Design and implement data processing, lifecycle management, and analytic workflows with the cutting-edge toolbox of Hadoop 2.
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introduction'
- A synopsis of the book’s content
- More information on Learning Hadoop 2

About the Authors

Garry Turkington has over 15 years of industry experience, most of which has been focused on the design and implementation of large-scale distributed systems. In his current role as the CTO at Improve Digital, he is primarily responsible for the realization of systems that store, process, and extract value from the company's large data volumes. Before joining Improve Digital, he spent time at Amazon.co.uk, where he led several software development teams, building systems that process the Amazon catalog data for every item worldwide. Prior to this, he spent a decade in various government positions in both the UK and the USA.

He has BSc and PhD degrees in Computer Science from Queens University Belfast in Northern Ireland, and a Master's degree in Engineering in Systems Engineering from Stevens Institute of Technology in the USA. He is the author of Hadoop Beginners Guide, published by Packt Publishing in 2013, and is a committer on the Apache Samza project.

I would like to thank my wife Lea and mother Sarah for their support and patience through the writing of another book and my daughter Maya for frequently cheering me up and asking me hard questions. I would also like to thank Gabriele for being such an amazing co-author on this project.
Gabriele Modena is a data scientist at Improve Digital. In his current position, he uses Hadoop to manage, process, and analyze behavioral and machine-generated data. Gabriele enjoys using statistical and computational methods to look for patterns in large amounts of data. Prior to his current job in ad tech he held a number of positions in Academia and Industry where he did research in machine learning and artificial intelligence.

He holds a BSc degree in Computer Science from the University of Trento, Italy and a Research MSc degree in Artificial Intelligence: Learning Systems, from the University of Amsterdam in the Netherlands.

First and foremost, I want to thank Laura for her support, constant encouragement and endless patience putting up with far too many "can't do, I'm working on the Hadoop book". She is my rock and I dedicate this book to her.

A special thank you goes to Amit, Atidhe, Davide, Jakob, James and Valerie, whose invaluable feedback and commentary made this work possible.

Finally, I'd like to thank my co-author, Garry, for bringing me on board with this project; it has been a pleasure working together.
Learning Hadoop 2

This book will take you on a hands-on exploration of the wonderful world that is Hadoop 2 and its rapidly growing ecosystem. Building on the solid foundation from the earlier versions of the platform, Hadoop 2 allows multiple data processing frameworks to be executed on a single Hadoop cluster.

To give an understanding of this significant evolution, we will explore both how these new models work and also show their applications in processing large data volumes with batch, iterative, and near-real-time algorithms.

What This Book Covers

Chapter 1, Introduction, gives the background to Hadoop and the Big Data problems it looks to solve. We also highlight the areas in which Hadoop 1 had room for improvement.

Chapter 2, Storage, delves into the Hadoop Distributed File System, where most data processed by Hadoop is stored. We examine the particular characteristics of HDFS, show how to use it, and discuss how it has improved in Hadoop 2. We also introduce ZooKeeper, another storage system within Hadoop, upon which many of its high-availability features rely.

Chapter 3, Processing – MapReduce and Beyond, first discusses the traditional Hadoop processing model and how it is used. We then discuss how Hadoop 2 has generalized the platform to use multiple computational models, of which MapReduce is merely one.

Chapter 4, Real-time Computation with Samza, takes a deeper look at one of these alternative processing models enabled by Hadoop 2. In particular, we look at how to process real-time streaming data with Apache Samza.

Chapter 5, Iterative Computation with Spark, delves into a very different alternative processing model. In this chapter, we look at how Apache Spark provides the means to do iterative processing.

Chapter 6, Data Analysis with Pig, demonstrates how Apache Pig makes the traditional computational model of MapReduce easier to use by providing a language to describe data flows.

Chapter 7, Hadoop and SQL, looks at how the familiar SQL language has been implemented atop data stored in Hadoop. Through the use of Apache Hive and describing alternatives such as Cloudera Impala, we show how Big Data processing can be made possible using existing skills and tools.
Chapter 8, *Data Lifecycle Management*, takes a look at the bigger picture of just how to manage all that data that is to be processed in Hadoop. Using Apache Oozie, we show how to build up workflows to ingest, process, and manage data.

Chapter 9, *Making Development Easier*, focuses on a selection of tools aimed at helping a developer get results quickly. Through the use of Hadoop streaming, Apache Crunch and Kite, we show how the use of the right tool can speed up the development loop or provide new APIs with richer semantics and less boilerplate.

Chapter 10, *Running a Hadoop Cluster*, takes a look at the operational side of Hadoop. By focusing on the areas of interest to developers, such as cluster management, monitoring, and security, this chapter should help you to work better with your operations staff.

Chapter 11, *Where to Go Next*, takes you on a whirlwind tour through a number of other projects and tools that we feel are useful, but could not cover in detail in the book due to space constraints. We also give some pointers on where to find additional sources of information and how to engage with the various open source communities.
This book will teach you how to build amazing systems using the latest release of Hadoop. Before you change the world though, we need to do some groundwork, which is where this chapter comes in.

