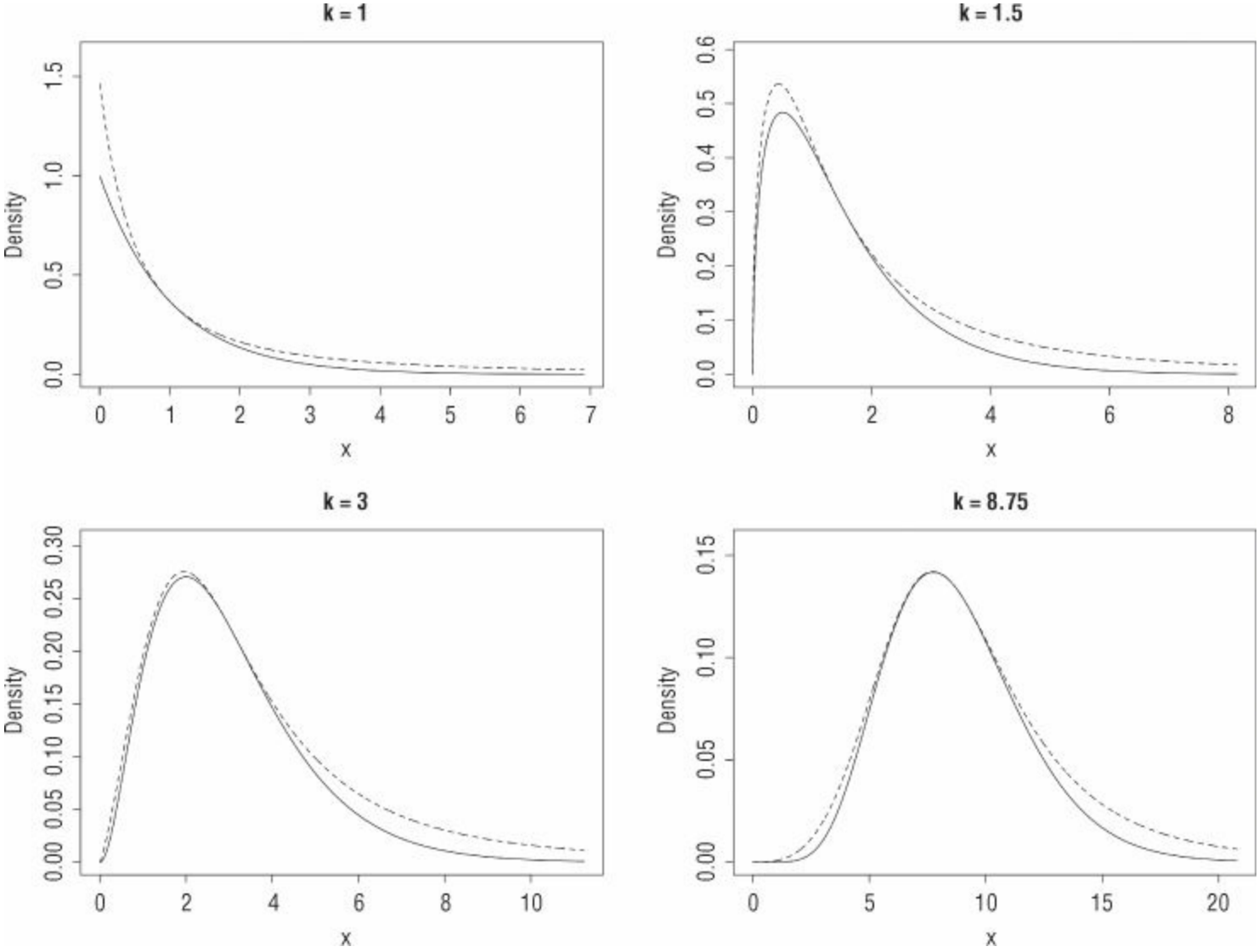


Figure 9.1

Given a method for sampling from the gamma distribution, sampling from the chi-square and beta distributions is straightforward. The chi-square, of course, is simply a special case of the gamma distribution so it can be drawn directly:

```
public double nextChiSq(double k) { return nextGamma(k/2,0.5); }
```

A beta random variable is also drawn by taking advantage of its relationship with the gamma distribution—in this case, by taking the ratio of one gamma distribution over the sum of two gamma distributions:

```
public double nextBeta(double a,double b) {
    double x = nextGamma(a,1);
    double y = nextGamma(b,1);
    return x/(x+y);
  }
```

Sampling Procedures

So far, this chapter has introduced the concept of the random variable and shown how one can generate "draws" from these distributions. When there is a large population, draws can also be generated by sampling from the population. Over the decades a number of sampling procedures and a massive body of literature have been developed. Many of these procedures are focused on the problem of surveys and polls in the situation that taking a census of the entire population is too costly, time consuming, or both. Because this book is focused primarily on streaming data, most of these procedures are beyond the scope of this book.

This section covers the basics of sampling from a fixed population insofar as it allows for understanding how to sample from a streaming dataset. From there, some modifications of the basic streaming procedure are introduced to cover some of the more interesting streaming analysis scenarios.

Sampling from a Fixed Population

The simplest form of sampling from a population is, unsurprisingly, known as *simple random sampling*. The goal of this procedure is the sample n elements from a population of N total elements such that any given element has an equal chance of being sampled. If all the elements can be held in RAM and a given element is allowed to be in the sample more than once, the sampling implementation is trivial:

```
public class SimpleRandomSample {
  static MersenneTwister rng = new MersenneTwister();
  public static <E> E[] withReplacement(E[] in,int n) {
    Object[]     sample = new Object[n];
    for(int i=0;i<n;i++)
      sample[i] = in[(int)Math.floor(n*rng.nextInt())];
    return (E[])sample;
  }
}
```

If the entire dataset fits in RAM, but an element should be sampled *without replacement*, a somewhat different algorithm is used. The base for the algorithm is the Fisher-Yates Shuffle, which was developed as a pen-and-paper method in the late 1930s. It is very simple to implement as an in-place shuffling method:

```
public static <E> void permute(E[] array) {
    for(int i=0;i<array.length;i++) {
      int j = i + (int)Math.floor((array.length - i)*rng.nextDouble());
      E temp = array[i];array[i] = array[j];array[j] = temp;
    }
  }
```

This method will produce all n! possible permutations of the array with equal probability. In the preceding algorithm, a value will only be swapped no more than once, so a sampler without replacement can be implemented by running the shuffle algorithm over the first n elements and returning the first n elements to use as the sample:

```
public static <E> E[] withoutReplacement(E[] array,int n) {
  for(int i=0;i<n;i++) {
    int j = i
+ (int)Math.floor((array.length - i)*rng.nextDouble());
```

```
      E temp = array[i];array[i] = array[j];array[j] = temp;
    }
    return Arrays.copyOf(array, n);
  }
```

Sampling from a Streaming Population

The Fisher-Yates shuffle works well when the total size of the population is known and fits into an array held in RAM. When the array cannot be held in RAM, other techniques were developed to allow for sampling from the data in a single pass. These methods form the basis of sampling from a streaming population where the complete dataset is somewhere between "very large" and "infinite."

The Reservoir Algorithm

Reservoir algorithms are a class of algorithms developed to sample from streaming populations. They are closely related in concept to the Fisher-Yates shuffle from the last section. This is most easily seen in an early version of the Fisher-Yates shuffle:

```
  public static <E> void durstenfeldPermute(E[] array) {
    for(int i=array.length-1;i>0;i--) {
      int j = (int)Math.floor(i*rng.nextDouble());
      E temp = array[i];array[i] = array[j];array[j] = temp;
    }
  }
```

In this version of the algorithm, the index is shuffled from back to front. This means that elements from farther out in the array are swapped into earlier sections. It is this notion that leads to the basic reservoir algorithm.

In the reservoir algorithm, an array of elements that form the reservoir is first allocated. This takes the role of the first n elements of the array when sampling without replacement:

```
  public class Reservoir<E> implements List<E> {
    MersenneTwister rng = new MersenneTwister();
    Object[] reservoir;
    int      N;
    public Reservoir(int n) {
      super();
      reservoir = new Object[n];
      N         = 0;
    }
```

The first `reservoir.length` elements can simply be added to the array. After that, a random position is drawn between N and 0. If that position falls into the array, it replaces the element currently there; otherwise, it is ignored:

```
  public boolean add(E arg0) {
      if(N < reservoir.length) {
        reservoir[N] = (Object)arg0;
      } else {
        long k = (long)Math.floor(N*rng.nextDouble());
        if(k < (long)reservoir.length) reservoir[(int)k] = (Object)arg0;
      }
      N++;
      return true;
    }
```

This is effectively the same thing that the Durstenfeld version of the Fisher-Yates algorithm did when

it moved backward throughout the array. The proof is not given here, but this ensures that each element in the stream has an equal probability of being in the sample at every point in the analysis.

Biased Streaming Sampling

The basic reservoir algorithm, also known as Algorithm R (the name given in Knuth's "The Art of Computer Programming"), ensures a uniform sample from the beginning of time until the time the stream stops. This is largely due to its original application, which was to sample from a "stream" in the sense of only being allowed a single pass over an otherwise finite dataset. This makes the algorithm well suited to providing samples in a batch setting where Map-Reduce is being used to process a finite dataset.

It is less interesting in the streaming sense used by this book, which is more interested in the dynamics of the sample over time. In fact, as the sample ages, it is less and less likely that a recent data point will actually be included in the sample (the probability that a random number between [0,N] is smaller than n gets smaller as N increases). Instead, it would be preferable if the algorithm could be biased toward including newer events in preference to older events.

Sliding Window Reservoir Sampling

A simple method for introducing time dynamics into reservoir sampling is to maintain a sliding window of samples. In this implementation, a number of reservoir "buckets" are maintained in a linked list:

```
public class SlidingWindow<E> {
  int n,k,max;
  int last = Integer.MAX_VALUE;
  public SlidingWindow(int n,int k,int max) {
    this.n = n;this.k = k;this.max = max;
  }
```

This implementation takes three parameters: `n` is the size of each reservoir sample; `k` is the interval between the start of each bucket; and `max` is the number of elements that are considered for each of the buckets before it expired. If all of these parameters are set to the same value then this is not so much a sampler as it is a method for bucketing the data.

On each insert, several different things may happen. If more than `k` elements have been added, a new bucket is created. Then, expired buckets are removed from the active list and placed on a `completed` queue for further processing by another thread. Finally, the element is added to each of the active buckets:

```
public void add(E object) {
  if(last >= this.k) {
    last = 0;
    live.add(new Reservoir<E>(n));
  }
  while(live.peek() != null && live.peek().total() >= max)
    completed.add(live.poll());
  for(Reservoir<E> e : live) e.add(object);
  last++;
}
```

With this method, each individual sample only contains a sample from a specific time range. These samples can then be individually processed to produce estimates that change over time.

Exponentially Biased Sampling

Although sliding windows are easy to implement, they can also use quite a bit of memory. This is particularly true if the size of the samples and the windows have a lot of overlap. It also introduces an operation that is linear in the number of live windows.

Another approach is to maintain a single sample that will tend to contain newer elements by replacing elements in the sample with newer ones according to the relative age of the two elements. This section describes one method that was presented by Charu Aggarwal ("On Biased Reservoir Sampling in the Presence of Stream Evolution," VLDB Conference, 2006).

This method begins with the definition of a bias rate `p`, which has the same functional form as the exponential distribution. The algorithm defines a reservoir by the inverse of the bias rate:

```
public class BiasedReservoir<E> {
  MersenneTwister rng = new MersenneTwister();
  ArrayList<E> reservoir = new ArrayList<E>();
  double rate;
  double size,N;
  public BiasedReservoir(double rate) {
    this.rate = rate;
    size      = Math.ceil(1.0/rate);
  }
```

Note that the reservoir is not actually an array as it was in the original formulation. This is because the number of elements in the array will be allowed to vary somewhat over time. To implement the `add()` method, first an element is removed with the probability proportional to how many elements are in the reservoir versus the size. If the number of elements is equal to the size, then an element is always removed. The element to be inserted is then added to the array:

```
public void add(E obj) {
  if(rng.nextDouble() < ((double)reservoir.size())/size) {
    reservoir.remove(rng.nextInt() % reservoir.size());
  }
  reservoir.add(obj);
}
```

Aggarwal is able to show that this simple algorithm produces exponentially biased samples with a rate equal to the "size" of the reservoir. The primary drawback of this technique is that it can take some time to fill the reservoir completely. In most streaming applications, the calculations are very long lived, so it is not as likely that the initialization of the stream is a problem.

Conclusion

This chapter has provided a broad survey of the field of statistics and probability. Although it is not particularly specific to streaming data analysis, it is the key to analyzing samples from a population. In some ways, any data stream is a sample from the population of all possible data streams, but even if the data stream is considered to be the population it can be too large to manipulate directly.

Because the stream may be too big to use directly, this chapter also introduces a few simple ways of obtaining samples. These samples can either be over a very large, but finite population or from a truly infinite stream of data. The next chapter takes a different tack, focusing on methods for summarizing the data stream in a "lossy" fashion. In statistics, this is known as *dimension reduction*. After that, the final chapter applies the techniques from this chapter and Chapter 10 to actually analyze the data rather than simply tabulate it.

Chapter 10
Approximating Streaming Data with Sketching

Questions about a set of distinct elements often arise in analytics, streaming or otherwise. Common questions are things like "Have I seen this before?" (set membership), "How many different things have I seen?" (cardinality estimation), or "How often do I see this thing?" (frequency).

When the number of different elements is small (low cardinality), this can be computed directly even in a streaming system. Some of the storage mechanisms introduced in earlier chapters, such as Redis, even have specialized data structures to allow for maintaining sets and histograms with real-time updates.

However, when the cardinality of the set becomes large (that is, there are many distinct items) direct maintenance of these sets becomes problematic. These data structures require O(log n) update time and O(n) storage, which can be infeasible in a streaming setting and expensive (at the very least) in terms of hardware costs.

To combat these problems, a number of algorithms, collectively known as *sketch* algorithms, have been developed to approximate the answers to these questions. Sketch algorithms have three features that make them desirable. The first feature is constant time updates of the data, which allows them to be easily maintained in a streaming setting. Secondly, the storage space needed is usually independent of the amount of data. Finally, querying the data structure can be completed in at worst linear time. The downside is that to achieve these properties, errors are introduced into the reported results. In this sense, they are like the sampling approaches discussed in Chapter 9, "Statistical Approximation of Streaming Data."

This chapter discusses four sketch algorithms, each with a different focus or performance characteristics. The Bloom Filter is a general set representation that can be used to answer questions about set membership and cardinality. It is also the oldest of the algorithms, and its variations are used in many areas. The two Distinct Value Sketches—Min-Count and HyperLogLog algorithms—use different approaches to approximate the size of a set, also known as the *cardinality*. Finally, the Count-Min sketch approximates a multi-set of the frequencies of the elements in the set. When the elements of the multi-set are ordered, the multi-set approximates a histogram.

This chapter starts with a review of the background concepts common to all the sketch algorithms. In particular, these algorithms make extensive use of hash functions. These functions are important in many facets of computer science, but it is their statistical properties that are useful in sketch applications. This chapter also reviews some aspects of mathematical sets. Most of the algorithms in this chapter deal with set operations, and there are some important properties of sets that are exploited to improve the performance of the algorithms.

Registers and Hash Functions

All sketches use the same building blocks to implement their particular approach: registers and hash functions. This section discusses the properties of these two building blocks, concentrating on hashes.

Registers

Registers are a straightforward concept. They are simply an array of counters that store the data for the sketch. Registers in most sketch applications are small relative to the arrays used in the direct representation. They require only constant space that does not depend on the number of input elements, typically with smaller ranges than the original input. This smaller range allows for representing registers with fewer bits than the original. For example, the basic versions of the Bloom Filter only sets registers to 1 or 0, so you can use a bit array instead of a byte array and cut space requirements by eight times.

Hash Functions

Hash functions are functions that take an input and return a value in some finite space of m output values, regardless of the length of the input. For example, a hash function can take a string, which could theoretically be infinitely long, and return, say, a 32-bit integer, giving you 2^{32} possible outputs.

Hash functions are widely used in computing, and for decades people have researched and developed a variety of hash functions. All hash functions are deterministic: The same input will always result in the same output. They also try to be uniformly distributed such that random inputs produce a uniform distribution among the outputs.

For sketch applications, speed is the primary requirement for any hash function, particularly when used in a streaming setting. This mostly eliminates cryptographic hash functions such as MD5 and SHA1 from contention. Although these hashes are very random, they are intentionally designed to be slow to compute. Despite this fact, cryptographic hashes are often used in real-world implementations because they are built in to many programming languages' common libraries.

Rather than use cryptographic hashes, most sketch implementations use a variety of fast hash functions that better suit their performance needs. These include the FNV hash, the Jenkins hash, and the MurmurHash. The Fowler, Noll, and Vo (FNV) hash function was developed in 1991 with variants supporting output spaces of between 32 and 1,024 bits. It has one of the simplest implementations of any hash functions, relying on two constant values chosen from a table for the number of bits to be generated. For example, the 64-bit Java hash is implemented as follows:

```
static long fnv64(byte[] input) {
  long output = 0xcbf29ce484222325L;
  for(var i=0;i<input.length;i++) {
    output ^= (long)input[i];
    output *= 0x100000001b3L;
  }
  return output;
}
```

The FNV hash works quite well on fairly small inputs and is quite fast on Intel hardware, but the other two hash functions—Jenkins and Murmur—are often reported to have better real-world performance. The MurmurHash seems to perform well in real-world applications and has 32-, 64-, and 128-bit variants available. Code for the Murmur hash is not provided here—it involves large

constant arrays that would be meaningless to reproduce—but the code provided with this book makes use of the Java implementation provided by the Colt library. The Murmur code is also widely available for other languages under a variety of licenses.

Independent Hash Functions

Many algorithms call for several independent, or pairwise independent, hash functions. Creating hash functions with these properties is a well-studied part of computer science, and it turns out to be a difficult thing to do. Instead, most actual implementations rely on the much more easily generated *universal hash functions*. These are hash functions that are simply selected from the same family at random.

This happens in one of two ways. The first, and most obvious method, is when the hash function takes an initial seed value, much like a random number generator (the hash functions used in this chapter are similar in implementation to many random number generators). To generate k hash functions, k seeds are selected uniformly from the possible space of initial values (usually an integer the same size as produced by the hash function).

The other way these hash functions are generated requires only a single seed value to initialize the hash function. This is used to generate the first hash value, which is then used as the initial value to generate the next hash function, until k hash values are produced. For example, the previously mentioned FNV implementation can be modified to produce any number of output hash values for the same input as follows:

```
public class FNV {
  public static final long INIT  = 0xcbf29ce484222325L;
  public static final long PRIME = 0x100000001b3L;
  long   hash;
  byte[] input;
  public void set(byte[] input) {
    hash = INIT;
    this.input = input;
  }
  public long next() {
    for(int i=0;i<input.length;i++) {
      hash ^= input[i];
      hash *= PRIME;
    }
    return hash;
  }
}
```

Double-Hashing

To speed things further, Kirsch and Mitzenmacher showed in 2007 that you can use a trick called double hashing to generate as many hash functions as you like using only two rounds of the base hash function. They also showed that the difference in performance using this method instead of k independent hash functions was small, making it appropriate for use in real-world settings.

Double hashing is normally used to resolve collisions in hash tables and defines a hash function g(x,i) = h1(x) + i*h2(x) + i⋀2. In this case, h1 and h2 are two rounds of the earlier original hash function, and i is an integer that ranges from 0 to k-1. In this case k is the desired number of hash functions. After computing the first two hash values, it is now very inexpensive to compute any further required hash values with only two multiplications rather than further rounds of expensive hash function

calculation.

Hash Collisions and the Birthday Paradox

One of the more famous probability parlor tricks is the so-called Birthday Paradox. It states that when you get a group of more than 23 people together there will be a 50 percent chance that at least two people share a birthday. By the time you have 60 people, there is a better than 99 percent chance that at least two of them will share a birthday.