In this introductory chapter, we will cover the following topics:

- A brief refresher on the background to Hadoop
- A walk-through of Hadoop's evolution
- The key elements in Hadoop 2
- The Hadoop distributions we'll use in this book
- The dataset we'll use for examples

A note on versioning

In Hadoop 1, the version history was somewhat convoluted with multiple forked branches in the 0.2x range, leading to odd situations, where a 1.x version could, in some situations, have fewer features than a 0.23 release. In the version 2 codebase, this is fortunately much more straightforward, but it's important to clarify exactly which version we will use in this book.

Hadoop 2.0 was released in alpha and beta versions, and along the way, several incompatible changes were introduced. There was, in particular, a major API stabilization effort between the beta and final release stages.
Hadoop 2.2.0 was the first general availability (GA) release of the Hadoop 2 codebase, and its interfaces are now declared stable and forward compatible. We will therefore use the 2.2 product and interfaces in this book. Though the principles will be usable on a 2.0 beta, in particular, there will be API incompatibilities in the beta. This is particularly important as MapReduce v2 was back-ported to Hadoop 1 by several distribution vendors, but these products were based on the beta and not the GA APIs. If you are using such a product, then you will encounter these incompatible changes. It is recommended that a release based upon Hadoop 2.2 or later is used for both the development and the production deployments of any Hadoop 2 workloads.

The background of Hadoop

We're assuming that most readers will have a little familiarity with Hadoop, or at the very least, with big data-processing systems. Consequently, we won't give a detailed background as to why Hadoop is successful or the types of problem it helps to solve in this book. However, particularly because of some aspects of Hadoop 2 and the other products we will use in later chapters, it is useful to give a sketch of how we see Hadoop fitting into the technology landscape and which are the particular problem areas where we believe it gives the most benefit.

In ancient times, before the term "big data" came into the picture (which equates to maybe a decade ago), there were few options to process datasets of sizes in terabytes and beyond. Some commercial databases could, with very specific and expensive hardware setups, be scaled to this level, but the expertise and capital expenditure required made it an option for only the largest organizations. Alternatively, one could build a custom system aimed at the specific problem at hand. This suffered from some of the same problems (expertise and cost) and added the risk inherent in any cutting-edge system. On the other hand, if a system was successfully constructed, it was likely a very good fit to the need.

Few small- to mid-size companies even worried about this space, not only because the solutions were out of their reach, but they generally also didn't have anything close to the data volumes that required such solutions. As the ability to generate very large datasets became more common, so did the need to process that data.
Even though large data became more democratized and was no longer the domain of the privileged few, major architectural changes were required if the data-processing systems could be made affordable to smaller companies. The first big change was to reduce the required upfront capital expenditure on the system; that means no high-end hardware or expensive software licenses. Previously, high-end hardware would have been utilized most commonly in a relatively small number of very large servers and storage systems, each of which had multiple approaches to avoid hardware failures. Though very impressive, such systems are hugely expensive, and moving to a larger number of lower-end servers would be the quickest way to dramatically reduce the hardware cost of a new system. Moving more toward commodity hardware instead of the traditional enterprise-grade equipment would also mean a reduction in capabilities in the area of resilience and fault tolerance. Those responsibilities would need to be taken up by the software layer. Smarter software, dumber hardware.

Google started the change that would eventually be known as Hadoop, when in 2003, and in 2004, they released two academic papers describing the Google File System (GFS) (http://research.google.com/archive/gfs.html) and MapReduce (http://research.google.com/archive/mapreduce.html). The two together provided a platform for very large-scale data processing in a highly efficient manner. Google had taken the build-it-yourself approach, but instead of constructing something aimed at one specific problem or dataset, they instead created a platform on which multiple processing applications could be implemented. In particular, they utilized large numbers of commodity servers and built GFS and MapReduce in a way that assumed hardware failures would be commonplace and were simply something that the software needed to deal with.

At the same time, Doug Cutting was working on the Nutch open source web crawler. He was working on elements within the system that resonated strongly once the Google GFS and MapReduce papers were published. Doug started work on open source implementations of these Google ideas, and Hadoop was soon born, firstly, as a subproject of Lucene, and then as its own top-level project within the Apache Software Foundation.

Yahoo! hired Doug Cutting in 2006 and quickly became one of the most prominent supporters of the Hadoop project. In addition to often publicizing some of the largest Hadoop deployments in the world, Yahoo! allowed Doug and other engineers to contribute to Hadoop while employed by the company, not to mention contributing back some of its own internally developed Hadoop improvements and extensions.
Components of Hadoop

The broad Hadoop umbrella project has many component subprojects, and we'll discuss several of them in this book. At its core, Hadoop provides two services: storage and computation. A typical Hadoop workflow consists of loading data into the Hadoop Distributed File System (HDFS) and processing using the MapReduce API or several tools that rely on MapReduce as an execution framework.

Both layers are direct implementations of Google's own GFS and MapReduce technologies.

Common building blocks

Both HDFS and MapReduce exhibit several of the architectural principles described in the previous section. In particular, the common principles are as follows:

- Both are designed to run on clusters of commodity (that is, low to medium specification) servers
- Both scale their capacity by adding more servers (scale-out) as opposed to the previous models of using larger hardware (scale-up)
- Both have mechanisms to identify and work around failures
- Both provide most of their services transparently, allowing the user to concentrate on the problem at hand
- Both have an architecture where a software cluster sits on the physical servers and manages aspects such as application load balancing and fault tolerance, without relying on high-end hardware to deliver these capabilities
Storage
HDFS is a filesystem, though not a POSIX-compliant one. This basically means that it does not display the same characteristics as that of a regular filesystem. In particular, the characteristics are as follows:

• HDFS stores files in blocks that are typically at least 64 MB or (more commonly now) 128 MB in size, much larger than the 4-32 KB seen in most filesystems
• HDFS is optimized for throughput over latency; it is very efficient at streaming reads of large files but poor when seeking for many small ones
• HDFS is optimized for workloads that are generally write-once and read-many
• Instead of handling disk failures by having physical redundancies in disk arrays or similar strategies, HDFS uses replication. Each of the blocks comprising a file is stored on multiple nodes within the cluster, and a service called the NameNode constantly monitors to ensure that failures have not dropped any block below the desired replication factor. If this does happen, then it schedules the making of another copy within the cluster.

Computation
MapReduce is an API, an execution engine, and a processing paradigm; it provides a series of transformations from a source into a result dataset. In the simplest case, the input data is fed through a map function and the resultant temporary data is then fed through a reduce function.

MapReduce works best on semistructured or unstructured data. Instead of data conforming to rigid schemas, the requirement is instead that the data can be provided to the map function as a series of key-value pairs. The output of the map function is a set of other key-value pairs, and the reduce function performs aggregation to collect the final set of results.
Hadoop provides a standard specification (that is, interface) for the map and reduce phases, and the implementation of these are often referred to as mappers and reducers. A typical MapReduce application will comprise a number of mappers and reducers, and it's not unusual for several of these to be extremely simple. The developer focuses on expressing the transformation between the source and the resultant data, and the Hadoop framework manages all aspects of job execution and coordination.

Better together

It is possible to appreciate the individual merits of HDFS and MapReduce, but they are even more powerful when combined. They can be used individually, but when they are together, they bring out the best in each other, and this close interworking was a major factor in the success and acceptance of Hadoop 1.

When a MapReduce job is being planned, Hadoop needs to decide on which host to execute the code in order to process the dataset most efficiently. If the MapReduce cluster hosts are all pulling their data from a single storage host or array, then this largely doesn't matter as the storage system is a shared resource that will cause contention. If the storage system was more transparent and allowed MapReduce to manipulate its data more directly, then there would be an opportunity to perform the processing closer to the data, building on the principle of it being less expensive to move processing than data.

The most common deployment model for Hadoop sees the HDFS and MapReduce clusters deployed on the same set of servers. Each host that contains data and the HDFS component to manage the data also hosts a MapReduce component that can schedule and execute data processing. When a job is submitted to Hadoop, it can use the locality optimization to schedule data on the hosts where data resides as much as possible, thus minimizing network traffic and maximizing performance.

Hadoop 2 – what's the big deal?

If we look at the two main components of the core Hadoop distribution, storage and computation, we see that Hadoop 2 has a very different impact on each of them. Whereas the HDFS found in Hadoop 2 is mostly a much more feature-rich and resilient product than the HDFS in Hadoop 1, for MapReduce, the changes are much more profound and have, in fact, altered how Hadoop is perceived as a processing platform in general. Let's look at HDFS in Hadoop 2 first.
Storage in Hadoop 2

We'll discuss the HDFS architecture in more detail in Chapter 2, Storage, but for now, it's sufficient to think of a master-slave model. The slave nodes (called DataNodes) hold the actual filesystem data. In particular, each host running a DataNode will typically have one or more disks onto which files containing the data for each HDFS block are written. The DataNode itself has no understanding of the overall filesystem; its role is to store, serve, and ensure the integrity of the data for which it is responsible.

The master node (called the NameNode) is responsible for knowing which of the DataNodes holds which block and how these blocks are structured to form the filesystem. When a client looks at the filesystem and wishes to retrieve a file, it's via a request to the NameNode that the list of required blocks is retrieved.

This model works well and has been scaled to clusters with tens of thousands of nodes at companies such as Yahoo! So, though it is scalable, there is a resiliency risk; if the NameNode becomes unavailable, then the entire cluster is rendered effectively useless. No HDFS operations can be performed, and since the vast majority of installations use HDFS as the storage layer for services, such as MapReduce, they also become unavailable even if they are still running without problems.

More catastrophically, the NameNode stores the filesystem metadata to a persistent file on its local filesystem. If the NameNode host crashes in a way that this data is not recoverable, then all data on the cluster is effectively lost forever. The data will still exist on the various DataNodes, but the mapping of which blocks comprise which files is lost. This is why, in Hadoop 1, the best practice was to have the NameNode synchronously write its filesystem metadata to both local disks and at least one remote network volume (typically via NFS).