When you think of birthdays as a simple hash function that maps each person into a number between 1 and 365, then sharing a birthday is basically a hash collision. Knowing the mathematics behind the paradox is useful for understanding many of the calculations in the next sections. Also, it's a nifty trick to use at parties.

To start, assume that birthdays are evenly distributed through the year, just like hash functions uniformly distribute their values. In reality, birthdays are not quite evenly distributed throughout the year, and leap years are a problem, but that is a discussion for a different book.

The easiest way to compute the probability that at least two people share a birthday is to compute the probability that nobody in the room shares a birthday. In this case the first person can choose whatever day she wants so she can have any possible birthday A. The second person cannot choose the birthday the first person chose, so he chooses a birthday B from one of the 364 remaining birthdays. The third person chooses birthday C, and so on up to n people. Writing down the probability is:

$$P(\text{nobody shares}) = (365/365)*(364/365)*\ldots*((365-n-1)/365)$$

rewritten as,

$$P(\text{nobody shares}) = 365!/365^n(365-n)!$$

Where ! represents the factorial function, the product of all integer values from 1 to n. So, 5! Is $5\times4\times3\times2\times1$.

Finally, the probability that at least two people share a birthday is simply 1-P(nobody shares):

$$1-365!/365^n(365-n)!$$

The final equation contains factorial functions that are expensive to compute. To make the calculation easier, an approximation is often used. The equation (365-n-1)/365 can be rewritten as (1 – (n-1)/365) and the fact that ex is approximately 1+x when x is much smaller than 1 (via the first order Taylor expansion of e^x). Combining the two you see that $e^{-(n-1/365)}$ is approximately equal to (1-(n-1)/365). You can then rewrite the probability as:

Knowing that the sum of integers between 1 and n-1 is n×(n-1)/2 further simplifies the formula to:

If n is large enough, n*(n-1) can be replaced with the further approximation n^2 because the relative difference between n and n-1 will be very small. Setting this equation to be 0.5 and solving for n yields the famous result of 23 people needed to have a better than 50 percent chance of sharing a birthday.

Working with Sets

Sketch algorithms fundamentally deal with sets, particularly approximating the size of a set, which is called cardinality estimation. This section reviews some aspects of "naïve set theory" that is useful when using or analyzing the algorithms discussed in the next several sections.

A *set* is simply a collection of distinct elements. A set can be empty, which is called the *empty* or *null* set. Uppercase letters such as A or B are usually used to denote sets. The null symbol (Ø) is used to denote the empty set.

NOTE

This book uses some basic mathematic terms and formulas. If your math skills are rusty and you find these concepts a little challenging, a helpful resource is *A First Course in Probability* by Sheldon Ross.

Sets can be combined in three basic ways. The *union* of a set, written A ∪ B, is the set that contains all of the elements in A and all of the elements in B. The *intersection* of a set, written AB, is the set the contains all of the elements in A that are also elements of B. Finally, the *complement* of the set, written "A \ B", is the set that contains elements of set A that are *not* in set B.

In Java, the basic union, intersection and complement methods are `addAll()`, `retainAll()`, and `removeAll()`, respectively. These primitive methods can be used to implement union and intersection for an arbitrary number of sets as follows:

```
public static Set<E> union(Set<E>...sets) {
  Set<E> union = new Set<E>();
  for(Set<E> s : sets) union.addAll(s);
  return union;
}
public static Set<E> intersection(Set<E>...sets) {
  Set<E> intersection = new Set<E>();
  intersection.addAll(sets[0]);
  for(int i=1;i<sets.length;i++)
    intersection.retainAll(sets[i]);
  return intersection;
}
```

The number of elements in a set is called the *cardinality* of the set. It is denoted by enclosing the set in | symbols. For example, the cardinality of the set A is written as |A|. In Java, the `size()` method returns the basic set cardinality.

Computing the cardinality of the set union or intersection is somewhat more interesting. In most of the algorithms given later in this chapter, computing the union of two or more sets is trivial, whereas computing the intersection of two or more sets is difficult (if it is possible at all). The intuition for this behavior is that all of the following algorithms support adding new elements to a set. This is equivalent to computing the union of the original set and the set containing only the new element to be added. However, most of the algorithms do not support removing the elements from the set, which would be required to compute the intersection. Additionally, many of the algorithms support computing the cardinality of the union without having to explicitly compute a representation of the union, making the calculation fairly inexpensive.

To compute the cardinality of an intersection of sets requires taking advantage of the *inclusion-exclusion* principle. The simplest form of this principle relates the cardinality of the union of two sets with the cardinality of the intersection:

$$|A \cup B| = |A| + |B| - |A \cap B|$$

Essentially, this relationship says that if the cardinality of A and the cardinality of B are added, the elements in the intersection have been double counted. To correct for this, subtract the cardinality of the intersection. Rearranging this equation gives a formula for the size of an intersection:

$$|A \cap B| = |A| + |B| - |A \cup B|$$

The same can be done for the intersection of three sets using the same principle:

$$|A \cap B \cap C| = |A \cup B \cup C| - |A| - |B| - |C| + |A \cap B| + |A \cap C| + |B \cap C|$$

Substituting from the first equation to eliminate the other intersections yields a final equation:

$$|A \cap B \cap C| = |A \cup B \cup C| + |A| + |B| + |C| - |A \cup B| - |A \cup C| - |B \cup C|$$

This can be further generalized to provide an expression for the cardinality of the union of any number of sets: Add the cardinality of all the sets. Subtract the cardinality of the intersection of each pair of sets. Add the cardinality of the intersection to each combination of three sets. Subtract the cardinality of the intersections of each combination of four sets, and so on.

Although this allows for the approximation of arbitrarily large intersections of sets, it has two drawbacks:

- The total number of operations required grows large very quickly.
- The Bloom Filter and Distinct Value sketch algorithms presented in this chapter each have errors associated with them, and the errors accumulate with each addition or subtraction. Even the intersection of two sets will have an estimation error three times larger than the error in the estimated size of any one set or any union of sets.

The Bloom Filter

Bloom filters are a data structure used by a variety of applications to store set membership information. This data structure is compact and does not depend on the number of items to be stored in the set. The tradeoff is that the Bloom Filter may incorrectly report that an item is in the set when it is not—a false positive. It will never report a false negative. It is this false positive rate that is controlled by the size of the data structure.

The Algorithm

The Bloom Filter approximately represents a set using an array of m 1-bit registers and k independent hash functions. In Java, a BitSet and k MurmurHash functions, each seeded with independent random values, can be used to implement the `java.util.Set<E>` interface:

```
public class BloomSet<E extends Serializable> implements Set<E> {
  BitSet               bits;
  int                  m;
  SerializableHasher[] hashes;
  ObjectOutputStream[] outputs;
  protected void initialize(int m,int[] seeds) throws IOException {
    this.m = m;
    bits   = new BitSet(m);
    hashes = new SerializableHasher[seeds.length];
    outputs= new ObjectOutputStream[seeds.length];
    for(int i=0;i<seeds.length;i++) {
      hashes[i]  = (new SerializableHasher()).seed(seeds[i]);
      outputs[i] = new ObjectOutputStream(hashes[i]);
    }
    bits.clear();
  }
  public BloomSet(int m,int[] seeds) throws IOException {
    initialize(m,seeds);
  }
```

The `SerializableHasher` and `ObjectOutputStream` arrays implement the 32-bit MurmurHash using Java's serialization mechanism to make it possible to use this class for any object that implements `Serializable`.

In practice, it is common to also implement specialized sets for storing things like `String` or `Long` values. These data types are very common in streaming analysis, so having a specialized class can result in a large performance improvement. In fact, the MurmurHash implementation used in the sample code provided with this book has specialized versions of the hash optimized for these use cases.

Adding a new element to the set is easy. First, each hash function is used to generate k different hash values. These hash values are mapped into the array of m registers using the modulus operator. The bit at that position in the register array is then set to 1. A Java implementation is as follows:

```
public boolean add(E arg0) {
  if(!contains(arg0)) {
    for(int i=0;i<outputs.length;i++) {
      hashes[i].reset();
      try {
        outputs[i].writeObject(arg0);
        outputs[i].flush();
```

```
          bits.set(hashes[i].hash() % m);
        } catch(IOException e) { }
      }
      return true;
    }
    return false;
  }
```

Part of the preceding implementation includes a check for the element using `contains` so that the method correctly returns whether this is considered to be a new element. An element can only be considered to be in the set if all the registers indicated by the k hash functions are set to 1. You can easily check this with the following implementation:

```
  public boolean contains(Object arg0) {
    for(int i=0;i<outputs.length;i++) {
      try {
        hashes[i].reset();
        outputs[i].writeObject(arg0);
        outputs[i].flush();
        if(!bits.get(hashes[i].hash() % m)) return false;
      } catch(IOException e) { }
    }
    return true;
  }
```

It is easy to see how this implementation can lead to false positives. As more and more elements are added to the set, it is more and more likely that at least one of the elements will map to each of the registers of this new item to enter into the set. When that happens, the implementation of `contains()` returns `true` for the item even when it hasn't been entered.

Filtering Duplicate Events in Online Advertising

In online advertising, it is a fairly common practice to assign a unique identifier to each viewing of an ad. This unique identifier is used to identify situations where an ad view or click may register more than once in the system. The reasons that this happens range from the mundane case where a user habitually double-clicks links to malicious "bots" that attempt to generate revenue by placing ads on fraudulent sites.

The advertising industry is generally conservative when it comes to validated page views (also known as impressions) and clicks, so it is generally preferable to err on the side of caution and throw away a small percentage of valid events if it means cleaning out more of the invalid events. The industry has also taken a general turn toward so-called "programmatic buying," where ads are traded across the Internet in exchanges similar to a real-world commodity exchange. This happens via a bidding mechanism that typically completes in less than 100 milliseconds.

With a large number of unique identifiers and the need to maintain near real-time counts of impressions and clicks to allow for optimal bidding, filtering these duplicate events is a perfect application of data structures like the Bloom Filter.

An initial approach might be to maintain a Bloom Filter for views and another filter for clicks. However, views happen 100 percent of the time, whereas clicks occur perhaps 1 percent to 5 percent of the time. For the purposes of accounting, it is still desirable to filter impressions, but this can likely be done offline. Clicks, on the other hand, are used to determine bids, so you can maintain a single Bloom Filter to filter out duplicate clicks that would artificially inflate the apparent quality of an ad placement.

Choosing a Filter Size

The likelihood that the scenario mentioned at the end of the last section occurs is clearly a function of three parameters: the number of elements inserted into the set (n), the size of the bit set itself (m), and the number of hash functions used to set bits (k). The algorithm's user selects the latter two parameters, and the former can typically be estimated. This section discusses the selection of these two parameters and their effect on the false positive rate.

From before, to correctly identify an element as *not* being in the set, at least one of the bits identified by the k hash functions must be zero. If the hash functions are independent and uniformly distributed, the probability p that a bit will still be zero after *n* elements have been inserted into the filter is $p=(1-1/m)^{kn}$. This equation is approximated by $p=e^{-kn/m}$. The probability of a false positive is approximately $(1-p)^k$, the chance that all k registers being checked will be set. This yields a final equation of $(1-e^{-kn/m})^k$.

The goal is to make p as small as possible, which happens when $k = (m/n) \times \ln 2$. The number of hashes to use then depends on the size of the filter and an idea of how many items will be entered into the Bloom Filter. Substituting this into the equation above yields:

$$P = (1-e^{-\ln 2/n)(m/n)*\ln 2}$$

Solving for m yields the equation:

$$m = -n \times (\ln p)/(\ln 2)^2$$

Setting n to 1, the number of bits required to store a single value is given as approximately 2.08*ln(1/p) or 1.44*log2(1/p). From this approximation, a false positive rate of 5 percent requires 6.22 bits of storage per element to be entered into the set. Halving that rate only increases the storage per element to 7.7 bits and a false positive rate of 1 percent only requires 9.6 bits per storage.

Of course, register arrays must be allocated in a round number of bits. So, rather than specify the error rate, it is also common to specify a number of bits-per-element c, and then compute the size of the bit array as well as the number of hash functions from that. The total size of the array given a target number of elements is m = cn. The number of hash functions to use is k = c × ln 2. The expected false positive rate can be quickly approximated as $p = 0.6185^c$. For example, 8 bits per element yields k = 6 and an approximate false positive rate of 2.14 percent.

This may not seem like much improvement, but in real-world applications it can be enormous. For example, domain names can be at most 63 characters long with 3 characters as the practical minimum length. Anecdotal evidence suggests that domain names have a median length of about 10 characters. An application that needs to track a million domain names would need somewhere around 10MB of storage. A Bloom Filter to track those same million domains with an error smaller than 2.5 percent would require less than 1MB.

Unions and Intersections

Bloom Filters have the nice property that, if they have been constructed using the same hash functions, it is possible to construct unions and intersections of the two filters using bitwise OR and bitwise AND operators respectively.

In the case of the union, this operation is exactly equivalent to constructing the set directly with the same false positive rates as if the filter had been constructed directly from the union of the original sets. This is not surprising since the jth operation is essentially the union of a Bloom Filter containing a single element with the existing Bloom Filter containing the previous j-1 inserts. This is convenient as it allows filters to be constructed in a distributed fashion and later combined through the exchange of the much smaller filters.

The intersection of two Bloom Filters presents a more complicated scenario. In this case, the intersection of the two filters will have a false positive rate that is at most the largest of the false positive probabilities of the original Bloom Filters, which may be larger than the false positive probability of the intersection had it been constructed directly from the intersection of the original sets.

Cardinality Estimation

Although there are more efficient methods of maintaining an estimate of a set's cardinality, the Bloom Filter can also approximate cardinality.

The basic trick is to go back to the probability that a particular bit is set to 1. When choosing an

optimal filter size and number of hash functions, the probability that a bit was set was $p = 1-e^{-kn/m}$.

Assuming that the probability of each bit being set to 1 is independent, the register array can be considered m independent, identically distributed Bernoulli trials. This is the same as flipping a coin m times and counting how many coins come up heads.

After inserting n elements, the probability of a 1 can be estimated by the number of 1 bits x divided by the total number of bits m. Plugging x/m into the equation and solving for n yields:

$$N = -m \times \log(1-x/m)/k$$

A slightly better estimate can be obtained by not using the approximation from the earlier analysis. This yields a somewhat more complicated expression for n:

$$N = \log(1-x/m)/(k \times \log(1-1/m))$$

However, practically speaking, the difference between -m/k and 1/k × log(1-1/m) is going to be quite small.

The naïve implementation of the `size()` method of the Bloom Filter `Set` simply uses `BitSet`'s native `cardinality()` method:

```
public int size() {
  int X = bits.cardinality();
  return (int) (Math.log(1 - (double)X/
  (double)bits.length())/
  (Math.log(1.0 - 1.0/
  (double)bits.length())*(double)hashes.length));
}
```

The implementation of `cardinality()` typically uses parallel counting tricks to improve performance, but it will still operate in O(m) time. If this method is going to be called many times, better performance is achieved by integrating the computation of `X` into the `add()` method as follows:

```
public boolean add(E arg0) {
  if(!contains(arg0)) {
    for(int i=0;i<outputs.length;i++) {
      hashes[i].reset();
      try {
        outputs[i].writeObject(arg0);
        outputs[i].flush();
        int h = hashes[i].hash() % m;
        X += bits.get(h) ? 0 : 1;
        bits.set(h);
      } catch(IOException e) { }
    }
    return true;
  }
  return false;
}
```

The `reset()` method is also modified to set `X` back to zero when the filter is reset.

Real-Time Identification of Fraudulent Websites

One way a fraudulent website generates traffic is by employing a so-called “bot network” to generate large amounts of essentially fake traffic for their website. This type of traffic can actually be quite difficult to detect because most of the devices in the network are legitimate computers on the Internet that have the misfortune of being infected with malware. In fact, under normal circumstances the legitimate user of the device is simply using the device in a normal and nonfraudulent way, and it is only when the network is activated that the traffic can be identified as fraudulent.

To combat this problem, you want to blacklist certain websites in real-time when they exhibit behavior consistent with the application of a botnet to their website to generate revenue.

To do this you first generate a base profile of botnet traffic by taking the patterns of IP addresses of visiting sites that you have previously identified as being fraudulent through manual investigation or through one of the industry sources devoted to maintaining this data.

Next you need a distance metric that allows you to compare two IP address profiles. One such metric could be the Jaccard Index, which is defined as the size of the intersection of the two sets being compared divided by the size of the union of the two sets being compared.

If you maintain your website profiles as Bloom Filters along with a Bloom Filter representing your botnet profile then you can estimate the cardinality of the union of the two filters quite easily using the equation just derived for cardinality applied to the union of the two sets. In fact, since you only need the count of bits set to 1, you do not actually need to compute the Bloom Filter of the union—simply count the bits that are set in one or the other.

To compute the cardinality of the intersection, you can take advantage of a property of set cardinality: $|A \cup B| + |AB| = |A| + |B|$. With a little rearrangement, the cardinality of the intersection is simply $|A|+|B|-|AB|$. Now we can estimate the Jaccard Similarity of any website's traffic pattern to the botnet profile. You can immediately blacklist or flag for investigation those websites that are too similar.

Interesting Variations

The Bloom Filter has been around a long time and this has led to a number of variants of the original algorithm. This section covers two of the more interesting variations. In particular, these two variations address issues that tend to arise in streaming situations. In these cases, the basic Bloom Filter is likely to become saturated over time, making it somewhat useless.

Counting Bloom Filters

One of the more useful variants is the Counting Bloom Filter. This variant was introduced to overcome the fact that elements cannot be removed from the basic Bloom Filter implementation.

In this variant, the hashing part of the algorithm is identical to the original algorithm. It also has the same criteria for selecting the number of registers and the number of hash functions.

However, the registers in this variant are no longer single bits but counters. The number of bits assigned to each counter is quite small; 4 bits have been shown to be adequate for most applications.

To add an element, the counter is simply incremented rather than setting the bit to 1. To remove an element, the counter for each of the k hash functions is decremented *unless* the counter has reached its maximum value. In this case, the counter is considered to have overflowed and left permanently at its maximum value.

Allocating a generous 8 bits to each counter, the `add(E arg0)` method only needs a simple change to increment the counter when it has not yet overflowed:

```
public boolean add(E arg0) {
  boolean added = false;
  for(int i=0;i<outputs.length;i++) {
    hashes[i].reset();
    try {
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      int j = hashes[i].hash() % m;
      if(counters[j] < Byte.MAX_VALUE)
        counters[j]++;
      if(counters[j] == 1) added = true;
    } catch(IOException e) { }
  }
  return added;
}
```

The `remove(E arg0)` method is also easily implemented, taking into account the fact that the counter has overflowed:

```
public boolean remove(E arg0) {
  boolean removed = false;
  for(int i=0;i<outputs.length;i++) {
    hashes[i].reset();
    try {
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      int j = hashes[i].hash() % m;
      if(counters[j] > 0 && counters[j] < Byte.MAX_VALUE)
        counters[j]--;
      if(counters[j] == 0) removed = true;
    } catch(IOException e) { }
  }
  return removed;
}
```

Implementing the test for set membership is then as simple as making sure all the counters are positive —the same as the original Bloom Filter implementation:

```
public boolean contains(Object arg0) {
  for(int i=0;i<outputs.length;i++) {
    try {
      hashes[i].reset();
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      if(counters[hashes[i].hash() % m] == 0) return false;
    } catch(IOException e) { }
  }
  return true;
}
```

Finally, the cardinality calculation merely checks to see whether the counter is positive:

```
public int size() {
  double X = 0.0;
  for(int x : counters) if(x > 0) X += 1.0;
  return (int) (Math.log(1 - (double)X
```

```
    /(double)counters.length)
    /(Math.log(1.0 - 1.0
    /(double)counters.length)*(double)hashes.length));
}
```

Stable Bloom Filters

The Stable Bloom Filter addresses the deletion of elements from the Bloom Filter in the context of a time series. As mentioned briefly, if a Bloom Filter is allowed to simply acquire elements from a stream of data, it is possible that the Bloom Filter will simply saturate and report `true` for all set membership queries as well as report a maximum set cardinality.