Several NameNode high-availability (HA) solutions have been made available by third-party suppliers, but the core Hadoop product did not offer such resilience in Version 1. Given this architectural single point of failure and the risk of data loss, it won't be a surprise to hear that NameNode HA is one of the major features of HDFS in Hadoop 2 and is something we'll discuss in detail in later chapters. The feature provides both a standby NameNode that can be automatically promoted to service all requests should the active NameNode fail, but also builds additional resilience for the critical filesystem metadata atop this mechanism.
HDFS in Hadoop 2 is still a non-POSIX filesystem; it still has a very large block size and it still trades latency for throughput. However, it does now have a few capabilities that can make it look a little more like a traditional filesystem. In particular, the core HDFS in Hadoop 2 now can be remotely mounted as an NFS volume. This is another feature that was previously offered as a proprietary capability by third-party suppliers but is now in the main Apache codebase.

Overall, the HDFS in Hadoop 2 is more resilient and can be more easily integrated into existing workflows and processes. It's a strong evolution of the product found in Hadoop 1.

Computation in Hadoop 2

The work on HDFS 2 was started before a direction for MapReduce crystallized. This was likely due to the fact that features such as NameNode HA were such an obvious path that the community knew the most critical areas to address. However, MapReduce didn't really have a similar list of areas of improvement, and that's why, when the MRv2 initiative started, it wasn't completely clear where it would lead.

Perhaps the most frequent criticism of MapReduce in Hadoop 1 was how its batch processing model was ill-suited to problem domains where faster response times were required. Hive, for example, which we'll discuss in Chapter 7, Hadoop and SQL, provides a SQL-like interface onto HDFS data, but, behind the scenes, the statements are converted into MapReduce jobs that are then executed like any other. A number of other products and tools took a similar approach, providing a specific user-facing interface that hid a MapReduce translation layer.

Though this approach has been very successful, and some amazing products have been built, the fact remains that in many cases, there is a mismatch as all of these interfaces, some of which expect a certain type of responsiveness, are behind the scenes, being executed on a batch-processing platform. When looking to enhance MapReduce, improvements could be made to make it a better fit to these use cases, but the fundamental mismatch would remain. This situation led to a significant change of focus of the MRv2 initiative; perhaps MapReduce itself didn't need change, but the real need was to enable different processing models on the Hadoop platform. Thus was born Yet Another Resource Negotiator (YARN).

Looking at MapReduce in Hadoop 1, the product actually did two quite different things; it provided the processing framework to execute MapReduce computations, but it also managed the allocation of this computation across the cluster. Not only did it direct data to and between the specific map and reduce tasks, but it also determined where each task would run, and managed the full job life cycle, monitoring the health of each task and node, rescheduling if any failed, and so on.
This is not a trivial task, and the automated parallelization of workloads has always been one of the main benefits of Hadoop. If we look at MapReduce in Hadoop 1, we see that after the user defines the key criteria for the job, everything else is the responsibility of the system. Critically, from a scale perspective, the same MapReduce job can be applied to datasets of any volume hosted on clusters of any size. If the data is 1 GB in size and on a single host, then Hadoop will schedule the processing accordingly. If the data is instead 1 PB in size and hosted across 1,000 machines, then it does likewise. From the user's perspective, the actual scale of the data and cluster is transparent, and aside from affecting the time taken to process the job, it does not change the interface with which to interact with the system.

In Hadoop 2, this role of job scheduling and resource management is separated from that of executing the actual application, and is implemented by YARN.

YARN is responsible for managing the cluster resources, and so MapReduce exists as an application that runs atop the YARN framework. The MapReduce interface in Hadoop 2 is completely compatible with that in Hadoop 1, both semantically and practically. However, under the covers, MapReduce has become a hosted application on the YARN framework.

The significance of this split is that other applications can be written that provide processing models more focused on the actual problem domain and can offload all the resource management and scheduling responsibilities to YARN. The latest versions of many different execution engines have been ported onto YARN, either in a production-ready or experimental state, and it has shown that the approach can allow a single Hadoop cluster to run everything from batch-oriented MapReduce jobs through fast-response SQL queries to continuous data streaming and even to implement models such as graph processing and the Message Passing Interface (MPI) from the High Performance Computing (HPC) world. The following diagram shows the architecture of Hadoop 2:
This is why much of the attention and excitement around Hadoop 2 has been focused on YARN and frameworks that sit on top of it, such as Apache Tez and Apache Spark. With YARN, the Hadoop cluster is no longer just a batch-processing engine; it is the single platform on which a vast array of processing techniques can be applied to the enormous data volumes stored in HDFS. Moreover, applications can build on these computation paradigms and execution models.

The analogy that is achieving some traction is to think of YARN as the processing kernel upon which other domain-specific applications can be built. We'll discuss YARN in more detail in this book, particularly in Chapter 3, Processing – MapReduce and Beyond, Chapter 4, Real-time Computation with Samza, and Chapter 5, Iterative Computation with Spark.

**Distributions of Apache Hadoop**

In the very early days of Hadoop, the burden of installing (often building from source) and managing each component and its dependencies fell on the user. As the system became more popular and the ecosystem of third-party tools and libraries started to grow, the complexity of installing and managing a Hadoop deployment increased dramatically to the point where providing a coherent offer of software packages, documentation, and training built around the core Apache Hadoop has become a business model. Enter the world of distributions for Apache Hadoop.