The Stable Bloom Filter overcomes this by introducing the notion of aging into the filter itself. The basic structure of the filter uses counters as with the Counting Bloom Filter. To simplify things, the implementation of the Stable Bloom Filter extends the Counting Bloom Filter with two extra parameters `d` and `C`. These two parameters define the number of counters to decrement before each addition and the maximum value of each counter, respectively. This is used to "age" the filter by ensuring that values that are not incremented will eventually return to zero, as in the code example below:

```
public class StableBloomSet<E extends Serializable>
  extends CountingBloomSet<E> {
  int  d = 0;
  byte C = Byte.MAX_VALUE;
  public StableBloomSet(int d,byte C,int m, int[] seeds)
    throws IOException {
      super(m,seeds);
      this.d = d;
      this.C = C;
  }
```

In the theoretical analysis of the algorithm, the `d` counters are chosen at random to be decremented. However, this is quite slow and the only requirement from the analysis is that each counter has a d/m chance of being decremented at each round. Therefore, the authors choose a single point k at random and then decrement the subsequent d-1 counters, as shown in the following code:

```
private final MersenneTwister random = new MersenneTwister();
public void decrement() {
  int k = random.nextInt();
  for(int i=0;i<d;i++)
    if(counters[(k+i) % m] > 0) counters[(k+i) % m]--;
}
```

The update step first calls this decrement function to "age" the filter. It then updates the filter with the new element by setting the appropriate counts to their maximum values rather than simply incrementing them:

```
 public boolean add(E arg0) {
   decrement();
   for(int i=0;i<outputs.length;i++) {
     hashes[i].reset();
     try {
       outputs[i].writeObject(arg0);
       outputs[i].flush();
       int j = hashes[i].hash() % m;
       counters[j] = C;
     } catch(IOException e) {
     }
```

```
    }
    return true;
  }
```

The calculation of set membership and estimates of the size of the filter remain the same.

The downside of this approach is that it introduces the ability to produce false negatives. To minimize the probability of a false negative in practice, the authors of the original paper conducted a number of empirical experiments using different values of `d` and `C`. The general rule of thumb is to keep `C` as small as possible, because the larger `C` gets the larger `d` needs to be. This introduces more steps in each update and slows the overall algorithm. After a size for the counters has been established, the number of elements to decrement is set via empirical testing of the sketch.

Distinct Value Sketches

The Bloom Filter is a fairly flexible algorithm for approximating set membership and cardinality. To only approximate the cardinality of the set it is possible to improve the Bloom Filter's error while decreasing memory requirements using a specific Distinct Value (DV) Sketch. A variety of DV sketch algorithms have been developed over the years. The various approaches usually take on one of two different flavors, and this section discusses one of each type.

The Min-Count Algorithm

The first type category, represented by the *Min-Count algorithm* (Frédéric Giroire, "Order Statistics and Estimating Cardinalities of massive Data Sets," Discrete Applied Mathematics, 2009, Vol 157 Issue 2 (January), 406-427), relies on the distribution of order statistics. To understand how this works, recall the earlier discussion of hash functions. A good hash function takes an input value and maps it into a space of 2^p values such that the distribution of values is uniform in $[0,2^p-1]$.

This uniform distribution gives the probability that an input will hash to a value smaller than x as $x/2^p$. Assuming n unique hash values $S_1,..,S_n$ have been observed, take X to be the smallest of these values. The probability that X is at least as small as some value x is 1 minus the probability that *all* the values of S are larger than x:

$$P(X <= x) = 1 - ((1-P(S_1 < x)) * (1-P(S_2 < x)) * \ldots * (1-P(S_n < x))$$

Because the distributions of all the S values are the same, this simplifies to:

$$P(X<=x) = 1-(1-P(S_1<x))^n$$

Using this to compute the expected value of X in terms of n yields:

$$E[X] = 2^p/(n+1)$$

If M is then the observed minimum after inserting n elements, the Delta Method from Chapter 9 allows for the calculation of E[1/X] as approximately equal to 1/E[X]. Solving for n in terms of M yields:

$$n = 2^p/M - 1 \sim 2^p/M$$

This approximation is easy to calculate, but has an obvious problem that M can be 0 where the Delta Method calculation is not defined. Additionally, the variance of such approximations is often higher than desired. To get around these problems, Min-Count introduces two techniques to improve the error of the approximation as well as deal with the fact that the approximation blows up when M gets very small.

Taking the second problem first, the algorithm deals with the problem in a very simple way. Instead of simply recording the minimum, the algorithm keeps track of the k smallest values and then performs its computations on the k^{th} smallest value, which does have a defined expectation. This requires k more storage than the minimum, but k is usually a relatively small number. In fact, the simple version of the algorithm works best when k = 3.

To improve the error, one approach would be to use the same approach employed by the Bloom Filter, introducing *m* different hash functions to produce *m* independent streams of data. Taking the arithmetic mean of these *m* independent streams yields the same expectation as a single stream, but with a standard deviation proportional to thereby improving the standard error. Using m independent hash functions is computationally expensive, so Min-Count uses a technique called *stochastic averaging*, which was introduced by Flajolet in 1985 to simulate independent streams.

To simulate independent random streams, the first b bits of the hash value are used to define 2^b registers where b is typically smaller than half the number of bits used in the hash function. If each register that stores the three smallest is normalized to be a number in [0,1] the cardinality estimate becomes:

$$N = 2^*(1/M_1)+(1/M_2)+\ldots+(1/M_m)$$

with a standard error of . To use a concrete example, a 32-bit hash function with 65,536 registers would require 384k of storage. By contrast, a sampling approach using the same 32-bit hash to reduce the storage requirements would be able to store about 98,000 values in the same amount of space, which may not be sufficient if some of the distinct values are quite rare and the data stream very large.

Implementing Min-Count

Like the Bloom Filter implementation, the Min-Count implementation uses the standard Java Set interface and works on serializable elements. This implementation also uses the third smallest value to avoid the issues mentioned in the previous section. Using a parameter b to define the number of bits to use and defining 2b registers, the class is initialized as follows:

```
public class MinCount<E extends Serializable> implements Set<E> {
  int   b;
int[]              M;
SerializableHasher hash = new SerializableHasher();
ObjectOutputStream out;
public MinCount(int b) throws IOException {
this.b = b;
int m  = (int) Math.pow(2, b);
this.M = new int[3*m];
out    = new ObjectOutputStream(hash);
reset();
  }
```

Implementing the `add()` method is straightforward. Simply hash the input value and then update the three smallest values:

```
public boolean add(E arg0) {
  try {
hash.reset();
out.writeObject(arg0);
out.flush();
int h = hash.hash();
int m = (h >>> (32-b));
int v = (h << b) >>> b;
if(M[m] > v) {
M[m+2] = M[m+1];
M[m]   = v;
} else if(M[m+1] > v) {
M[m+2] = M[m+1];
M[m+1] = v;
} else if(M[m+2] > v) {
M[m+2] = v;
}
return true;
} catch(IOException e) {
return false;
}
}
```

The only thing that remains is to implement the `size()` method to produce a cardinality estimate:

```
public int size() {
  double range    = Math.pow(2,32-b);
  double estimate = 0.0;
  for(int i=0;i<M.length/3;i+=3)
    estimate += range/(double)M[i+2];
  return (int)(2.0*estimate);
}
```

Computing Set Similarity

Min-Count sketches can also be used to compute the approximate similarity between two sets using a technique called *Min-wise Hashing*. The basic idea is to find an approximation to the Jaccard Similarity metric used in the "Real-Time Identification of Fraudulent Websites" sidebar earlier in the chapter.

The key observation is that the probability that the minimum of the hash of one set is equal to the minimum of the hash of another set is the ratio of the size of the intersection of the set over the size of the union of the set. This is exactly the Jaccard Similarity metric from the "Real-Time Identification of Fraudulent Websites" sidebar.

Like most sketches, the approximation using only a single hash function is a very rough approximation. The original variant of the algorithm used multiple hash functions to overcome this limitation, much like the Bloom Filter. Of course, this is a very expensive operation, requiring at least 400 hash functions to achieve an error of 5 percent. Later variations use a single hash function and then subsample the hashes into various streams, just like the Min-Count sketch. In either case, the Jaccard Similarity is then the proportion of matching registers between two sets that have been hashed using the same mechanism:

```
public double distance(MinCount other) {
  double x = 0.0;
```

```
    for(int i=0;i<M.length;i++) {
      if(other.M[i] == M[i]) x += 1.0;
    }
    return x/(double)M.length;
  }
```

The HyperLogLog Algorithm

The *HyperLogLog*, introduced by Flajolet (of stochastic averaging fame in the "The Min-Count Algorithm" section) in 2007 is the latest in a long line of algorithms including LogLog and SuperLogLog. These algorithms all use the expected distribution of bit patterns to estimate cardinality rather than the order statistics of Min-Count and related algorithms. In the case of HyperLogLog, the largest run of leading zeros in the hashed value is recorded in the register rather than the smallest or largest value.

To understand how this works, recall that the hash function is supposedly uniformly distributed. This should mean that each bit of a k bit hash value is the same as the flip of a fair coin. From this, the probability that the first bit is a 1 is 50 percent. In other words, if two distinct hash values are observed, it is expected that one of them will start with a 1. The same logic can be applied to other patterns of leading zeros. A hash value starting with 01 has a probability of 25 percent, so it is expected to be observed once in four distinct hash values. In general, a hash pattern with r leading zeros has a probability $p = 2^{1-r}2^{-1} = 2^{-r}$ and an expected rate of observing once in $1/p = 2^r$ distinct values.

Like the Min-Count algorithm, HyperLogLog makes use of stochastic averaging to improve the performance. The stream is once again divided into 2^b streams, where b is between 4 and 16. Each register then holds the highest position of the first 1 and provides a rate of $2^{M[i]}$ for each stream. Because these are rates, the algorithm computes a normalized harmonic mean, which is useful when computing an average rate:

$$N = a(b)\times m^2/(2^{-M[1]}+2^{-M[2]}+\ldots+2^{-M[m]})$$

The function a(b) in this case is an adjustment factor that Flajolet uses to correct for the bias in the estimate. This function is an integral, but there are simple formulas given for the valid range of b. When b is 4,5,6, the function a(b) is considered to be constant with values 0.673, 0.697, and 0.709, respectively. When b is between 7 and 16, $a(b) = 0.7213/(1+1.079/2^b)$.

Although this estimator works well when the cardinality is in an "intermediate" range, there are further corrections to be performed when the cardinality is very small or large.

When the raw estimate is smaller than 5×m/2 there are likely to be registers that have not been used. Using V as the count of unused registers, if V is greater than 0 then N is better estimated by:

$$N^* = m\times\log(m/V)$$

This is actually very similar to the result obtained by a different algorithm called Linear Counting.

Similarly, when the cardinality gets very high, the introduction of hash collisions will begin to undercount the number of distinct items. A correction for this is introduced when the raw estimate exceeds $2^{32}/30$ (cardinalities of roughly 143 million):

$$N^* = -2^{32} \times \log(1 - N/2^{32})$$

Using these corrections, the relative error of the cardinality estimates is approximately $1.04\sqrt{(m)}$.

Implementing HyperLogLog

The immediately obvious benefit of using HyperLogLog compared to Min-Count is that the storage space required for the registers is much smaller. Rather than having to store the smallest hash value, the algorithm only needs to store the largest number of leading zeros. For a 32-bit hash function, that number can only be as large as 32 and only requires 5 bits to store rather than Min-Count's 32 bits for the same size hash function. That works out to 85 percent storage right off the bat. Of course, it is usually easier to just allocate 8 bits to each register, which only uses 75 percent less storage.

The algorithm is well defined when between 2^4 and 2^{16} registers are used, so this is enforced when creating the object:

```
int m;
public HyperLogLog(int b) throws IOException {
  output = new ObjectOutputStream(hash);
  this.b = b;
  if(b < 4) b = 4; else if(b > 16) b = 16;
  m = 1 << b;
  M = new byte[m];
  for(int i=0;i<m;i++) M[i] = 0;
  V = M.length;
  switch(b) {
    case 4: alpha = 0.673*m*m;break;
    case 5: alpha = 0.697*m*m;break;
    case 6: alpha = 0.709*m*m;break;
    default: alpha = (0.7212/(1+1.079/m))*m*m;
  }
}
```

Note that some of the constants used in the cardinality computation are precomputed here to save later processing.

Next comes the `add(E arg0)` method. Because it is expected that the size of the set will be requested frequently, this implementation maintains the sum of registers directly. This removes the O(m) computation from the cardinality estimate. For the same reason it also maintains the count of empty registers for use with the small cardinality correction. This gives a slightly more complicated implementation and Min-Count:

```
public boolean add(E e) {
  try {
    hash.reset();
    output.writeObject(e);
    int x = hash.hash();
    int j = x >>> (Integer.MAX_VALUE - b);
```

```
      x = (x << b)|(1 << (b-1)) + 1;
      int r = Integer.numberOfLeadingZeros(x)+1;
      if(M[j] == 0) {
        V--;
        M[j] = (byte)r;
        return true;
      } else if(r > M[j]) {
        Z -= Math.pow(2, -M[j]);
        Z += Math.pow(2, -r);
        M[j] = (byte)r;
        return true;
      }
      return false;
    } catch(IOException ex) {
      return false;
    }
  }
```

Finally, the implementation of the `size()` method uses the baseline value Z (what Flajolet calls the indicator function) to produce a corrected estimate of the cardinality:

```
  public int size() {
    double N = alpha/Z;
    if(N <= 5.0*m/2.0 && V > 0) {
      return (int) (m*Math.log((double)m/(double)V));
    } else if(N > Math.pow(2, 32)/30.0){
      return (int) (-Math.pow(2, 32)
      *Math.log(1.0 - N/Math.pow(2, 32)));
    } else
  return (int)N;
  }
```

Improvements to the HyperLogLog Algorithm

The HyperLogLog algorithm was originally designed for scenarios where the expected cardinality of the set to be measured is on the order of a billion. With this approximate cardinality in mind, HyperLogLog's creators introduced a correction to the raw estimate, N, in the implementation of the size calculation that is used as the raw estimate gets large. This correction comes into play to account for the increasing probability of hash collisions and fixes estimates for cardinalities in excess of 140 million or so.

For most users, this range of cardinality is more than sufficient. Applications operating on the scale of a single website will see cardinalities on the order of millions. Applications that deal with multiple websites, such as online advertising, can see cardinalities on the order of 100 million and perhaps 1 billion at the extremes.

Google, on the other hand, routinely deals with cardinalities well in excess of 1 billion. To deal with this, Google has introduced a number of modifications to HyperLogLog it calls HyperLogLog++ and presented in a paper in early 2013 (the paper also mentions that Google had previously been a heavy user of the Min-Count algorithm).

To use HyperLogLog with very large cardinalities, Google swaps the 32-bit hash function for a 64-bit hash function. It uses a proprietary hash function known only to Google, but there are 64-bit variants of the MurmurHash function used in this implementation.

This change has two effects. First, more space is required to minimally store the registers since 6 bits

will be required instead of the original 5. The implementation presented in the previous section is sub-optimal and uses 8 bits to represent the counter so the changes have no effect.

Second, the probability of hash collisions is much smaller than the original 32-bit hash function so the large range correction is no longer needed for practical cardinalities. If the cardinalities were to approach these levels, the authors suggest that better results are likely to be had by simply making the hash function larger. Doubling the size of the hash function only increases the storage space by 1 bit. The implementation given could use 256-bit hash functions without requiring any extra storage.

This still leaves the small cardinality corrections, which actually use an entirely different algorithm! To improve the bias when the cardinality is small, Google conducted an extensive empirical investigation for known cardinalities. This resulted in a set of interpolation points that can be used when cardinalities are small. The `size()` method then becomes:

```
public int size () {
  double N = alpha/Z;
if(N < 5.0*m) N -= estimateBias(N);
double H = (V > 0) ?
-Math.pow(2, 32)*Math.log(1.0 - N/Math.pow(2, 32)) : N;
  return (int)(H < threshold(b) ? H : N);
}
```

The two functions `threshold()` and `estimateBias()` are empirically determined by Google and available at http://goo.gl/iU8Ig.

Real-Time Unique Visitor Pivot Tables

In Chapter 6, “Storing Streaming Data,” a compressed bitmap was used to efficiently store the data to create pivot tables for the population of page views, clicks, and so on for a website. This is somewhat interesting, but it is often more interesting to talk about the audience of a website rather than its page views.

With compressed bitmaps, tracking unique users wasn't possible because adding new things is an append-only operation. Because of this, a monotonically increasing counter is required. In the original example, the order of arrival was chosen as this counter. In contrast, the unique visitors over some time period do not appear in any particular order and will likely even appear multiple times over that time period.

Instead, HyperLogLog sketches can be used in place of the compressed bitmap and the inclusion-exclusion principle used to estimate the intersections required by the pivot table. To begin, recall the input data for a page view on the website:

```
timestamp,user id,feature1:value,feature2:value,...,featureN:value
```

Each input record contains a user ID, which will be used as the element to enter into the HyperLogLog sketch. It also includes some number of feature elements that define demographic information about the user or other information about the page they are visiting (for example, the site section or the product page).

Essentially, HyperLogLog sketches for each of the feature:value combinations need to be maintained, containing the approximate number of unique user IDs associated with each. To build a pivot table for any two features A and B, simply retrieve the sketch for all the possible values of A: featureA:value1, featureA:value2, …, featureA:valueN and all of the sketches for the possible values of feature B: featureB:value1, featureB:value2,…,featureB:valueN. Then compute the intersection of each featureA:valueX, featureB:valueY combination using the inclusion-exclusion principle: |featureA:valueX| + |featureB:valueY| - |featureA:valueX union featureB:valueY|.

So long as the number of different values each feature can take on is small, this can be done interactively. If the possible values each feature can take on is very large, the best approach is to use the Count-Min sketch to identify the most common feature values to control the number presented in the pivot table user interface.

The Count-Min Sketch

The final sketch algorithm to discuss is the popular Count-Min sketch. This algorithm essentially approximates a map (also called a multi-set) where the distinct values act as the keys and the number of times an element has been seen is the value. The simplest implementations of the Count-Min sketch support approximate lookups in this map, called *point queries*. When the distinct values have some sort of intrinsic ordering, the sketch can be considered an approximation of a histogram (also known as a frequency table).

With a bit more work, the Count-Min sketch can also support range queries that approximate the frequency of elements in the range [a,b]. After range queries are supported, it is also possible to estimate order statistics using a binary search on the ranges. This is discussed in detail later in this chapter.

NOTE

Most of the sketches considered in this chapter only work when items are added to the sketch; they do not support removing items from the sketch. The authors of the original Count-Min paper call this a Cash Register model and it assumes that all frequencies are non-negative. The Count-Min sketch supports this model, but also supports what the authors call the Turnstile model, in which items can be subtracted from the set so values could potentially be negative. This chapter is primarily concerned with Cash Register models because they are more common in streaming data, but the Turnstile model is also of interest for the Count-Min sketch.

The implementation of the Count-Min sketch is quite similar to that of the Counting Bloom Filter. In fact, an extension of the Counting Bloom Filter called the Spectral Bloom Filter has been shown to be equivalent to the Count-Min sketch. Like the Counting Bloom Filter, the Count-Min sketch makes use of an array of counters for its registers. The primary difference is that each of the k hash functions receives its own set of m registers. This defines a two-dimensional matrix of register values where a hash function only updates a particular row. The value of k is commonly considered to be the "depth" of the sketch whereas the choice of m is considered to be the "width" of the sketch.

Adding an element to the sketch is very similar to the Counting Bloom Filter as well. The element is hashed by each hash function h(x), which is then mapped into m using the usual modulus to identify register j. For each of the hash functions, the algorithm increments the register at m_{ij}.

Point Queries

Finding the approximate frequency of an element is also very straightforward. After applying each of the hash functions, the algorithm computes $\min(m_{1j},\ldots,m_{kj})$ and considers this to be the approximate number of times this element has been added into the set. Although it's somewhat surprising that this works well enough to use in practice, it acts much like other sketches in that it relies on the fact that although the probability of a collision of one hash function might be too high, the probability that several hashes will collide is small enough to be useful.

Clearly, this approach tends to overestimate the approximate frequency of a value under the Cash Register model. Another element could (and will, when enough elements are entered) increment the same counter, much like the Bloom Filter. So, how badly does the Count-Min sketch overestimate this count?

In the original Count-Min paper, the authors show that the probability that the estimate of an element is between its true value x and an upper bound x + em, where m is the number of elements entered into a map, is larger than 1-d under two conditions. The first condition is that the width of the sketch is 2/e. The second condition is that the depth of the sketch is (log 1/d)/log 2. So, a Count-Min sketch where the estimate is within 5 percent of the sum with a 99 percent probability would have a width of 40 and a depth of 7. A depth of 8 with a width of 128 would have a relative error of approximately 1.5 percent with a probability of approximately 99.6 percent. Using 32-bit counters, the Count-Min sketch requires the same amount of space as a HyperLogLog sketch with b=12.

Count-Min Sketch Implementation

Rather than implementing a `Set<E>`, the Count-Min sketch implements the Java class `Map<E,Long>` albeit with some restrictions.

```
public class CountMinSketch<E extends Serializable>    implements Map<E,Long> {
  int width = 0;
  int depth = 0;
  byte[]  m;
  SerializableHasher[] hashes;
  ObjectOutputStream[] outputs;
  protected void initialize(int m,int[] seeds) throws IOException {
    hashes = new SerializableHasher[seeds.length];
    outputs= new ObjectOutputStream[seeds.length];
    for(int i=0;i<seeds.length;i++) {
      hashes[i]  = (new SerializableHasher()).seed(seeds[i]);
      outputs[i] = new ObjectOutputStream(hashes[i]);
    }
  }
  public CountMinSketch(int width,int[] seeds) throws IOException {
    this.width = width;
    this.depth = seeds.length;
    m = new byte[width*depth];
    initialize(m.length,seeds);
  }
```

The update step simply needs to increment the appropriate register locations and report the smallest to provide an estimate of the frequency. The following code implements this update with a specific increment amount, which is useful for merging two sketches. It also simplifies the lookup process as shown in the following code:

```
public Long increment(Object arg0,long amount) {
  int min = Integer.MAX_VALUE;
  for(int i=0;i<outputs.length;i++) {
    hashes[i].reset();
    try {
      outputs[i].writeObject(arg0);
      outputs[i].flush();
      int h = hashes[i].hash() % width;
      m[i*width + h] += amount;
      if(m[i*width + h] < min) min = m[i*width + h];
    } catch(IOException e) {
    }
  }
  return (long) min;
}
public Long increment(Object arg0) { return increment(arg0,1); }
public Long get(Object arg0) {
  return increment(arg0,0);
}
```

Top-K and "Heavy Hitters"

A common Count-Min application is maintaining lists of frequent items. The two basic forms of this list are the top-K list and the so-called Heavy Hitters list. The former is simply a list of the k most common items in the data stream, whereas the latter are the items with frequencies higher than some predetermined value f.

The basic implementation of either sort uses a Count-Min sketch to store the frequencies and a heap structure to hold the top values. In Java, a suitable heap can be implemented using a `PriorityQueue` with a suitable `Comparator` implementation, as in this example:

```
public class SketchComparator<T extends Serializable>    implements Comparator<T> {
  CountMinSketch<T> sketch;
```

```
    public SketchComparator(CountMinSketch<T> sketch) {
      this.sketch = sketch;
    }
    public int compare(T o1, T o2) {
      return (int)(sketch.get(o1) - sketch.get(o2));
    }
  }
```

To implement a top-K list, all you need is to increment an incoming element and then add it to the priority queue. If the queue is larger than k (it can be at most k+1 elements), remove the smallest element to return it to size k:

```
  public class TopList<E extends Serializable> {
    CountMinSketch<E> sketch;
    PriorityQueue<E>  heap;
    int               k       = Integer.MAX_VALUE;
    public TopList(int k,CountMinSketch<E> sketch) {
      this.sketch = sketch;
      this.k      = k;
      this.heap   = new PriorityQueue<E>(k+1,new SketchComparator<E>(sketch));
    }
    public void add(E element) {
      sketch.increment(element);
      heap.add(element);
      while(heap.size() > k) heap.remove();
    }
  }
```

The Heavy Hitter implementation is nearly identical, except that elements are removed from the queue if they do not meet the frequency requirements. This also requires the maintenance of a counter of all elements seen thus far.

```
  public class HeavyHitters<E extends Serializable> {
    CountMinSketch<E> sketch;
    PriorityQueue<E>  heap;
    double            f = 1.0;
    int               N = 0;
    public HeavyHitters(double f,CountMinSketch<E> sketch) {
      this.sketch = sketch;
      this.heap   = new PriorityQueue<E>(11,
        new SketchComparator<E>(sketch));
      this.f      = f;
    }
    public void add(E element) {
      N++;
      sketch.increment(element);
      heap.add(element);
      while(heap.size() > 0 &&
            f <= (double)sketch.get(heap.peek())/(double)N)
        heap.remove();
    }
  }
```

An alternative implementation would not trim the queue on each insert. Instead, that implementation could wait until the queue was queried for some reason.

Most Popular Website Items

A commerce website, such as Amazon.com or Etsy.com, usually has a most popular items sidebar. These websites often have thousands of products available in each department. To maintain a top-K list in real time, the sites would have to maintain the current number of purchases for each of those items in some accessible way. For Amazon.com, which has so much hardware that it started a successful side business renting it out, this may not be an issue, but it is very expensive to maintain for a relatively small feature.

Using the top-K list approach in the last section, it is possible to cheaply implement an approximate top-10 most popular item list using only a few kilobytes of RAM. To improve the accuracy, top-K lists could be maintained for every department on the website so that each page has a unique top-K list.

Range and Quantile Queries

Many times, the data being stored has some sort of natural ordering. When this happens it is natural to think of the data as having an empirical distribution (or histogram depending on the discipline), for example, the prices paid (in cents) for a particular advertising unit on a website or the number of seconds a user spent on a given page before taking an action.

In these cases, the questions of interest are usually concerned one way or another with a range query of some kind: What fraction of users spent less than 10 seconds on this page? What is the median sale price of this advertising unit? The naïve answer to the first question is to simply sum the point estimates from 0 to 10 seconds and report the total frequency. To answer the question of the median time, you would begin calculating the frequency of a range starting from $0.01 until the sum of the point estimates was approximately 50 percent of the total number of elements seen so far.

This naïve approach has two problems. The first is obvious in the second question: The number of operations required could be very large. If the median price is $15 then 1,500 point estimates are required, all using k hash calculations and so on. Less obvious is the second problem: These individual point estimates all have some error. Adding them together, as discussed in Chapter 9, means that their errors increase with each addition. In the best case, you'd expect the error of that median calculation to be 1,500 times larger than any single point estimate.

A less naïve solution is to find some way to reduce the total number of point queries required to compute the frequency of the range. One approach is to use multiple Count-Min sketches that store all the data at different resolutions. The first sketch is the basic sketch that stores the point queries. The second sketch halves the resolution by combining the first and second elements into one counter, and so on. The third sketch combines the first two elements of the second sketch so its first element is the count of the first, second, third, and fourth elements in the original sketch. The resolution is halved until the last sketch contains only two elements.

Doing this allows any range query to be divided into at most $2 \times \lg(N)$ intervals, where N is the total size of the domain. For example, if the domain is the space of 32-bit integers, then at most 64 subintervals are needed to compute any range query. Compared to the O(n) compute time of the naïve implementation, this is quite an improvement.

Dyadic Intervals

The intervals described in the last section are known as *dyadic intervals*. Dyadic intervals are

defined by two parameters: a level parameter and a start parameter:

```
public class DyadicInterval {
  public int level = 0;
  public int start = 0;
  public DyadicInterval() { }
  public DyadicInterval(int level,int start) {
    this.level = level;
    this.start = start;
  }
```

These two parameters define the start and end points of the interval as $[2^{level} \times start, 2^{level} \times (start+1) - 1]$:

```
public int min() { return (1 << level)*start; }
public int max() { return (1 << level)*(start+1) - 1; }
```

The subintervals can then be computed recursively by finding the largest dyadic interval that fits within the current interval. This interval is added to the final output and potentially generates two further intervals, the interval to the left of the range and the interval to the right at the range. These ranges are recursively divided in the same way until all points have been included in one range:

```
public static List<DyadicInterval> createIntervals(int a,int b) {
    ArrayList<DyadicInterval> output = new ArrayList<DyadicInterval>();
    Stack<Integer> left = new Stack<Integer>();
    Stack<Integer> right = new Stack<Integer>();
    left.push(Math.min(a, b));
    right.push(Math.max(a, b)+1);
    while(left.size() > 0) {
      int l = left.pop();
      int r = right.pop();
      for(int k = 32;k>=0;k--) {
        long J = (1L << k);
        long L = (int)(J*Math.ceil((double)l/(double)J));
        long R = L + J - 1;
        if(R < r) {
          //Segment fits in the range
          output.add(new DyadicInterval((int)k,(int)(L/J)));
          if(L > l) {
            left.push(l);right.push((int)L);
          }
          if(R+1 < r) {
            left.push((int)(R+1));right.push(r);
          }
          break;
        }
      }
    }
```

Implementing Count-Min Ranges

The implementation of a Count-Min sketch that can cover the space of 32-bit integers uses 32 different levels from 0 to 31, each with the same width and depth:

```
public class CountMinRange {
  ArrayList<CountMinSketch<Integer>> sketches =
    new ArrayList<CountMinSketch<Integer>>();
  long N = 0;
  int  min = Integer.MAX_VALUE;
  int  max = Integer.MIN_VALUE;
```

```
  public CountMinRange(int width,int[] seeds)
    throws IOException {
    for(int i=0;i<32;i++)
      sketches.add(
      new CountMinSketch<Integer>(width,seeds));
  }
```

The update step adds the count to the appropriate start parameter of each of the different levels:

```
public void increment(int value,long n) {
  if(value < min) min = value;
  if(value > max) max = value;
  for(int i=0;i<32;i++) {
    sketches.get(i).increment(value/(1 << i),n);
  }
  N += n;
}
```

Range calculations are a little bit more complicated. First, the appropriate set of dyadic intervals is computed and then the appropriate element from each of the levels is added to the total. Each level contributes at most two elements to each range query:

```
public long range(int a,int b) {
  long count = 0;
  for(DyadicInterval d : DyadicInterval.createIntervals(a, b)) {
    count += sketches.get(d.level).get(d.start);
  }
  return count;
}
```

The computation of quantiles, such as the median, is accomplished via a binary search of the space. It is unlikely that the exact probability is in the set, but the algorithm returns the closest value to the appropriate quantile.

```
public int quantile(double q){
  if(q == 0.0) return min;
  if(q == 1.0) return max;
  int lo = min;
  int hi = max;
  while(lo <= hi) {
    int mid = lo + (hi - lo)/2;
    double val = frequency(min,mid);
    if(val < q) hi = mid - 1;
    else if(val > q) lo = mid + 1;
    else return mid;
  }
  return lo; //Return the nearest value
}
```

Other Applications

This chapter provides an introduction to a class of dimension reduction tools with similar properties. Those properties are, essentially, storage and updates that do not directly depend on the number of elements to be considered. This allows for streaming applications to maintain control over processing time, which is the key to high-performance, low-latency applications.

Although the focus in this book is on streaming applications, these algorithms are also generally applicable. In particular, these data structures are often useful in Map-Reduce applications where the data coming into the map and reduce phases behaves very much like streaming data.

For example, the Bloom Filter is often used in filtering applications to do a rough filtering of data at the Map step before doing the final trimming in the Reduce step. One such application is the so-called "attribution" process. In this setting, users identified by a unique identifier engage in a number of events before possibly engaging in an event of interest, called a "conversion" in this context. The attribution process is interested in the events that happened in some window before the final conversion event. Because most users will not convert (low single-digit percentages of conversion are normal), a Bloom Filter containing the IDs of the users who did convert can be used in the Map step of an attribution Map-Reduce job. Even with a high error of 10 percent, it still tends to reduce the amount of data going to the reduce step by 80 percent to 90 percent.

Conclusion

One of the main challenges of processing streaming data is keeping up with the number of events to be processed. Even with the advent of the high-performance solid-state disk (SSD), this data must generally be stored in main memory (RAM) to achieve acceptable performance. If the data to be stored is simple, such as sums or averages, this does not present a problem.

When the data to be stored becomes more complicated, like the number of unique values in the stream, this can present a problem. Attempting to store the data directly can result in storage requirements that are proportional to the size of the data stream and can quickly overrun the available RAM.

This chapter has presented a number of methods for storing certain values such as sets and their size in such a way that the memory usage is controlled by the application rather than the data, ensuring that RAM requirements can be met. The downside of these techniques is that they introduce estimation error into computed values. In some cases, this error may not be tolerable, but the error is also a function of storage so it may be controlled by the application into acceptable levels.

Chapter 11
Beyond Aggregation

For many organizations, aggregation and visualization are the end of the road. Dashboards are created, aggregates are graphed, and reports are generated. To what end? The answer is usually so that "decision-makers" can take the "pulse" of whatever system is being monitored. Interpreting this statement a bit, it would seem to imply that the role of these systems is to surface information to a human decision-maker so that they can take some action that affects the system in a desirable way or react to an undesirable change in that system.

But, these systems are all operating in real time and humans are not real-time creatures. We eat, sleep, and generally do things other than stare at the continuously updating dashboard. How do we keep up? For mission-critical things, the solution has usually been to have a large number of humans working in shifts to keep an eye on the systems.

This works fairly well for relatively small systems, but even reasonably sized systems like power plants quickly reach the limits of feasibility for the "herd of humans" approach. This leads to the introduction of automated elements of the real-time system, such as alarms and automatic shutdown.

These automated systems are the focus of this chapter—helping the human decision-maker do their job by allowing them to focus on anomalous behaviors or automating the moment-to-moment decision-making process altogether. This turns out to be easier said than done, but a number of approaches have proven successful over time in at least limited capacities.

Automating either the decision process or the process of alerting an operator to anomalous data requires that the system have some sort of model that enables it to predict the behavior of the system. The first two sections of this chapter, "Models for Real-Time Data" and "Forecasting with Models", discuss the process of building models, using the concepts introduced in Chapter 9, "Approximating Streaming Data with Sampling." This processing of building a model is called "fitting" and is the essential task of statistical and machine learning applications. It introduces some of the methods behind classical modeling, which is usually done "offline" and then applied to the real-time data, as well as some approaches to fitting the model in an "online" fashion to the streaming data itself.

When it is possible to build models, those models need applications. The two most popular types of applications for real-time data are monitoring and optimization applications. It is no coincidence that they correspond to the idea of identifying anomalous behaviors in the data and making moment-to-moment decisions, respectively.

Monitoring involves the classical subject of real-time data. Collection of the data to be monitored and visualization of those results has been covered at length in previous chapters. What remains is the identification of anomalous events. The "Monitoring" section of this chapter discusses two types of anomaly detection. The first type is outlier detection, when the system enters an anomalous but ultimately transient state. The second type is change detection where the system enters a fundamentally different state of operation.

The last section of this chapter covers a hot topic in the Internet world, optimization, where it is fairly easy to make changes and monitor outcomes. Website optimization, in particular, is a popular subject with any number of methods available for so-called A/B testing. In fact, most website traffic monitoring software seems to have some sort of A/B testing framework built-in. In this chapter, a

specific technique called the multi-armed bandit is used along with the modeling approaches from the first section to implement optimization in a real-time environment.

Models for Real-Time Data

Anything that hopes to predict the behavior of a system must have an underlying model that describes it. Ideally, this description is compact relative to the data it describes. For example, in Newtonian physics, a simple set of equations describing the actions of forces on a collection of objects is sufficient to predict their motion over long periods of time.

With this concept in mind, a model can be broken into two parts. The first is the behavior of the underlying system, which describes how various components of the world interact to result in the observed behavior. Unfortunately, it is very rare that all the variables that make up this model can be observed, so it is usually not possible to completely determine this model. It is, however, possible to construct a model that considers the variables that can be observed. The second part of the model is the noise introduced by the variables that cannot be observed.

This applies to both the underlying process and to the mechanisms of measure. Any tool used to measure data can also be modeled in the same way as the underlying process, and it can have large effects on the outcome. For example, if the same data is being collected by several different systems, knowing the bias and variance of each collection system allows them to be normalized.

Toward this end, this section introduces some of the methods used to identify the underlying model in the presence of noise. As this book is focused on real-time data, the first part of this section focuses on some of the simpler methods for modeling time-series models. (Real-time data is inherently time-series data due to the nature of data collection.) These methods are widely used in forecasting procedures in all different fields.

The second part of this section discusses linear models, also known as regression models. These models are a classic technique in statistics and are probably the most-used modeling technique in the world. They describe linear relationships between variables being measured and some outcome, which is also being measured. One variant, logistic regression, is a popular method for producing models for the odds that an event will occur.

When the data is highly nonlinear or its structure is poorly understood, the Artificial Neural Network has become a popular option for modeling time-series data. The final discussion in this section is about the basic neural network model. Neural network models are easily adapted to real-time problems as the training and prediction cycles are identical.

Simple Time-Series Models

Any real-time data system is essentially made up of a series of sequential values. As discussed in Chapter 8, these values are computed for a particular time "bucket" or quantization, and then they are analyzed. Most of these measurements are inherently noisy, so time-series models attempt to get at the underlying shape of the time series by smoothing over the noise, usually through some sort of averaging technique, several of which are presented in this section.

Moving Average

The simplest averaging approach for a time series is the *moving average*. As the name implies, this is the average of the values over a "sliding window" of values. It is usually expressed in terms of the number of time buckets being used in the average and is often written as `MA(k)`. For instance, moving averages for stock market data are often expressed in terms of days.

The moving average is easy to implement as it only requires the storage of k different values. For efficiency, it is usually best to keep track of the running sum, adding and removing values as appropriate. Otherwise, every update of the moving average would require k operations instead of, at most, two. This is shown in the following implementation:

```
public class MovingAverage {
  LinkedList<Double> values = new LinkedList<Double>();
  double sum;
  int    k;
  public MovingAverage(int k) {
    this.k = k;
  }
  public double observe(double value) {
    sum += value;
    values.add(value);
    if(values.size() > k) {
      sum -= values.removeFirst();
      return sum/(double)k;
    }
    return Double.NaN;
  }
}
```

Notice that the moving average returns `NaN` if it has not received enough observations. To produce a usable observation, the moving average loses the first `k-1` values of the time series. Selecting the appropriate window size for a moving average is a bit of an art. Moving averages act as a low-pass filter on the actual data, and choosing too small a range does not smooth out enough of the noise. Conversely, too large a range smoothes out important signals in the data. Figure 11.1 shows the effect of making different choices about moving average windows on some simulated sine waves with a small amount of added noise. In the case of Figure 11.1, a window of 30, perhaps 35 would best recover the underlying signal.