Hadoop distributions are conceptually similar to how Linux distributions provide a set of integrated software around a common core. They take the burden of bundling and packaging software themselves and provide the user with an easy way to install, manage, and deploy Apache Hadoop and a selected number of third-party libraries. In particular, the distribution releases deliver a series of product versions that are certified to be mutually compatible. Historically, putting together a Hadoop-based platform was often greatly complicated by the various version interdependencies.

Cloudera (http://www.cloudera.com), Hortonworks (http://www.hortonworks.com), and MapR (http://www.mapr.com) are amongst the first to have reached the market, each characterized by different approaches and selling points. Hortonworks positions itself as the open source player; Cloudera is also committed to open source but adds proprietary bits for configuring and managing Hadoop; MapR provides a hybrid open source/proprietary Hadoop distribution characterized by a proprietary NFS layer instead of HDFS and a focus on providing services.

Another strong player in the distributions ecosystem is Amazon, which offers a version of Hadoop called Elastic MapReduce (EMR) on top of the Amazon Web Services (AWS) infrastructure.
With the advent of Hadoop 2, the number of available distributions for Hadoop has increased dramatically, far in excess of the four we mentioned. A possibly incomplete list of software offerings that includes Apache Hadoop can be found at [http://wiki.apache.org/hadoop/Distributions%20and%20Commercial%20Support](http://wiki.apache.org/hadoop/Distributions%20and%20Commercial%20Support).

**A dual approach**

In this book, we will discuss both the building and the management of local Hadoop clusters in addition to showing how to push the processing into the cloud via EMR.

The reason for this is twofold: firstly, though EMR makes Hadoop much more accessible, there are aspects of the technology that only become apparent when manually administering the cluster. Although it is also possible to use EMR in a more manual mode, we'll generally use a local cluster for such explorations. Secondly, though it isn't necessarily an either/or decision, many organizations use a mixture of in-house and cloud-hosted capacities, sometimes due to a concern of over reliance on a single external provider, but practically speaking, it's often convenient to do development and small-scale tests on local capacity and then deploy at production scale into the cloud.

In a few of the later chapters, where we discuss additional products that integrate with Hadoop, we'll mostly give examples of local clusters, as there is no difference between how the products work regardless of where they are deployed.

**AWS – infrastructure on demand from Amazon**

AWS is a set of cloud-computing services offered by Amazon. We will use several of these services in this book.

**Simple Storage Service (S3)**

Amazon's Simple Storage Service (S3), found at [http://aws.amazon.com/s3/](http://aws.amazon.com/s3/), is a storage service that provides a simple key-value storage model. Using web, command-line, or programmatic interfaces to create objects, which can be anything from text files to images to MP3s, you can store and retrieve your data based on a hierarchical model. In this model, you create buckets that contain objects. Each bucket has a unique identifier, and within each bucket, every object is uniquely named. This simple strategy enables an extremely powerful service for which Amazon takes complete responsibility (for service scaling, in addition to reliability and availability of data).
Elastic MapReduce (EMR)

Amazon's Elastic MapReduce, found at [http://aws.amazon.com/elasticmapreduce/](http://aws.amazon.com/elasticmapreduce/), is basically Hadoop in the cloud. Using any of the multiple interfaces (web console, CLI, or API), a Hadoop workflow is defined with attributes such as the number of Hadoop hosts required and the location of the source data. The Hadoop code implementing the MapReduce jobs is provided, and the virtual Go button is pressed.

In its most impressive mode, EMR can pull source data from S3, process it on a Hadoop cluster it creates on Amazon's virtual host on-demand service EC2, push the results back into S3, and terminate the Hadoop cluster and the EC2 virtual machines hosting it. Naturally, each of these services has a cost (usually on per GB stored and server-time usage basis), but the ability to access such powerful data-processing capabilities with no need for dedicated hardware is a powerful one.

Getting started

We will now describe the two environments we will use throughout the book: Cloudera's QuickStart virtual machine will be our reference system on which we will show all examples, but we will additionally demonstrate some examples on Amazon's EMR when there is some particularly valuable aspect to running the example in the on-demand service.

Although the examples and code provided are aimed at being as general-purpose and portable as possible, our reference setup, when talking about a local cluster, will be Cloudera running atop CentOS Linux.

For the most part, we will show examples that make use of, or are executed from, a terminal prompt. Although Hadoop's graphical interfaces have improved significantly over the years (for example, the excellent HUE and Cloudera Manager), when it comes to development, automation, and programmatic access to the system, the command line is still the most powerful tool for the job.

All examples and source code presented in this book can be downloaded from [https://github.com/learninghadoop2/book-examples](https://github.com/learninghadoop2/book-examples). In addition, we have a home page for the book where we will publish updates and related material at [http://learninghadoop2.com](http://learninghadoop2.com).
Chapter 1

**Cloudera QuickStart VM**

One of the advantages of Hadoop distributions is that they give access to easy-to-install, packaged software. Cloudera takes this one step further and provides a freely downloadable Virtual Machine instance of its latest distribution, known as the CDH QuickStart VM, deployed on top of CentOS Linux.

In the remaining parts of this book, we will use the CDH5.0.0 VM as the reference and baseline system to run examples and source code. Images of the VM are available for VMware (http://www.vmware.com/nl/products/player/), KVM (http://www.linux-kvm.org/page/Main_Page), and VirtualBox (https://www.virtualbox.org/) virtualization systems.