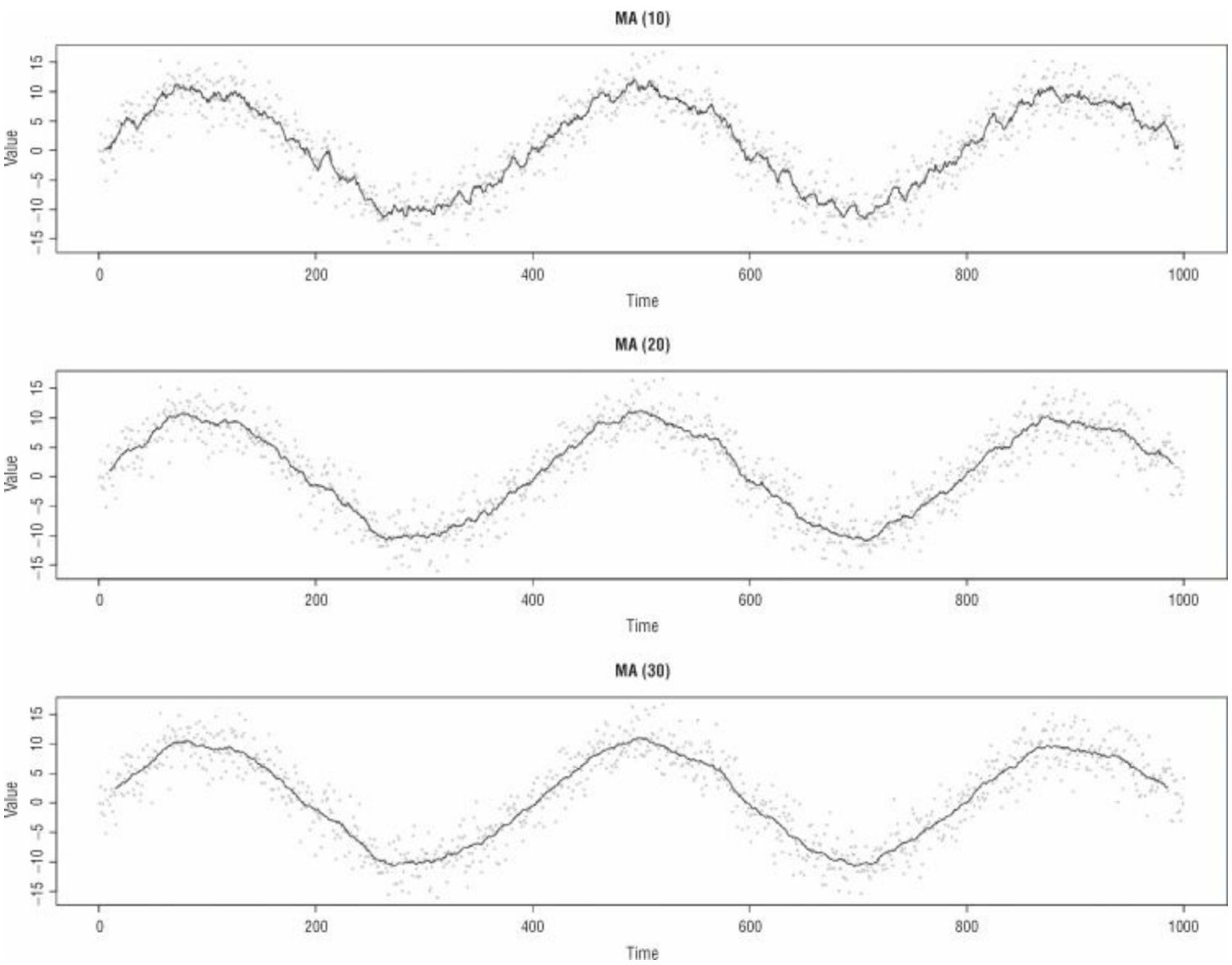


Figure 11.1

Weighted Moving Average

The weighted moving average is a generalization of the standard moving average that uses different weights for each of the elements in the window. This collection of weights is known as the *kernel*; various kernels are standard in different industries. The normal moving average is simply a weighted moving average with a kernel that assigns `1/k` to all values. Implementing a weighted moving average is a little more complex:

```
public class WeightedMovingAverage {
  double[] kernel;
  double[] values;
  double   kernelSum = 0.0;
  int      k = 0;
  long     N = 0;
  public WeightedMovingAverage(double[] kernel) {
    this.kernel = kernel;
    for(double j : kernel) kernelSum += j;
    values = new double[kernel.length];
  }
  public double observe(double x) {
    values[k++] = x;
    if(k == values.length) k = 0;
```

```
      N++;
      if(N < kernel.length) return Double.NaN;
      double y = 0;
      for(int i=0;i<kernel.length;i++)
      y += kernel[i]*values[(k+i) % values.length];
      return y/kernelSum;
    }
  }
```

The important thing to notice with the weighted moving average is that the computation is much slower than the normal moving average, requiring k operations for every observed value. For most offline analyses, this is not a concern, but it can be a problem with a real-time environment's online processing.

Exponential Moving Average

Many times, the kernel in a weighted moving average is used to place more weight on recent observations than on older observations. The exponential moving average also does this by taking the weighted average of the current moving average value and the new observation:

EMA = a*X + (1-a)*EMA

The exponential moving average is not quite the same as the weighted moving average, and it has several advantages. The first is that it only requires a single operation to obtain a new value for the exponential moving average instead of k operations. The second is that it only requires the storage of a single value instead of the k values needed by both the moving average and the weighted moving average. As expected, the implementation is very simple:

```
  public class ExponentialMovingAverage {
    double value = 0.0;
    double alpha;
    public ExponentialMovingAverage(double alpha) {
      this.alpha = Math.min(alpha, 1.0);
    }
    public double observe(double x) {
      value = alpha*x + (1-alpha)*value;
      return value;
    }
  }
```

Like the normal moving average, selecting a value for `alpha` is more of an art than a science. The most common rule of thumb is to set alpha to be `2/(k+1)` where `k` is the number of values used in the moving average. The result is roughly 86.5 percent of the weight to be derived from the most recent k values and, as shown in Figure 11.2, will very closely match the standard moving average. Figure 11.2 also shows the same dataset smoothed by two other exponential moving averages with k values of 20 and 30, respectively. The original moving average is also shown as the dotted line for comparison. Notice how close the two forms of moving average track each other. Because the values are close, the exponential moving average is the algorithm most commonly used in practice.

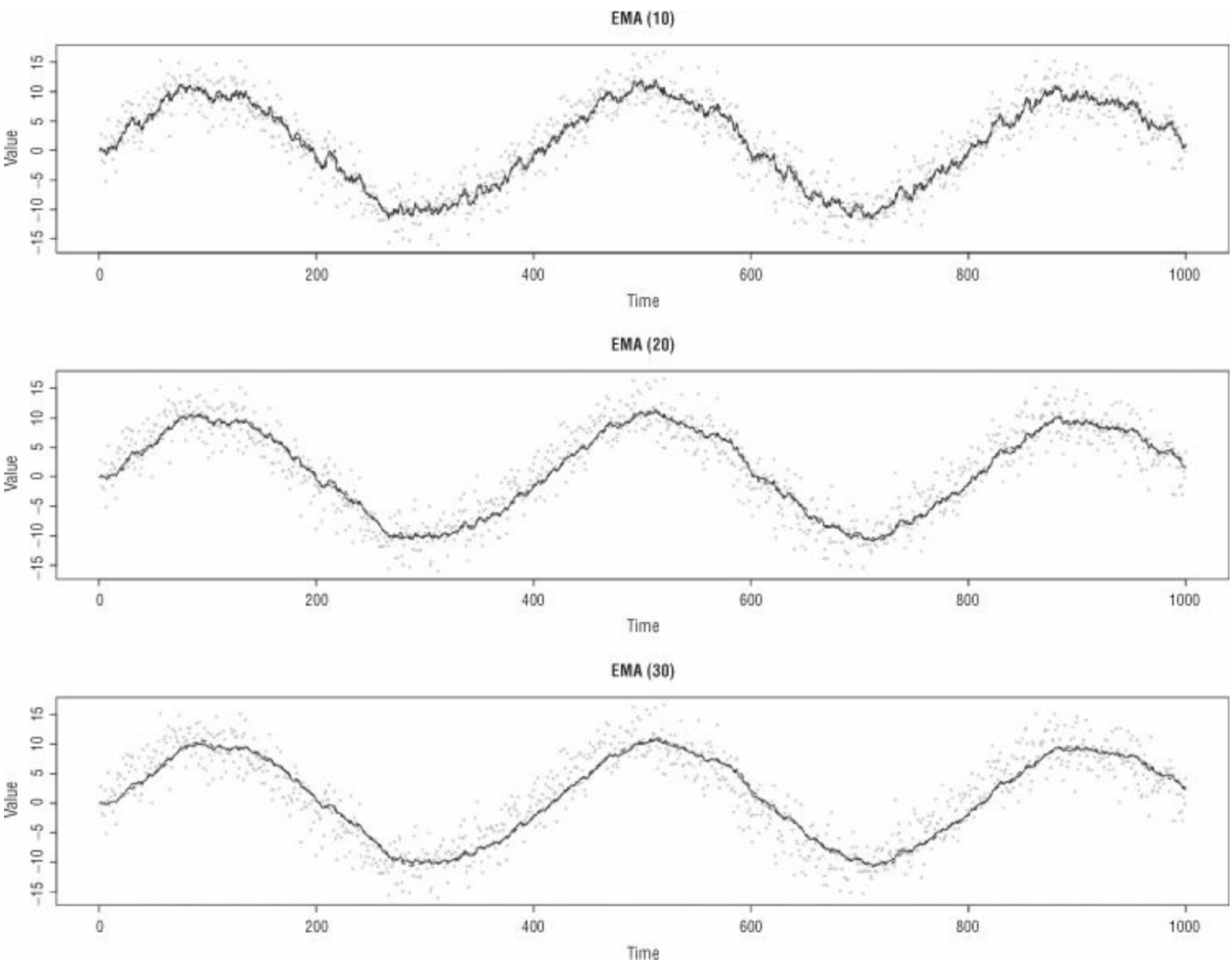


Figure 11.2

Linear Models

Linear models are the most popular form of statistical modeling in many fields, from economics to biology and beyond. (One of its first applications was in the study of genetics.) This type of model is also known as a regression model, the term used in this book, as well as a least squares model, which refers to the technique used to find the coefficients of the model.

The essential idea is that an outcome variable, denoted as , has a linear relationship with an array of explanatory variables, denoted as *x[0]*, *x[1]* and so on, after the different elements of the array are multiplied by a coefficient array, denoted as `B[0]`, `B[1]`, and so on:

```
public class LinearModel {
  double[] B;
  public LinearModel(double[] B) {
    this.B = B;
  }
  public double y(double[] x) {
    double y = 0;
    for(int i=0;i<B.length;i++) y += B[i]*x[i];
    return y;
  }
}
```

"Fitting" these models means finding appropriate values of B given that the values of x are observed along with y, which can be interpreted as a noisy version of . In the original formulation this noise is normally distributed (normal distributions are discussed in Chapter 9) with a mean of zero and a variance of s. This is done by minimizing the square of the difference between the observed values of y and the values of given x returned by the `y` method of the `LinearModel` class.

In other words, the goal is to find an array for B that returns the least sum of squares or "least squares error," given in the following function:

```
public double error(double[] y,double[][] x) {
  double error = 0.0;
  for(int i=0;i<y.length;i++) {
    double diff = y[i] - y(x[i]);
    error += diff*diff;
  }
  return error;
}
```

This error is also known as the residual sum of squares (RSS) and is often used to determine how well a model fits the data. Inspecting the individual elements of the error (usually without the square) is used to determine whether or not the model is well specified. Trends or a periodic signal in the data is often a sign that the model is missing a term.

Simple Linear Regression

In the case that the x array is only ever two values, with the first value being the constant 1 (known as the intercept term), there is a simple closed form solution for the two values of the B array. When this happens, the first value of B is usually called a, and the second value called b and the equation for reduced to the following simple form:

```
public class SimpleLinearModel {
  public double a,b;
  public SimpleLinearModel(double a,double b) {
    this.a = a;
    this.b = b;
  }
  public double y(double x) {
    return a + b*x;
  }
}
```

The error to minimize is similarly simplified:

```
public double error(double[] y,double [] x) {
  double error = 0.0;
  for(int i=0;i<y.length;i++) error += (y[i]-a-b*x[i])*(y[i]-a-b*x[i]);
  return error;
}
```

After a little bit of calculus, it is found that this function is minimized when b is equal to the covariance of x and y divided by the variance of x. The value of a best minimizes the error when it is equal to the mean of y minus the estimated value of b multiplied by the mean of x. If all the data are present in an array, the following function will find the appropriate values of a and b:

```
public void fit(double[] y,double[] x) {
  double sumX = 0.0, sumY = 0.0;
  double sumXY = 0.0, sumX2 = 0.0;
```

```
    for(int i=0;i<y.length;i++) {
      sumX += x[i];
      sumY += y[i];
      sumXY += x[i]*y[i];
      sumX2 += x[i]*x[i];
    }
    double n = (double)y.length;
    b = (sumXY - (sumX*sumY)/n)/(sumX2 - (sumX*sumX)/n);
    a = sumY/n - (b*sumX)/n;
  }
```

Notice that the preceding function only requires a single pass over the data. This means that it is also amenable to a simple streaming formulation, useful for real-time analysis:

```
  public class StreamingSimpleLinearModel {
    double sumX = 0.0, sumY = 0.0;
    double sumXY = 0.0, sumX2 = 0.0;
    double n = 0.0;
    boolean dirty = true;
    double  a = 0.0,b = 0.0;
    private void update() {
      if(!dirty) return;
      b = (sumXY - (sumX*sumY)/n)/(sumX2 - (sumX*sumX)/n);
      a = sumY/n - b*sumX/n;
      dirty = false;
    }
    public void observe(double y,double x) {
      sumX += x;sumY += y;
      sumXY += x*y;sumX2 += x*x;
      n += 1.0;
      dirty = true;
    }
    public double b() { update();return b; }
    public double a() { update();return a; }
  }
```

Multivariate Linear Regression

Computing the best estimates for B when there is more than one x variable, excluding the intercept term, is a bit more complicated, but it's still straightforward. Using a technique called *ordinary least squares*, which is sometimes used as a synonym for multivariate linear regression, the values of B have a closed form so long as certain requirements are met.

The primary requirements are that there is no correlation between the different `x` variables and that the standard deviation of the `y` term does not depend on the value of `x` (only the mean of `y` varies with `x`). The first requirement is usually fairly easy to achieve by checking the correlations between the various `x` variables under consideration and dropping one of the two correlated variables from the equation. Another approach is to transform the matrix of `x` values using an orthogonal transformation, such as principal components analysis. This produces `x` values that are, by definition, uncorrelated. The second requirement is usually assumed more than it is ensured, but it is easy to check by inspecting the difference between the observed `y` values and their predicted values after fitting the model.

Assuming these conditions are met and the values of `x` are placed into a matrix `X` with `k` columns and `n` rows, where `k` is the number of different variables and `n` is the number of observations, the following expression finds a `B` vector that minimizes the mean square error:

$B = (X^T X)^{-1} X^T y$

This form is the solution to a linear system of equations called the normal equations. It is possible to solve these directly using linear algebra libraries such as the Apache Commons Math library. However, the direct computation can have problems with numerical stability, so most implementations use other techniques. There are a variety of options, but one of the most common is the use of the QR factorization. What QR factorization says is that any matrix A can be represented by the production of two matrices Q and R where Q is an orthonormal matrix (like those discussed earlier), and R is an upper triangular matrix (meaning roughly half of its values are zeroes). An orthonormal matrix is special because $Q^T Q = I$, where I is known as the identity matrix. The identity matrix is a matrix of all zeroes, except its diagonal, which is filled with ones. By replacing `X` with its QR factorization, the equation for `B` is then:

$B = R^{-1} Q^T y$

This form is much more numerically stable, although it does require the computation of the QR factorization. Rather than attempting to implement this, there are many different libraries that can be used. For example, in Java there is the Apache Common Math library, available through Maven:

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-math3</artifactId>
  <version>3.2</version>
</dependency>
```

Using the Apache Commons Math library's QR factorization classes, it is easy to simply write down the final equation in `LinearModel`'s fit implementation:

```
public void fit(double[] y,double[][] x) {
  RealMatrix X = new Array2DRowRealMatrix(x);
  RealVector Y = new ArrayRealVector(y);
  B = (new QRDecomposition(X)).getSolver().solve(Y).toArray();
}
```

Of course, the Apache Commons Math library also includes a class that implements ordinary least squares, so it is possible to simply use that instead:

```
public void fitOLS(double[] y,double[][] x) {
  OLSMultipleLinearRegression ols = new OLSMultipleLinearRegression();
  ols.newSampleData(y, x);
  B = ols.estimateRegressionParameters();
}
```

For use in streaming data environments, Apache Commons also includes an “online” version of the ordinary least squares solver. Called the Miller Updating Regression after its inventor, it relies on the fact that the QR factorization can be updated with new data. This allows observations to be streamed into the model with the ability to produce regression coefficients *B* after a sufficient number of observations (at least twice the number of coefficients in the model) have been made. To use it, simply initialize the model with the number of coefficients in the model:

```
int k = 14;
MillerUpdatingRegression rm = new MillerUpdatingRegression(k,true);
```

When new data arrives, the `addObservation` method takes an array *x* of size *k* and an observed output *y*:

```
rm.addObservation(x, y);
```

To retrieve the current fitted model, the `regress` method is used. This returns a `RegressionResults` class like the `OLDMultipleLinearRegression` class that can be used to retrieve parameters estimates and other values of interest:

```
double B[] = rm.regress().getParameterEstimates();
```

Logistic Regression

It is also possible to fit linear models that do not assume normally distributed observations using a generalized linear model (GLM). In these models, the observed values y are assumed to come from a distribution in the exponential family of distributions. This includes many of the named distributions mentioned in Chapter 9. For example, Poisson regression is used to model count data.

One of the most popular GLMs is the logistic regression model, which is used to model the Bernoulli distribution. It is used to model the probability of an event occurring or of an observation being the member of a class. There are also extensions to the model for multiclass modeling in which the outcome is a multinomial distribution rather than the Bernoulli distribution. Like other linear models, the probability is calculated by taking a weighted linear combination of the input `x` values. However, rather than using this directly, that linear combination is transformed to produce a value between 0 and 1, as in the following modification to the `LinearModel` class:

```
public double y(double[] x) {
  double y = 0;
  for(int i=0;i<B.length;i++) y += B[i]*x[i];
  return logit(y);
}
```

where the `logit` method has the following implementation:

```
public static double logit(double y) {
  return 1.0/(1.0 + Math.exp(-y));
}
```

This is easy enough to implement when the values of `B` are known, but finding appropriate values of `B` given observations of the input and output is much more difficult than the multivariate linear regression model. The most common approaches rely on so-called quasi-Newton techniques that are best left to others to implement. Although a number of high-quality numerical packages for C/C++ and FORTRAN exist for solving logistic regression, Java implementations are much more rare.

Fitting Regression with Logistic Regression

If using an interface to one of the high-quality native approaches is not an option, it is also possible to use a gradient descent approach to find the appropriate values of B. This approach iterates over the data multiple times, adjusting the values of each B by an amount proportional to the sum of the errors between the predicted values and the actual y values for each observation. This amount is controlled by a parameter `alpha`, which is known as the "learning rate." This parameter controls how quickly the algorithm converges on a final answer and is usually set to something like 0.1 as a starting point. It must always be smaller than 1 and larger than 0.