**Amazon EMR**

Before using Elastic MapReduce, we need to set up an AWS account and register it with the necessary services.

**Creating an AWS account**

Amazon has integrated its general accounts with AWS, which means that, if you already have an account for any of the Amazon retail websites, this is the only account you will need to use AWS services.

Note that AWS services have a cost; you will need an active credit card associated with the account to which charges can be made.

If you require a new Amazon account, go to http://aws.amazon.com, select Create a new AWS account, and follow the prompts. Amazon has added a free tier for some services, so you might find that in the early days of testing and exploration, you are keeping many of your activities within the noncharged tier. The scope of the free tier has been expanding, so make sure you know what you will and won't be charged for.
Introduction

Signing up for the necessary services
Once you have an Amazon account, you will need to register it for use with the required AWS services, that is, Simple Storage Service (S3), Elastic Compute Cloud (EC2), and Elastic MapReduce. There is no cost to simply sign up to any AWS service; the process just makes the service available to your account.

Go to the S3, EC2, and EMR pages linked from http://aws.amazon.com, click on the Sign up button on each page, and then follow the prompts.

Using Elastic MapReduce
Having created an account with AWS and registered all the required services, we can proceed to configure programmatic access to EMR.

Getting Hadoop up and running

Caution! This costs real money!

Before going any further, it is critical to understand that use of AWS services will incur charges that will appear on the credit card associated with your Amazon account. Most of the charges are quite small and increase with the amount of infrastructure consumed; storing 10 GB of data in S3 costs 10 times more than 1 GB, and running 20 EC2 instances costs 20 times as much as a single one. There are tiered cost models, so the actual costs tend to have smaller marginal increases at higher levels. But you should read carefully through the pricing sections for each service before using any of them. Note also that currently data transfer out of AWS services, such as EC2 and S3, is chargeable, but data transfer between services is not. This means it is often most cost-effective to carefully design your use of AWS to keep data within AWS through as much of the data processing as possible. For information regarding AWS and EMR, consult http://aws.amazon.com/elasticmapreduce/#pricing.

How to use EMR
Amazon provides both web and command-line interfaces to EMR. Both interfaces are just a frontend to the very same system; a cluster created with the command-line interface can be inspected and managed with the web tools and vice-versa.

For the most part, we will be using the command-line tools to create and manage clusters programmatically and will fall back on the web interface cases where it makes sense to do so.
Chapter 1

AWS credentials

Before using either programmatic or command-line tools, we need to look at how an account holder authenticates to AWS to make such requests.

Each AWS account has several identifiers, such as the following, that are used when accessing the various services:

- **Account ID**: each AWS account has a numeric ID.
- **Access key**: the associated access key is used to identify the account making the request.
- **Secret access key**: the partner to the access key is the secret access key. The access key is not a secret and could be exposed in service requests, but the secret access key is what you use to validate yourself as the account owner. Treat it like your credit card.
- **Key pairs**: these are the key pairs used to log in to EC2 hosts. It is possible to either generate public/private key pairs within EC2 or to import externally generated keys into the system.

User credentials and permissions are managed via a web service called **Identity and Access Management (IAM)**, which you need to sign up to in order to obtain access and secret keys.

If this sounds confusing, it's because it is, at least at first. When using a tool to access an AWS service, there's usually the single, upfront step of adding the right credentials to a configured file, and then everything just works. However, if you do decide to explore programmatic or command-line tools, it will be worth investing a little time to read the documentation for each service to understand how its security works. More information on creating an AWS account and obtaining access credentials can be found at [http://docs.aws.amazon.com/iam](http://docs.aws.amazon.com/iam).

The AWS command-line interface

Each AWS service historically had its own set of command-line tools. Recently though, Amazon has created a single, unified command-line tool that allows access to most services. The Amazon CLI can be found at [http://aws.amazon.com/cli](http://aws.amazon.com/cli).

It can be installed from a tarball or via the `pip` or `easy_install` package managers. On the CDH QuickStart VM, we can install `awscli` using the following command:

```
$ pip install awscli
```
In order to access the API, we need to configure the software to authenticate to AWS using our access and secret keys.

This is also a good moment to set up an EC2 key pair by following the instructions provided at https://console.aws.amazon.com/ec2/home?region=us-east-1#c=EC2&s=KeyPairs.

Although a key pair is not strictly necessary to run an EMR cluster, it will give us the capability to remotely log in to the master node and gain low-level access to the cluster.

The following command will guide you through a series of configuration steps and store the resulting configuration in the .aws/credential file:

$ aws configure

Once the CLI is configured, we can query AWS with aws <service> <arguments>. To create and query an S3 bucket use something like the following command. Note that S3 buckets need to be globally unique across all AWS accounts, so most common names, such as s3://mybucket, will not be available:

$ aws s3 mb s3://learninghadoop2
$ aws s3 ls

We can provision an EMR cluster with five m1.xlarge nodes using the following commands:

$ aws emr create-cluster --name "EMR cluster" \
--ami-version 3.2.0 \
--instance-type m1.xlarge \
--instance-count 5 \
--log-uri s3://learninghadoop2/emr-logs

Where --ami-version is the ID of an Amazon Machine Image template (http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html), and --log-uri instructs EMR to collect logs and store them in the learninghadoop2 S3 bucket.

If you did not specify a default region when setting up the AWS CLI, then you will also have to add one to most EMR commands in the AWS CLI using the --region argument; for example, --region eu-west-1 is run to use the EU Ireland region. You can find details of all available AWS regions at http://docs.aws.amazon.com/general/latest/gr/rande.html.
We can submit workflows by adding steps to a running cluster using the following command:

```bash
$ aws emr add-steps --cluster-id <cluster> --steps <steps>
```

To terminate the cluster, use the following command line:

```bash
$ aws emr terminate-clusters --cluster-id <cluster>
```

In later chapters, we will show you how to add steps to execute MapReduce jobs and Pig scripts.

More information on using the AWS CLI can be found at http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-manage.html.

**Running the examples**

The source code of all examples is available at https://github.com/learninghadoop2/book-examples.

Gradle (http://www.gradle.org/) scripts and configurations are provided to compile most of the Java code. The `gradlew` script included with the example will bootstrap Gradle and use it to fetch dependencies and compile code.

JAR files can be created by invoking the `jar` task via a `gradlew` script, as follows:

```bash
./gradlew jar
```

Jobs are usually executed by submitting a JAR file using the `hadoop jar` command, as follows:

```bash
$ hadoop jar example.jar <MainClass> [-libjars $LIBJARS] arg1 arg2 ... argN
```

The optional `-libjars` parameter specifies runtime third-party dependencies to ship to remote nodes.

Some of the frameworks we will work with, such as Apache Spark, come with their own build and package management tools. Additional information and resources will be provided for these particular cases.

The `copyJar` Gradle task can be used to download third-party dependencies into `build/libjars/<example>/lib`, as follows:

```bash
./gradlew copyJar
```
Introduction

For convenience, we provide a fatJar Gradle task that bundles the example classes and their dependencies into a single JAR file. Although this approach is discouraged in favor of using -libjar, it might come in handy when dealing with dependency issues.

The following command will generate `build/libs/<example>-all.jar`:

```
$ ./gradlew fatJar
```

Data processing with Hadoop

In the remaining chapters of this book, we will introduce the core components of the Hadoop ecosystem as well as a number of third-party tools and libraries that will make writing robust, distributed code an accessible and hopefully enjoyable task. While reading this book, you will learn how to collect, process, store, and extract information from large amounts of structured and unstructured data.

We will use a dataset generated from Twitter's (http://www.twitter.com) real-time fire hose. This approach will allow us to experiment with relatively small datasets locally and, once ready, scale the examples up to production-level data sizes.

Why Twitter?

Thanks to its programmatic APIs, Twitter provides an easy way to generate datasets of arbitrary size and inject them into our local- or cloud-based Hadoop clusters. Other than the sheer size, the dataset that we will use has a number of properties that fit several interesting data modeling and processing use cases.

Twitter data possesses the following properties:

- **Unstructured**: each status update is a text message that can contain references to media content such as URLs and images
- **Structured**: tweets are timestamped, sequential records
- **Graph**: relationships such as replies and mentions can be modeled as a network of interactions
- **Geolocated**: the location where a tweet was posted or where a user resides
- **Real time**: all data generated on Twitter is available via a real-time fire hose

These properties will be reflected in the type of application that we can build with Hadoop. These include examples of sentiment analysis, social network, and trend analysis.
Building our first dataset

Twitter's terms of service prohibit redistribution of user-generated data in any form; for this reason, we cannot make available a common dataset. Instead, we will use a Python script to programmatically access the platform and create a dump of user tweets collected from a live stream.

One service, multiple APIs

Twitter users share more than 200 million tweets, also known as status updates, a day. The platform offers access to this corpus of data via four types of APIs, each of which represents a facet of Twitter and aims at satisfying specific use cases, such as linking and interacting with twitter content from third-party sources (Twitter for Products), programmatic access to specific users' or sites' content (REST), search capabilities across users' or sites' timelines (Search), and access to all content created on the Twitter network in real time (Streaming).

The Streaming API allows direct access to the Twitter stream, tracking keywords, retrieving geotagged tweets from a certain region, and much more. In this book, we will make use of this API as a data source to illustrate both the batch and real-time capabilities of Hadoop. We will not, however, interact with the API itself; rather, we will make use of third-party libraries to offload chores such as authentication and connection management.

Anatomy of a Tweet

Each tweet object returned by a call to the real-time APIs is represented as a serialized JSON string that contains a set of attributes and metadata in addition to a textual message. This additional content includes a numerical ID that uniquely identifies the tweet, the location where the tweet was shared, the user who shared it (user object), whether it was republished by other users (retweeted) and how many times (retweet count), the machine-detected language of its text, whether the tweet was posted in reply to someone and, if so, the user and tweet IDs it replied to, and so on.

The structure of a Tweet, and any other object exposed by the API, is constantly evolving. An up-to-date reference can be found at https://dev.twitter.com/docs/platform-objects/tweets.
Twitter credentials

Twitter makes use of the OAuth protocol to authenticate and authorize access from third-party software to its platform.