```
public LogisticRegression fit(double[][] x,double[] y) {
  //Initialize the weights
  B = new double[x[0].length];
  double lastError = Double.POSITIVE_INFINITY;
```

```
    for(int iter=0;iter<MAX_ITER;iter++) {
      double err2 = 0;
      for(int i=0;i<x.length;i++) {
        double t = y[i] - y(x[i]);
        for(int j=0;j<x[i].length;i++)
          B[j] += alpha*t*x[i][j];
        err2 += t*t;
      }
      err2 = Math.sqrt(err2);
      if(err2 - lastError < 1e-6) break;
      lastError = err2;
    }
    return this;
  }
```

The iteration continues until either a maximum number of iterations are reached or the change in error becomes so small it is not worth continuing. A more general application of this approach is used to fit the parameters of the artificial neural network models introduced in the next section.

Tip

The gradient descent approach works for all types of regression, not just logistic regression. Although there are other options for multivariate regression, this allows for streaming implementations of the more complicated forms of regression. When working with streaming data it is assumed that data in the future will be equivalent to data in the past, which eliminates the need to iterate over the data multiple times.

Neural Network Models

Neural network models, also known as the artificial neural network (ANN), are a collection of nonlinear models used to predict some outcome given some set of input variables. The first examples of neural networks date to the early 1940s where they were used to solve classification problems. It is generally accepted that neural networks represent a type of nonlinear regression and can be applicable to the same problem domains.

Inspired by the activity of physical neurons in the brain, the artificial neuron, called a unit in neural network jargon, computes a function of the weighted sum of its inputs. This function is called the activation function and is usually chosen from the family of sigmoid functions, which range from functions like the logistic function described earlier to the Heaviside step function. Figure 11.3 shows three popular choices of activation function in neural networks: the logistic function, the hyperbolic tangent, and the Heaviside step function.

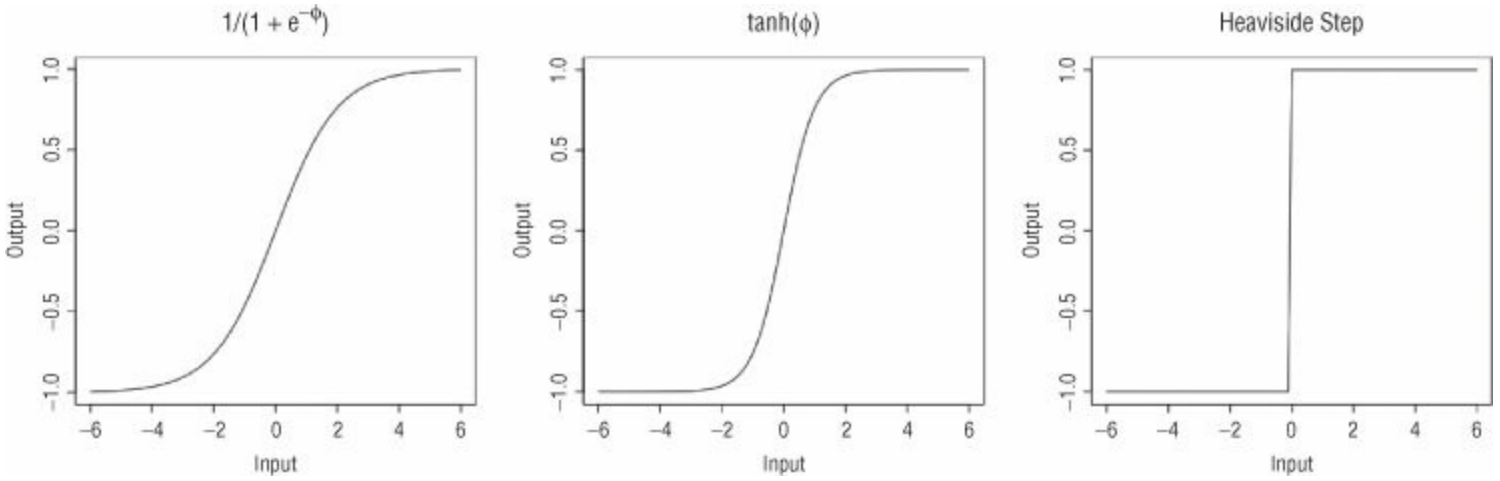


Figure 11.3

The units are then arranged into layers. There is an input layer, an output layer, and some number of hidden layers. The input layer is used to drive the activation function of units in the hidden layer (if any). The first hidden layer's units then drive the activation function of the next layer and so on, until the last hidden layer drives the activation function of the output layer. There is often a bias unit—which plays the same role as the intercept in linear models—that provides a constant input to each neuron.

A typical neural network arrangement is shown in Figure 11.4. In this case, the input layer and the hidden layer are the same size with the output layer only containing two units. In general, there is no restriction on the number of units in a layer relative to any other layer. Because each unit is connected to all the units of the previous layer, all of the units will in theory learn different aspects of the input signal.

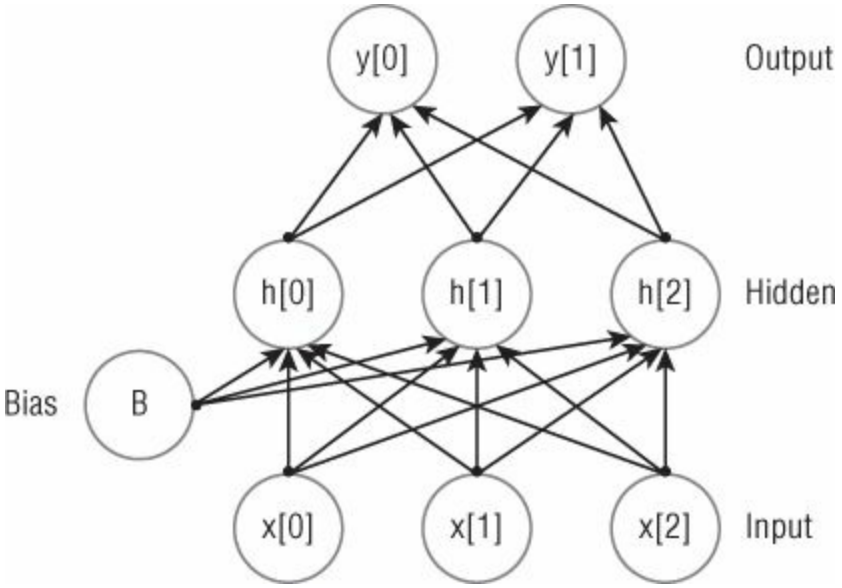


Figure 11.4

Multi-Layer Feed-Forward Network Implementation

The type of neural network described earlier is known as a feed-forward network. This is because inputs arrive at the input layer and then information is fed forward through the network to the output layer.

To implement this model, first define a `Layer` class. This class holds the current values for the layer as well as the weight matrix that is used to produce the weighted sum of values. The size of the weight matrix is defined by the number of units and the number of inputs to the layer. There is also an array of bias weights included in the layer implementation:

```
public class Layer {
  double[]   v;
  double[][] w;
  double[]   bW = null;
  Activation fn;
  public Layer(int units,int inputs,Activation fn,boolean bias) {
    this.fn = fn;
    v = new double[units];
    w = new double[units][];
    for(int i=0;i<v.length;i++) w[i] = new double[inputs];
    if(bias)
      bW = new double[units];
  }
```

For simplicity, this implementation uses the same activation function for all units. It is not required that all units in a layer use the same function, but it is often the case. However, any function used must have a derivative to be used in the backpropagation training algorithm discussed in the next section. The abstract `Activation` class defines methods for both the function and its derivative:

```
public abstract class Activation {
  public abstract double f(double x);
  public abstract double df(double fx);
}
```

Two commonly used activation functions are the logistic function (also called the sigmoid function) and the hyperbolic tangent. The logistic function is implemented with a constant `k` that is used to control the rate of change from positive to negative. A sufficiently large value for `k` of, say, 100, is

usually sufficient to approximate the Heaviside step function, which is not ordinarily usable in the backpropagation algorithm:

```
public static Activation logit(final double k) {
  return new Activation() {
    @Override
    public double f(double x) {
      return 1.0/(1.0 + Math.exp(-k*x));
    }
    @Override
    public double df(double fx) {
      return k*fx*(1.0-fx);
    }
  };
}
public static Activation sigmoid(double k) {
  return logit(k);
}
public static Activation logit() {
  return logit(1.0);
}
public static Activation sigmoid() {
  return logit();
}
```

The hyperbolic tangent activation function is another popular choice for activation function. It is also included as a standard option in this implementation:

```
public static Activation tanh() {
  return new Activation() {
    @Override
    public double f(double x) {
      return Math.tanh(x);
    }
    @Override
    public double df(double fx) {
  return 1 - fx*fx;
    }
  };
}
```

NOTE

The derivatives used in the `Activation` class implementations in the code are not quite the derivatives of `f(x)`. In this case, the derivatives of `f(x)` can be expressed in terms `f(x)`, in this case as `1-f(x)×f(x)`. To make the calculation of the derivative more efficient, the value of `f(x)` is assigned to `fx` and then used in the "derivative" function in place of `x` itself. This representation of the derivatives of activation functions is widely used in the neural network literature. Unfortunately, the fact that the derivative is stated in terms of the original function is not widely discussed in the literature. As a result, there are often errors in examples of backpropagation found online. This version of the derivative is used because the feed-forward portion of the neural network has already computed `f(x)` for each unit in the network. Because the value must be stored to compute the error at each layer, it is convenient to use the derivative stated in terms of `f(x)`. Otherwise, the original value of `x` would also need to be stored.

When data is fed to the layer, it must compute all of the weighted sums and apply the activation function. The implementation here is used for clarity rather than performance; consequently, it is fairly inefficient, requiring `O(n×m)` operations:

```
public double[] feed(double[] x) {
  for(int i=0;i<v.length;i++) {
    double[] W = w[i];
    v[i] = bW != null ? bW[i] : 0.0;
    for(int j=0;j<W.length;j++) v[i] += W[j]*x[j];
    v[i] = fn.f(v[i]);
  }
  return v;
}
```

Because the bias input is always 1, the output value for each unit can be initialized to its bias weight. Otherwise, it is simply set to zero. After the values have been updated, the function returns the new state so that it may be passed to the next level (or used as the output).

These layers are then assembled into a neural network with the `NeuralNetwork` class. This class maintains an array of layers, which includes, at the minimum, an output layer. Most networks also contain a hidden layer, and some contain several hidden layers. It is not necessary to define a discrete input layer, just the number of units in the layer, because its activation function is the identity function:

NOTE

"Deep learning" refers to neural networks with more than one hidden layer. It may also refer to the practice of stacking neural networks together.

```
public class NeuralNetwork {
  int inputUnits = 0;
  ArrayList<Layer> layers = new ArrayList<Layer>();
  public NeuralNetwork inputs(int inputUnits) {
    this.inputUnits = inputUnits;
    return this;
  }
```

To define each subsequent layer, the `NeuralNetwork` class inspects the previous layer (or the number of inputs) to determine the size of the weight matrix to be used:

```
public NeuralNetwork layer(int units,Activation fn,boolean bias) {
  int inputs = (layers.size() == 0) ?
    this.inputUnits : layers.get(layers.size()-1).units();
  layers.add(new Layer(units,inputs,fn,bias));
  return this;
}
public NeuralNetwork layer(int units,Activation fn) {
  return layer(units,fn,true);
}
public NeuralNetwork layer(int units) {
  return layer(units,Activation.tanh());
}
```

To make a prediction, the `feed` method iteratively applies each layer to the output of the previous layer or the input itself:

```
public double[] feed(double[] x) {
  for(Layer l : layers)
    x = l.feed(x);
  return x;
}
```

Training with Backpropagation

Of course, a newly constructed feed-forward network is useless because it has not been "trained" on any data. In fact, as currently implemented, the output vector is always zeroes because none of the connections in the network have any weight.

The most popular method for training a multilayer feed-forward neural network is the *backpropagation algorithm*. This algorithm is a relative of the generalized least squares algorithm used to fit the regression models earlier in the chapter. It works by minimizing the sum-of-squares error between the target output values and the output values observed after feeding the input values through the network. This is recorded in an error array that is added to the `Layer` implementation. For efficiency, the `Layer` implementation also maintains the total error for this layer:

```
double[]   err;
double     E;
public double[] errors() { return err; }
```

This array is initialized in the constructor to have the same size as the value array:

```
public Layer(int units,int inputs,Activation fn,boolean bias) {
```

```
    this.fn = fn;
    v   = new double[units];
    w = new double[units][];
    for(int i=0;i<v.length;i++) w[i] = new double[inputs];
    if(bias)
      bW = new double[units];
    err = new double[units];
  }
```

This error array is updated during the training phase by propagating the error from the output layer backward to the input layer. This backward propagation gives the algorithm its name. It first computes the error for each unit in the layer as the difference between the expected output and the activation of each unit multiplied by the derivative of the activation function. For the output layer, the difference, which is passed in the *s* array, is the difference between the training value and the output of the `feed` method in the `backprop` method of the `Layer` class:

```
  public double[] backprop(double[] s) {
    double[] out = new double[w[0].length];
    E = 0;
    for(int i=0;i<v.length;i++) {
      err[i] = fn.df(v[i])*s[i];
      E += err[i]*err[i];
```

To produce a difference array to propagate to the next `Layer`, the current `Layer` computes a weighted sum of its own error values. As shown in the following implementation, this essentially reverses the process of the feed-forward network. This array is then returned so that it may be used as the input to the `backprop` method of the next `Layer`:

```
      double[] W = w[i];
      for(int j=0;j<W.length;j++) out[j] += W[j]*err[i];
    }
    return out;
  }
```

When the errors have been propagated to each of the layers, the weights themselves are updated. Each weight is updated according to the delta rule, which states that the change in the weight between two units is proportional to the production of the current unit error, the derivative of the unit's activation function, and the activation value of the input unit. The error array already stores the production of the unit error and its derivative, so it only needs to be multiplied by the previous layer's activation value, which is passed into the following implementation as the array *o*:

```
  public double[] update(double[] o,double r) {
    for(int i=0;i<v.length;i++) {
      if(bW != null)
        bW[i] += r*err[i];
      double[] W = w[i];
      for(int j=0;j<W.length;j++)
        W[j] += r*err[i]*o[j];
    }
    return v;
  }
```

The other value passed to the update method, *r*, is a “learning rate” similar to the one used in the `LogisticRegression` example. This keeps the weight adjustment from moving too quickly, which helps the stability of the gradient descent method. Typical values for most networks are between 0.2 and 0.8, and it usually requires some trial and error to find the best rate.

Finally, before learning can begin there should be some initialization of weights. By default, all of the weights in a network start with a value of zero, but this can lead to the network being trapped in a local minimum and unable to get to the "best" network that fits the data. Usually, it is best to randomize the weights before training as shown in the following code, which is added to the `Layer` implementation:

```
public void initialize(Random rng) {
  for(int i=0;i<v.length;i++) {
    for(int j=0;j<w[i].length;j++)
      w[i][j] = 2*rng.nextDouble() - 1;
      bW[i] = 2*rng.nextDouble() - 1;
  }
}
public void initialize() {
  initialize(new Random());
}
```

The pieces of the backpropagation algorithm are finally assembled in the `train` method of the `NeuralNetwork` class in this example. The training example—consisting of an input array, x, and an output array, y—is then fed through the network to produce an output:

```
public NeuralNetwork train(double[] x,double[] y) {
  double[] out = feed(x);
```

The difference between the output and the input is then calculated and propagated backward through the network:

```
double[] s = new double[out.length];
for(int i=0;i<y.length;i++) s[i] = y[i] - out[i];
for(int i=layers.size()-1;i>=0;i--) s = layers.get(i).backprop(s);
```

Finally, each layer's weights are updated by walking forward through each layer:

```
  for(Layer l : layers) x = l.update(x, learningRate);
  return this;
}
```

Learning an Exclusive-Or Pattern

One of the interesting examples of neural network learning used to motivate the need for hidden layers is the ability to learn the exclusive-or (XOR) pattern. In these examples, the training set has two inputs and a single output. If any of the two inputs is active, the output is also active. If neither or both of the inputs are active, the output is inactive.

As it happens, many learning algorithms, including a neural network without hidden layers, cannot learn this pattern. The reason is that many classification techniques require that the classes be linearly separable to solve the problem, and the XOR pattern is not linearly separable.

Adding a hidden layer to the neural network allows it, essentially, to further subdivide the input space and learn a network that can reproduce the exclusive-or pattern from inputs.

This example uses two neural networks—one with a hidden layer and one without—that are both trying to learn an XOR pattern. The networks, `nn` and `bad` respectively, are easy to define in the framework:

```
NeuralNetwork nn = NeuralNetwork.build().inputs(2).layer(3).layer(1);
NeuralNetwork bad= NeuralNetwork.build().inputs(2).layer(1);
nn.initialize();
bad.initialize();
```

Both networks are then trained for 1,000 iterations on all possible XOR inputs and outputs and the error recorded:

```
for(int i=0;i<1000;i++) {
  double err = 0.0;
  double errBad = 0.0;
  for(Obs x : xorData) {
    nn.train(x.x,x.y);
    bad.train(x.x,x.y);
    err += nn.error();
    errBad += bad.error();
  }
  System.out.println(err+"\t"+errBad);
}
```

After this iteration it is clear that the neural network with a hidden layer has achieved a much better overall error than the network without the hidden layer, which has become trapped at a relatively high error, as shown in Figure 11.5. The solid line shows the error of a three-layer neural network learning an XOR pattern. The dotted line shows the two-layer neural network failing to learn the same pattern.

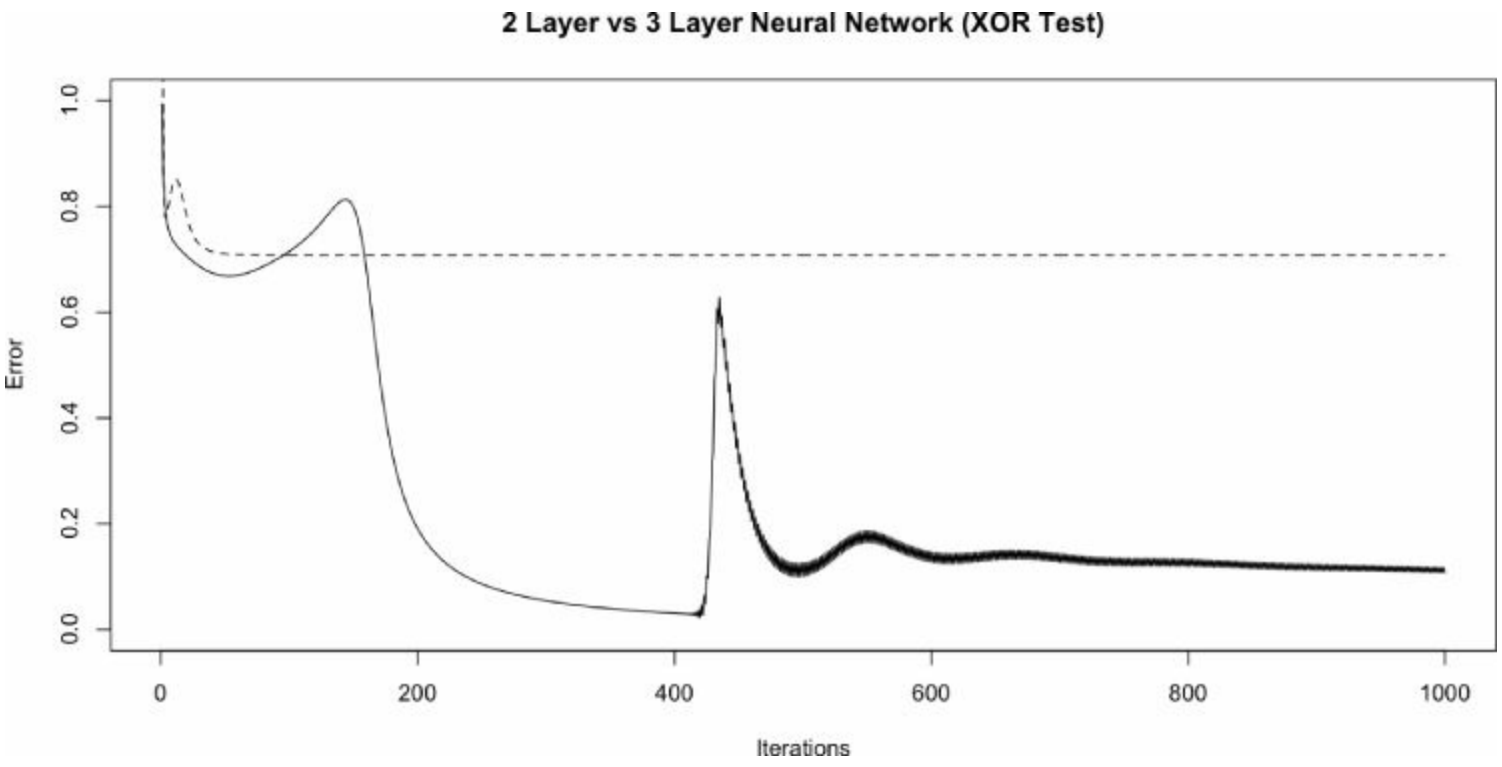


Figure 11.5

Momentum Backpropagation

This version of the algorithm uses the simplest method for computing the change in weight at each step. A common performance improvement is to introduce the notion of "momentum" into the computation of the change in weight.

Rather than using the basic implementation for the computation of the change in weight, the Layer implementation stores the weight delta values using a separate matrix:

```
double dW[][];
double dbW[];
double m = 0.1;
public MomentumLayer(int units, int inputs,
  Activation fn, boolean bias) {
  super(units, inputs, fn, bias);
  dW = new double[units][];
  for(int i=0;i<v.length;i++) dW[i] = new double[inputs];
  if(bias)
    dbW = new double[units];
}
```

The update step then uses a weighted average of the new delta for the weight and the previous delta for the weight with the weight being chosen by a constant value `m`:

```
@Override
public double[] update(double[] o, double r) {
  for(int i=0;i<v.length;i++) {
    if(bW != null) {
      dbW[i] = (1-m)*r*err[i] + m*dbW[i];
      bW[i] += dbW[i];
    }
    double[] W = w[i];
    for(int j=0;j<W.length;j++) {
      dW[i][j] = (1-m)*r*err[i]*o[j] + m*dW[i][j];
```

```
        W[j]     += dW[i][j];
      }
    }
    return v;
  }
```

This allows weight changes in the same direction to proceed more rapidly, while sudden changes in the direction of the weight have less of an effect. In the small examples used in this chapter, using the momentum modification has relatively little effect. However, if the network being trained is slow to converge without momentum, adding the momentum term can improve performance.

Forecasting with Models

Forecasting is widely used in real-time data analysis both on its own to predict future values (for example to allow for automated capacity planning) and to form the basis of comparison with the observed data used by the monitoring and optimization application areas discussed later in this chapter. The most naïve methods of forecasting are essentially trivial, simply using the historical average or perhaps the moving average as the forecast for future values. For very simple systems, this is sufficient, but most of the time these naïve methods will not capture sufficient information about the underlying system to produce good forecasts.

The topic of forecasting is large, and many books have been written about it over the years. This chapter covers a few techniques that have gained widespread popularity due to their simplicity, their effectiveness, or both.

Exponential Smoothing Methods

Exponential smoothing methods of forecasting are fairly widespread thanks to their relative ease of implementation, ability to handle the seasonality present in many time series, and general good performance when the signal is not too noisy. The simplest form of these forecasts, when the data has no trend or seasonal components, is to simply use the exponential moving average of the current time period as the forecast for the next time period.

Of course, most time series have either a trend or seasonality and often have both. When this is the case, it is possible to build a variety of different models with different types of trend and seasonality components. Of all these possibilities, the most well-known models are the Holt-Winters models. Dating back to the late 1950s, these models assume an additive trend with either additive or multiplicative seasonal components.

Holt's method is used to compute the portion of the model ascribed to the additive trend portion of the model. (The seasonal component is discussed later in this section.) To compute the trend components of the model, the forecast equation is decomposed into two parts: a level and a trend. In forecasts, where the x value is considered to be the number of time steps into the future, the `level` takes on the role of an intercept parameter, and the `trend` takes the role of a slope parameter in the simple linear regression. The basic forecast is then fairly simple:

```
public class HoltForecast {
  double level = 0;
  double trend = 0;
  public HoltForecast(double level,double trend) {
    this.level = level;
    this.trend = trend;
  }
  public double forecast(double x) {
    return level + trend*x;
  }
}
```

To update the forecast when new data arrives, the level and trend are maintained as separate exponentially smoothed variables. The smoothing parameter a is used for both the trend and the level component, and the smoothing parameter b is used only for the trend component. Each component is adjusted by the error in the forecast and the observed value, much like the gradient descent algorithms used by neural networks. This is equivalent to exponential smoothing and is known as the error

correcting form; it's shown here:

```
double a = 0.0;
double b = 0.0;
public HoltForecast(double level,double trend,double a,double b) {
  this.level = level;
  this.trend = trend;
  this.a     = a;
  this.b     = b;
}
public HoltForecast(double level,double trend) {
  this(level,trend,0.2,0.8);
}
public double error(double y) { return y - forecast(1.0); }
public double update(double y) {
  double e = error(y);
  level = level + trend + a*e;
  trend = trend + a*b*e;
  return e;
}
```

The seasonal component of this model was later added in what has become known as the Holt-Winters forecast. There are two types of seasonal models used in Holt-Winters: additive and multiplicative. Both require two extra parameters: a seasonal period `s` and a smoothing parameter `g`. In this example, the length of the period is passed in with the initial estimates of the seasonal contribution to the forecast:

```
public class SeasonalForecast extends HoltForecast {
  double[] s;
  double   g = 0.0;
  long     t = 0;
  public SeasonalForecast(double level, double trend,
    double a, double b,double[] s,double g) {
    super(level, trend, a, b);
    this.s = s;
    this.g = g;
    this.t = 0;
  }
}
```

The period of the seasonality in this method must be selected ahead of time, although it is usually fairly intuitive. For many of the classical applications of this method, the period is either 4 or 12 depending on whether the data are quarterly or monthly outputs. For real-time applications, the period is often something like hourly or daily, leading to periods of 24 or 7, respectively.

For the additive seasonal forecast, the seasonal estimate for each time period in the future is added to the output of the Holt forecast:

```
public class AdditiveForecast extends SeasonalForecast {
  public AdditiveForecast(double level, double trend,
      double a, double b,double[] s, double g) {
    super(level, trend, a, b, s, g);
  }
  @Override
  public double forecast(double x) {
    int h = (t + (int)Math.floor(x-1.0)) % s.length;
    return super.forecast(x) + s[h];
  }
```

When initializing the seasonal component, the sum of all the components should be approximately equal to zero.

For the update step, the error remains the same difference between the observed value and the forecasted value. The update of the seasonal value only depends on the error and the values of a and g, so it can be updated independently of the level and trend values:

```
@Override
public double update(double y) {
  double e = super.update(y);
  s[t] = s[t] + g*(1-a)*e;
  t = (t + 1) % s.length;
  return e;
}
```

The multiplicative forecast is somewhat more complicated. The forecast step multiplies the seasonal value by the output of the Holt forecast:

```
@Override
public double forecast(double x) {
  int h = (t + (int)Math.floor(x-1.0)) % s.length;
  return super.forecast(x)*s[h];
}
```

Using this forecast, using the normal error between the forecasted value and the observed value would require that the error term for the level and trend updates be divided by the seasonal value used for the forecast. To allow the use of the original Holt forecast update, the error term is stated with this division already in place:

```
@Override
public double error(double y) {
  return super.error(y)/s[t];
}
```

This is then corrected in the computation of the seasonal update. The seasonal update is divided by the sum of the level and trend terms in the same way as the level and trend updates were divided by the seasonal update:

```
@Override
public double update(double y) {
  double z = level() + trend();
  double e = super.update(y)*s[t];
  s[t] = s[t] + (1-a)*g*e/z;
  t = (t + 1) % s.length;
  return e;
}
```

Although the initial seasonal components of the additive forecast should roughly cancel each other out, the seasonal components of the multiplicative forecast should sum to roughly the length of the period. Both the models tend to work fairly well, but it is often the case that the multiplicative model fits real-world data a bit better than its additive counterpart.

Regression Methods

The same ideas as the exponential smoothing approach can also be used with a regression model. Although the choice of variables for regression is virtually infinite, assuming enough data has been

collected to support the chosen variables, one place to start is simply the k most recent observations.

```
public void streamingRegressionTest() {
  int k = 14;
  double[] x = new double[k];
  double   y;
  double   t = 0.0;
  MersenneTwisterFast twist = new MersenneTwisterFast();
  MillerUpdatingRegression rm = new MillerUpdatingRegression(k,true);
  for(int i=0;i<1000;i++) {
    double base = Math.sin(t);
    y = base + 0.5*twist.nextGaussian();
    //Update the input time series
    if(i >= x.length) {
      rm.addObservation(x, y);
      if(rm.getN() > 2*x.length) {
        double B[] = rm.regress().getParameterEstimates();
        double y2 = B[0];
        for(int j=0;j<x.length;j++)
          if(!Double.isNaN(B[j+1])) y2 += B[j+1]*x[j];
        for(int j=0;i<x.length-1;j++) x[i] = x[i+1];
        x[x.length-1]=y;
      }
    } else {
      x[i] = y;
    }
    t += 2.0*Math.PI/60.0;
  }
}
```

This example uses the Miller method, which is a technique for updating the QR factorization. It would also be possible to use the gradient descent technique discussed earlier, especially if you're using logistic regression instead of linear regression. However, this version works passably well, as shown in Figure 11.6. That said, neural networks often perform better at this task, as shown in the next section. In Figure 11.6 the original signal is identified by a dotted line; the observed data is represented by the gray dots; and the predicted values are identified by the solid line.

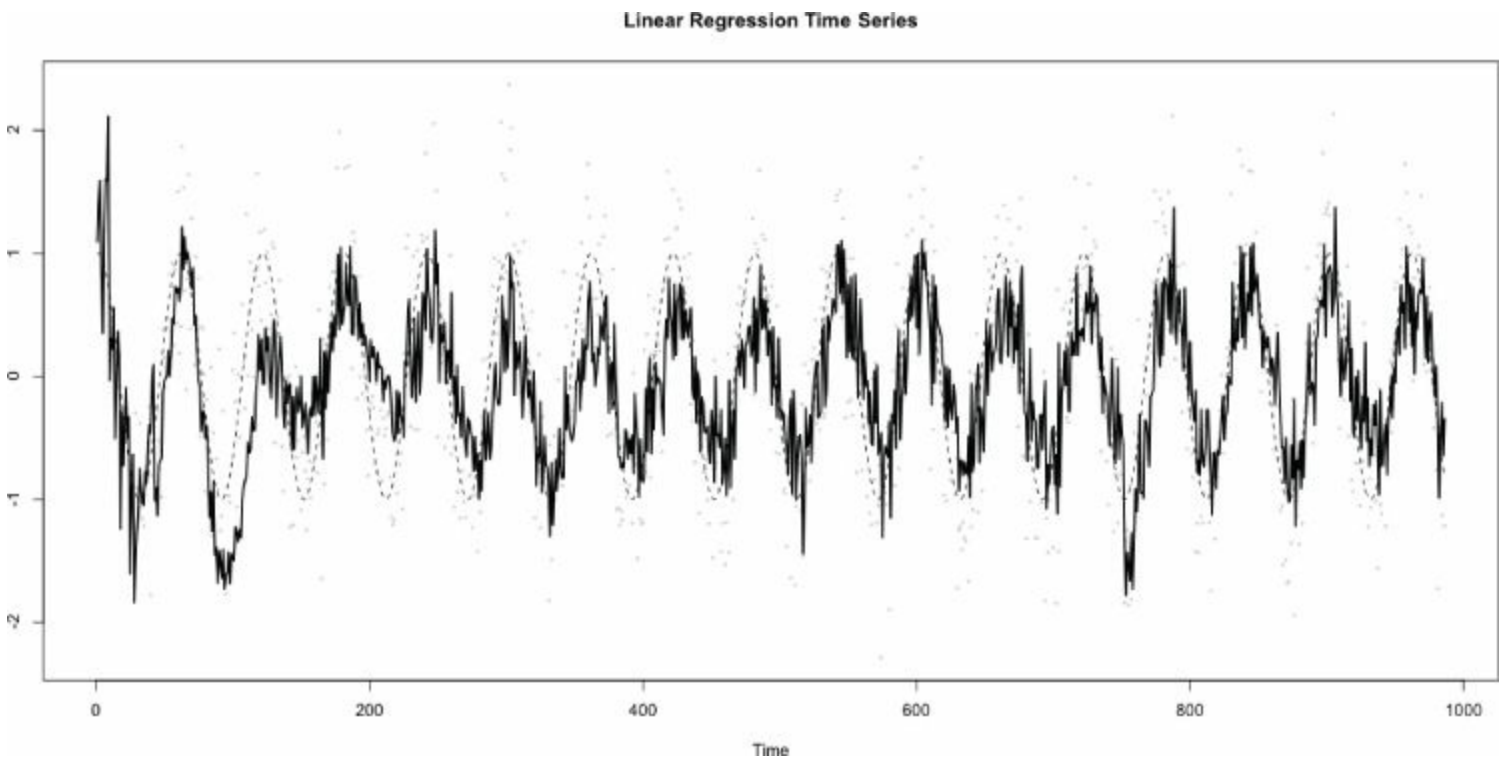


Figure 11.6

Neural Network Methods

Like the regression method, the neural network can be used to forecast time-series data. As with the regression method, the inputs are lagged observed values, and the output is a prediction of the next `n` values. For example, the following test code uses a simple neural network that uses the previous *k* observed values as its input. The output is a prediction for the next point in the time series.

The neural network itself is initialized with `k` input units, a hidden layer containing an arbitrarily chosen five units and a single output layer:

```
int k = 14;
double[] x = new double[k];
double[] y = new double[1];
double   t = 0.0;
MersenneTwisterFast twist = new MersenneTwisterFast();
NeuralNetwork       nn    = NeuralNetwork.build()
  .inputs(k)
  .layer(5)
  .layer(1)
.initialize();
```

In this example, simulated data is generated from a sine wave and then noise is added to the signal:

```
for(int i=0;i<1000;i++) {
  double base = Math.sin(t);
  y[0] = base + 0.5*twist.nextGaussian();
  //Update the input time series
  if(i >= x.length) {
    nn.train(x, y);
    System.out.println(nn.output()[0]+"\t"+y[0]+"\t"+base);
    //Move the time series over
    for(int j=0;i<x.length-1;j++) x[i] = x[i+1];
    x[x.length-1]=y[0];
  } else {
    x[i] = y[0];
```

```
    }
    t += 2.0*Math.PI/60.0;
  }
```

The first k steps of the process are used to fill the input vector. After the input vector has been filled, the update algorithm is run every time a new input arrives. In this particular implementation of a feed-forward neural network, training the model on an observation also generates a prediction for the outputs prior to the update of the model.

The neural network in this example is being updated in an online fashion. A particular data point is only used once in the model, and all data is eventually used to improve the model. This example produces the predicted value, base value without noise, and the noisy value used as the input to the model. Figure 11.7 shows these three values against each other, where the neural network manages to recover the original signal fairly well. The original signal is represented by a dotted line; the gray dots indicate the actual observed values; and the solid line indicates the prediction for each observed value made by the neural network.

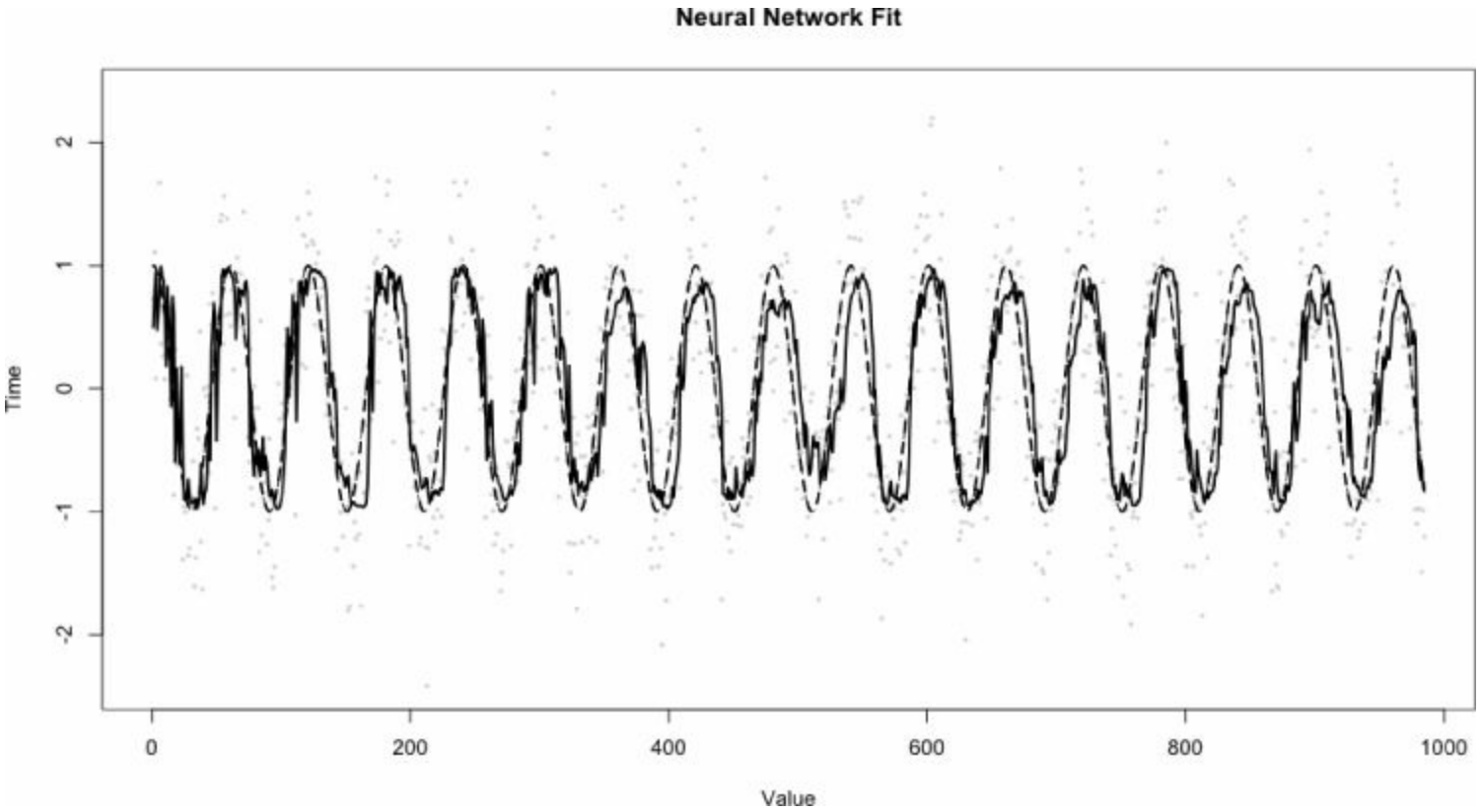


Figure 11.7

Monitoring

Systems monitoring is the classic application for real-time data collection and analysis. This usually involves collecting the number of events over a period of time quantized to a particular frequency.

For example, a website might collect the number of each type of HTTP responses per minute. Changes in the frequency of error (5xx) responses usually warrant some type of response, although they also occur due to transitory events such as extreme load spikes. A change in the frequency of correct (2xx) responses might also warrant a response. For a news site, a breaking story that "goes viral" could have any number of possible responses.

There are also the classic system-monitoring applications that track aspects of the machines running the website. In this case, different gauges are tracked: CPU load, fan speed, disk and network I/O. Changes in these values are often indicative of external changes or failing hardware. Other more fanciful applications might include large-scale medical monitoring for hospitals or traffic monitoring for freeways.

In any case, irrespective of the application area, the goal of monitoring systems is simple: Identify changes to the system, notify operators of the change, and minimize the occurrences of false alarms. This is more generally known as anomaly detection.

Outlier Detection

The simplest form of anomaly detection is outlier detection. Outliers are observations that fall far away from their expected values, but in such a way that the process producing the data remains unchanged. In a fixed dataset, it is usually fairly easy to identify the outliers with methods ranging from visual inspection to goodness-of-fit methods. These methods usually do not translate well to the online setting, where the most common approach is to use a threshold mechanism to identify outliers.

The online versions of these techniques rely on the deviation of the value from its predicted value. The primary difference between them is the metric they use to decide that a value is truly an outlier. Regardless, they rely on the implementation of a forecaster, which could be simply defined by this interface:

```
public interface Forecaster {
  public double forecast(double y);
}
```

Any forecasting mechanism discussed in this chapter can be used to support outlier detection. Building on the previous section, this forecaster is implemented using the neural network time-series forecaster with a hidden layer:

```
public class NeuralNetworkForecaster implements Forecaster {
  NeuralNetwork nn;
  int           n;
  double[]      x;
  double[]      Y = new double[1];
  public NeuralNetworkForecaster(NeuralNetwork nn) {
    this.nn = nn;
  }
  public NeuralNetworkForecaster(int n) {
    this(NeuralNetwork.build().inputs(n).layer(n).layer(1));
    this.n = n;
    this.x = new double[n];
```

```
    }
    public NeuralNetworkForecaster() {
      this(20);
    }
    public double forecast(double y) {
      if(n > 0) {
        x[x.length - n] = y;
        y = n < x.length ? x[(x.length - n) - 1] : y;
        n--;
        return y;
      } else {
        Y[0]=y;
        y=nn.train(x, Y).output()[0];
        for(int i=0;i<x.length-1;i++)
        x[i]=x[i+1];
        x[x.length-1] = Y[0];
        return y;
      }
    }
  }
```

This is then used as the basis for an outlier detector. The simplest form of detector simply tracks values that are more than some number of standard deviations—usually 3—from the forecasted value based on the errors of the last *n* observations, which are stored in the outlier detector:

```
  public class ThresholdDetector {
    Forecaster f;
    int        n;
    double     sigma;
    LinkedList<Double> values = new LinkedList<Double>();
    double     s2;
    public ThresholdDetector(Forecaster f,int n,double sigma) {
      this.f = f;
      this.n     = n;
      this.sigma = sigma;
    }
    public ThresholdDetector(Forecaster f,int n) {
      this(f,n,3);
    }
    public ThresholdDetector(Forecaster f) {
      this(f,30);
    }
```

When a new observation arrives, it is run through the forecaster to determine the error. If the error exceeds the number of standard deviations specified by sigma, it is considered an outlier. In this case, it is not used to update the standard deviation of the errors to avoid skewing the data. Otherwise, the standard deviation calculation is updated to reflect a non-outlier value:

```
    public boolean observe(double y) {
      double err = y - f.forecast(y);
      double sig = Math.sqrt(s2/((double)n-1.0));
      //If this is an outlier don't include it in s2
      if(Math.abs(err)/sig > sigma)
        return true;
      //Otherwise update our standard deviation
      s2 += err*err;
      if(values.size() == n)
        s2 -= values.removeFirst();
      values.add(err*err);
```

```
      return false;
    }
```

There are other approaches, but they mostly employ this basic framework for their updates. For example, rather than using the standard deviation, many outlier detectors declare an outlier as being outside 1.5 or 3 times the interquartile range. This was originally used to identify outliers in boxplot visualizations and has since been repurposed for outlier detection. This is further generalized by scan statistic approaches, which use the percentiles of the error to determine whether the process is in an outlier state.

Change Detection

Outlier detection as discussed in the previous section assumes that the observed outlier is a true anomaly and does not represent a fundamental change in the process generating the time series. These changes are much longer lived and are usually fit in a retrospective manner using fairly sophisticated approaches, such as clustering or Hidden Markov Models. There are fewer online approaches to change detection, and those that exist generally depend on the maintenance of at least two forecasts over the data. These various forecasts' errors are compared to determine if the underlying process has undergone a shift.

An example of this approach can be found in the finance community in the form of the Moving Average Convergence Divergence (MACD) indicator. Introduced in the 1970s, this indicator uses three different exponential moving averages to indicate changes in trends. Originally this approach was applied to detecting changes in the trend of a stock's price, but the approach is very similar to more modern approaches to change detection.

In the standard formulation, the closing prices for a stock are used to compute an exponential moving average (EMA) with a period of 12 days, a period of 26 days, and a period of 9 days. The difference between the 12-day EMA and the 26-day EMA is known as the MACD. The 9-day EMA is known as the signal line. When the MACD crosses from below the signal line to above the signal line, the stock is interpreted to have shifted to a positive trend. When the opposite happens, the stock is interpreted to have shifted from a positive trend to a negative trend. Similarly, the MACD line moving from negative to positive and from positive to negative has the same interpretation, but it is generally considered to be weaker evidence than the crossing of the signal line.

There are many variations on this approach that use shorter and longer EMAs depending on the application, but it allows for an easily implemented method of detecting changes in trend. The primary problem is that it is prone to false signaling of changes in the trend. An extension would do the same thing, but it would track trends in the error between a forecaster and the actual value. This can be useful when there is known seasonality in the data that should be removed before tracking the trend. Using the Holt-Winters forecaster, one approach would track changes in the trend component of the model.

Real-Time Optimization

If a metric is being monitored for changes, presumably there is some action that can be taken to mitigate undesirable changes in the metric or emphasize positive changes in the metric. Otherwise, outside of its entertainment value, there is little use in tracking the metric.

The more interesting situations that arise are when the metric is an outcome that can be affected by a process that can be automatically controlled. For example, an e-commerce website might have two different designs that are hypothesized to have different returns (either the number of sales or the dollars per session for each design). Testing which design performs best is known as A/B testing and is often performed sequentially by launching a new site and tracking the average session value compared to the old site. A more sophisticated approach, often used by large consumer websites, is to perform the experiment by exposing some fraction of users to the new design and tracking the same value.

However, if the size of the exposed populations for each design can be controlled, then it is possible to automatically choose the best design over time without explicit control. One way of doing this is using the so-called multi-armed bandit optimization strategy.

The premise of this strategy is that a player has entered a casino full of slot machines (also known as one-armed bandits). Each machine has a different rate of payout, but the only way to determine these rates is to spend money and play the machine. The challenge for the player is then to maximize their return on the investment (or if it is a real casino, minimize its loss). Intuitively, the player would begin by assuming that all machines are equal and playing all of them equally. If some machines have higher payouts than the rest, the player would then begin to focus more of their attention on that subset of machines, eventually abandoning all other machines.

In the context of website optimization, the two different designs are the one-armed bandits, and "playing" the game assigns a visitor to one of the two designs when they arrive at the site. The only problem is to decide which design a visitor should see.

For the purposes of demonstration, assume that a visitor either buys a product or not, and all of the products have the same price. In this case, it is only necessary to model the probability of a purchase rather than modeling the values directly. As each user arrives, the exposed design is tracked, as is the fact that the users have purchased something using the following class. The exposure is incremented using the `expose` method, and the purchase is tracked using the `success` method as shown here:

```
public class BernoulliThompson {
  double[] n;
  double[] x;
  public BernoulliThompson(int classes) {
    n = new double[classes];
    x = new double[classes];
  }
  public void expose(int k) {
    if(k < n.length)
      n[k] += 1.0;
  }
  public void success(int k) {
    if(k < x.length) x[k] += 1.0;
  }
```

Recall the discussion of the beta distribution in Chapter 9 as a model for the probability of a

Bernoulli trial. When the experiment begins, it is unknown what the conversion rates will be, so perhaps it is best to assign the rates for the different classes a uniform distribution, which can be modeled with a `Beta(1,1)` distribution. After some visits have been observed, the Beta distribution for each class can be updated to be a `Beta(1+x,1+n-x)` distribution. This is known as a posterior distribution and is used to select the design for a new visitor. Next a conversion rate is drawn from the posterior distribution of each design. The design with the largest conversion rate is then assigned to this visitor as follows:

```
Distribution d = new Distribution();
public int choose() {
  int    max = -1;
  double mu = Double.NEGATIVE_INFINITY;
  for(int i=0;i<x.length;i++) {
    double alpha = 1.0 + x[i];
    double beta  = 1.0 + (n[i] - x[i]);
    double x = d.nextBeta(alpha,beta);
    if(x > mu) {
      max = i;
      mu  = x;
    }
  }
  return max;
}
```

This technique is known as Thompson Sampling, and it can be extended to any distribution. For example, using the average value rather than the conversion rate might be modeled by using an exponential distribution instead of the beta distribution. The sampling procedure is still the same; it simply uses the largest payout sampled from the distribution rather than the largest conversion value. For the most part, these values are easily updated and allow the optimization of the website design to be conducted in real time.

After using a basic estimate, the next logical step is to use a model rather than simply using the empirical value for each design. It might be the case that different populations of users respond differently to each design. In that case, being able to predict an average value or a conversion rate using one of the models discussed earlier in this chapter enables the optimizer to choose the best design for each user to maximize the return for the site.

Conclusion

The techniques in this chapter are by no means the only approaches to solving problems such as forecasting, anomaly detection, and optimization. Entire books could and have been written on both the general topics and the specific techniques covered in this chapter. Additionally, the techniques presented here remain active areas of academic and industrial research; new refinements and approaches are being developed all the time. The goal for this chapter was to provide a brief introduction into the approaches to give practitioners some grounding in their use and a basis of comparison for other techniques.

With that, it is also important to remember that even the simplest techniques can yield good results. It is often better to use the simplest method that could possibly work across a number of different problems before returning to the original problem to further optimize the approach. In many cases, going from 50 percent of optimal to 80 percent of optimal is achievable by relatively simple approaches, whereas it takes a very sophisticated approach to get from 80 percent of optimal to 90 percent of optimal. For example, the famous Netflix prize offered $1 million to the team that could produce a 10 percent improvement in its recommendation engine. Although a team accomplished this feat, the full algorithm was never implemented because the cost of implementation outweighed the gains from further improvement of the algorithm.

Real-Time Analytics

Techniques to Analyze and Visualize Streaming Data

Byron Ellis

WILEY

Real-Time Analytics: Techniques to Analyze and Visualize Streaming Data

Published by

John Wiley & Sons, Inc.

10475 Crosspoint Boulevard

Indianapolis, IN 46256

www.wiley.com

Published simultaneously in Canada

ISBN: 978-1-118-83791-7

ISBN: 978-1-118-83793-1 (ebk)

ISBN: 978-1-118-83802-0 (ebk)

Library of Congress Control Number: 2014935749

As always, for Natasha.

Credits

1. Executive **Editor**
2. Robert Elliott
3.
4. Project **Editor**
5. Kelly Talbot
6.
7. Technical **Editors**
8. Luke Hornof
9. Ben Peirce
10. Jose Quinteiro
11.
12. Production **Editors**
13. Christine Mugnolo
14. Daniel Scribner
15.
16. Copy **Editor**
17. Charlotte Kugen
18.
19. Manager **of Content Development and Assembly**
20. Mary Beth Wakefield
21.
22. Director **of Community Marketing**
23. David Mayhew
24.
25. Marketing **Manager**
26. Carrie Sherrill
27.
28. Business **Manager**
29. Amy Knies
30.
31. Vice **President and Executive Group Publisher**
32. Richard Swadley
33.
34. Associate **Publisher**
35. Jim Minatel
36.
37. Project **Coordinator, Cover**
38. Todd Klemme
39.
40. Proofreader
41. Nancy Carrasco
42.
43. Indexer

44. John Sleeva
45.
46. Cover **Designer**
47. Wiley

About the Author

Byron Ellis is the CTO of Spongecell, an advertising technology firm based in New York, with offices in San Francisco, Chicago, and London. He is responsible for research and development as well as the maintenance of Spongecell's computing infrastructure. Prior to joining Spongecell, he was Chief Data Scientist for Liveperson, a leading provider of online engagement technology. He also held a variety of positions at adBrite, Inc, one of the world's largest advertising exchanges at the time. Additionally, he has a PhD in Statistics from Harvard where he studied methods for learning the structure of networks from experimental data obtained from high throughput biology experiments.

About the Technical Editors

With 20 years of technology experience, **Jose Quinteiro** has been an integral part of the design and development of a significant number of end-user, enterprise, and Web software systems and applications. He has extensive experience with the full stack of Web technologies, including both front-end and back-end design and implementation. Jose earned a B.S. in Chemistry from The College of William & Mary.

Luke Hornof has a Ph.D. in Computer Science and has been part of several successful high-tech startups. His research in programming languages has resulted in more than a dozen peer-reviewed publications. He has also developed commercial software for the microprocessor, advertising, and music industries. His current interests include using analytics to improve web and mobile applications.

Ben Peirce manages research and infrastructure at Spongecell, an advertising technology company. Prior to joining Spongecell, he worked in a variety of roles in healthcare technology startups, he and co-founded SET Media, an ad tech company focusing on video. He holds a PhD from Harvard University's School of Engineering and Applied Sciences, where he studied control systems and robotics.

Acknowledgments

Before writing a book, whenever I would see “there are too many people to thank” in the acknowledgements section it would seem cliché. It turns out that it is not so much cliché as a simple statement of fact. There really are more people to thank than could reasonably be put into print. If nothing else, including them all would make the book really hard to hold.

However, there are a few people I would like to specifically thank for their contributions, knowing and unknowing, to the book. The first, of course, is Robert Elliot at Wiley who seemed to think that a presentation that he had liked could possibly be a book of nearly 400 pages. Without him, this book simply wouldn't exist. I would also like to thank Justin Langseth, who was not able to join me in writing this book but was my co-presenter at the talk that started the ball rolling. Hopefully, we will get a chance to reprise that experience. I would also like to thank my editors Charlotte, Rick, Jose, Luke, and Ben, led by Kelly Talbot, who helped find and correct my many mistakes and kept the project on the straight and narrow. Any mistakes that may be left, you can be assured, are my fault.

For less obvious contributions, I would like to thank all of the DDG regulars. At least half, probably more like 80%, of the software in this book is here as a direct result of a conversation I had with someone over a beer. Not too shabby for an informal, loose-knit gathering. Thanks to Mike for first inviting me along and to Matt and Zack for hosting so many events over the years.

Finally, I'd like to thank my colleagues over the years. You all deserve some sort of medal for putting up with my various harebrained schemes and tinkering. An especially big shout-out goes to the adBrite crew. We built a lot of cool stuff that I know for a fact is still cutting edge. Caroline Moon gets a big thank you for not being too concerned when her analytics folks wandered off and started playing with this newfangled “Hadoop” thing and started collecting more servers than she knew we had. I'd also especially like to thank Daniel Issen and Vadim Geshel. We didn't always see eye-to-eye (and probably still don't), but what you see in this book is here in large part to arguments I've had with those two.

Introduction

Overview and Organization of This Book

Dealing with streaming data involves a lot of moving parts and drawing from many different aspects of software development and engineering. On the one hand, streaming data requires a resilient infrastructure that can move data quickly and easily. On the other, the need to have processing "keep up" with data collection and scale to accommodate larger and larger data streams imposes some restrictions that favor the use of certain types of exotic data structures. Finally, once the data has been collected and processed, what do you do with it? There are several immediate applications that most organizations have and more are being considered all the time.This book tries to bring together all of these aspects of streaming data in a way that can serve as an introduction to a broad audience while still providing some use to more advanced readers. The hope is that the reader of this book would feel confident taking a proof-of-concept streaming data project in their organization from start to finish with the intent to release it into a production environment. Since that requires the implementation of both infrastructure and algorithms, this book is divided into two distinct parts.

Part I, "Streaming Analytics Architecture," is focused on the architecture of the streaming data system itself and the operational aspects of the system. If the data is streaming but is still processed in a batch mode, it is no longer streaming data. It is batch data that happens to be collected continuously, which is perfectly sufficient for many use cases. However, the assumption of this book is that some benefit is realized by the fact that the data is available to the end user shortly after it has been generated, and so the book covers the tools and techniques needed to accomplish the task.

To begin, the concepts and features underlying a streaming framework are introduced. This includes the various components of the system. Although not all projects will use all components at first, they are eventually present in all mature streaming infrastructures. These components are then discussed in the context of the key features of a streaming architecture: availability, scalability, and latency.

The remainder of Part I focuses on the nuts and bolts of implementing or configuring each component. The widespread availability of frameworks for each component has mostly removed the need to write code to implement each component. Instead, it is a matter of installation, configuration, and, possibly, customization.

Chapters 3 and 4 introduce the tool needed to construct and coordinate a data motion system. Depending on the environment, software might be developed to integrate directly with this system or existing software adapted to the system. Both are discussed with their relevant pros and cons.

Once the data is moving, the data must be processed and, eventually stored. This is covered in Chapters 5 and Chapter 6. These two chapters introduce popular streaming processing software and options for storing the data.

Part II of the book focuses on on the application of this infrastructure to various problems. The dashboard and alerting system formed the original application of streaming data collection and are the first application covered in Part II.

Chapter 7 covers the delivery of data from the streaming environment to the end user. This is the core mechanism used in the construction of dashboards and other monitoring applications. Once delivered, the data must be presented to the user, and this chapter also includes a section on building dashboard visualizations in a web-based setting.

Of course, the data to be delivered must often be aggregated by the processing system. Chapter 8

covers the aggregation of data for the streaming environment. In particular, it covers the aggregation of multi-resolution time-series data for the eventual delivery to the dashboards discussed in Chapter 7.

After aggregating the data, questions begin to arise about patterns in the data. Is there a trend over time? Is this behavior truly different than previously observed behavior? To answer these questions, you need some knowledge of statistics and the behavior of random processes (which generally includes anything with large scale data collection). Chapter 9 provides a brief introduction to the basics of statistics and probability. Along with this introduction comes the concept of statistical sampling, which can be used to compute more complicated metrics than simple aggregates.

Though sampling is the traditional mechanism for approximating complicated metrics, certain metrics can be better calculated through other mechanisms. These probabilistic data structures, called sketches, are discussed in Chapter 10, making heavy use of the introduction to probability in Chapter 9. These data structures also generally have fast updates and low memory footprints, making them especially useful in streaming settings.

Finally, Chapter 11 discusses some further topics beyond aggregation that can be applied to streaming data. A number of topics are covered in this chapter, providing an introduction to topics that often fill entire books on their own. The first topic is models for streaming data taken from both the statistics and machine learning community. These models provide the basis for a number of applications, such as forecasting. In forecasting, the model is used to provide an estimate of future values. Since the data is streaming, these predictions can be compared to the reality and used to further refine the models. Forecasting has a number of uses, such as anomaly detection, which are also discussed in Chapter 11.

Chapter 11 also briefly touches on the topic of optimization and A/B testing. If you can forecast the expected response of, for example, two different website designs, it makes sense to use that information to show the design with the better response. Of course, forecasts aren't perfect and could provide bad estimates for the performance of each design. The only way to improve a forecast for a particular design is to gather more data. Then you need to determine how often each design should be shown such that the forecast can be improved without sacrificing performance due to showing a less popular design. Chapter 11 provides a simple mechanism called the multi-armed bandit that is often used to solve exactly these problems. This offers a jumping off point for further explorations of the optimization problem.

Who Should Read This Book

As mentioned at the beginning of this "Introduction," the intent of this book is to appeal to a fairly broad audience within the software community. The book is designed for an audience that is interested in getting started in streaming data and its management. As such, the book is intended to be read linearly, with the end goal being a comprehensive grasp of the basics of streaming data analysis.

That said, this book can also be of interest to experts in one part of the field but not the other. For example, a data analyst or data scientist likely has a strong background in the statistical approaches discussed in Chapter 9 and Chapter 11. They are also likely to have some experience with dashboard applications like those discussed in Chapter 7 and the aggregation techniques in Chapter 8. However, they may not have much knowledge of the probabilistic data structures in Chapter 10. The first six chapters may also be of interest if not to actually implement the infrastructure but to understand the design tradeoffs that affect the delivery of the data they analyze.

Similarly, people more focused on the operational and infrastructural pieces likely already know quite a bit about the topics discussed in Chapters 1 through 6. They may not have dealt with the specific software, but they have certainly tackled many of the same problems. The second part of the book, Chapters 7 through 11, is likely to be more interesting to them. Systems monitoring was one of the first applications of streaming data, and tools like anamoly detection can, for example, be put to good use in developing robust fault detection mechanisms.

Tools You Will Need

Like it or not, large swaths of the data infrastructure world are built on the Java Virtual Machine. There are a variety of reasons for this, but ultimately it is a required tool for this book. The software and examples used in this book were developed against Java 7, though it should generally work against Java 6 or Java 8. Readers should ensure that an appropriate Java Development Kit is installed on their systems.

Since Java is used, it is useful to have an editor installed. The software in this book was written using Eclipse, and the projects are also structured using the Maven build system. Installing both of these will help to build the examples included in this book.

Other packages are also used throughout the book, and their installation is covered in their appropriate chapters.

This book uses some basic mathematic terms and formulas. If your math skills are rusty and you find these concepts a little challenging, a helpful resource is *A First Course in Probability* by Sheldon Ross.

What's on the Website

The website includes code packages for all of the examples included in each chapter. The code for each chapter is divided into separate modules.

Some code is used in multiple chapters. In this case, the code is copied to each module so that they are self-contained.

The website also contains a copy of the Samza source code. Samza is a fairly new project, and the codebase has been moving fast enough that the examples in this book actually broke between the writing and editing of the relevant chapters. To avoid this being a problem for readers, we have opted to include a version of the project that is known to work with the code in this book. Please go to `www.wiley.com/go/realtimeanalyticsstreamingdata`.

Time to Dive In

It's now time to dive into the actual building of a streaming data system. This is not the only way of doing it, but it's the one I have arrived at after several different attempts over more years than I really care to think about it. I think it works pretty well, but there are always new things on the horizon. It's an exciting time. I hope this book will help you avoid at least some of the mistakes I have made along the way.

Go find some data, and let's start building something amazing.

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.