The application obtains through an external channel, for instance a web form, the following pair of credentials:

- Consumer key
- Consumer secret

The consumer secret is never directly transmitted to the third party as it is used to sign each request.

The user authorizes the application to access the service via a three-way process that, once completed, grants the application a token consisting of the following:

- Access token
- Access secret

Similarly, to the consumer, the access secret is never directly transmitted to the third party, and it is used to sign each request.

In order to use the Streaming API, we will first need to register an application and grant it programmatic access to the system. If you require a new Twitter account, proceed to the signup page at https://twitter.com/signup, and fill in the required information. Once this step is completed, we need to create a sample application that will access the API on our behalf and grant it the proper authorization rights. We will do so using the web form found at https://dev.twitter.com/apps.
When creating a new app, we are asked to give it a name, a description, and a URL. The following screenshot shows the settings of a sample application named Learning Hadoop 2 Book Dataset. For the purpose of this book, we do not need to specify a valid URL, so we used a placeholder instead.

Once the form is filled in, we need to review and accept the terms of service and click on the **Create Application** button in the bottom-left corner of the page.

We are now presented with a page that summarizes our application details as seen in the following screenshot; the authentication and authorization credentials can be found under the OAuth Tool tab.
Introduction

We are finally ready to generate our very first Twitter dataset.

Programmatic access with Python

In this section, we will use Python and the tweepy library, found at https://github.com/tweepy/tweepy, to collect Twitter's data. The stream.py file found in the ch1 directory of the book code archive instantiates a listener to the real-time fire hose, grabs a data sample, and echoes each tweet's text to standard output.

The tweepy library can be installed using either the easy_install or pip package managers or by cloning the repository at https://github.com/tweepy/tweepy.

On the CDH QuickStart VM, we can install tweepy using the following command line:

$ pip install tweepy
When invoked with the \texttt{-j} parameter, the script will output a JSON tweet to standard output; \texttt{-t} extracts and prints the text field. We specify how many tweets to print with \texttt{-n <num tweets>}. When \texttt{-n} is not specified, the script will run indefinitely. Execution can be terminated by pressing \textit{Ctrl} + \textit{C}.

The script expects OAuth credentials to be stored as shell environment variables; the following credentials will have to be set in the terminal session from where \texttt{stream.py} will be executed.

\begin{verbatim}
$ export TWITTER_CONSUMER_KEY="your_consumer_key"
$ export TWITTER_CONSUMER_SECRET="your_consumer_secret"
$ export TWITTER_ACCESS_KEY="your_access_key"
$ export TWITTER_ACCESS_SECRET="your_access_secret"
\end{verbatim}

Once the required dependency has been installed and the OAuth data in the shell environment has been set, we can run the program as follows:

\begin{verbatim}
$ python stream.py -t -n 1000 > tweets.txt
\end{verbatim}

We are relying on Linux's shell I/O to redirect the output with the \texttt{>} operator of \texttt{stream.py} to a file called \texttt{tweets.txt}. If everything was executed correctly, you should see a wall of text, where each line is a tweet.

Notice that in this example, we did not make use of Hadoop at all. In the next chapters, we will show how to import a dataset generated from the Streaming API into Hadoop and analyze its content on the local cluster and Amazon EMR.

For now, let's take a look at the source code of \texttt{stream.py}, which can be found at https://github.com/learninghadoop2/book-examples/blob/master/ch1/stream.py:

\begin{verbatim}
import tweepy
import os
import json
import argparse

consumer_key = os.environ['TWITTER_CONSUMER_KEY']
consumer_secret = os.environ['TWITTER_CONSUMER_SECRET']
access_key = os.environ['TWITTER_ACCESS_KEY']
access_secret = os.environ['TWITTER_ACCESS_SECRET']

class EchoStreamListener(tweepy.StreamListener):

\end{verbatim}
```python
def __init__(self, api, dump_json=False, numtweets=0):
    self.api = api
    self.dump_json = dump_json
    self.count = 0
    self.limit = int(numtweets)
    super(tweepy.StreamListener, self).__init__()

def on_data(self, tweet):
    tweet_data = json.loads(tweet)
    if 'text' in tweet_data:
        if self.dump_json:
            print tweet.rstrip()
        else:
            print tweet_data['text'].encode("utf-8").rstrip()

        self.count = self.count+1
        return False if self.count == self.limit else True

def on_error(self, status_code):
    return True

def on_timeout(self):
    return True

if __name__ == '__main__':
    parser = get_parser()
    args = parser.parse_args()

    auth = tweepy.OAuthHandler(consumer_key, consumer_secret)
    auth.set_access_token(access_key, access_secret)
    api = tweepy.API(auth)
    sapi = tweepy.streaming.Stream(
        auth, EchoStreamListener(
            api=api,
            dump_json=args.json,
            numtweets=args.numtweets))

    sapi.sample()
```

First, we import three dependencies: `tweepy`, and the `os` and `json` modules, which come with the Python interpreter version 2.6 or greater.

We then define a class, `EchoStreamListener`, that inherits and extends `StreamListener` from `tweepy`. As the name suggests, `StreamListener` listens for events and tweets being published on the real-time stream and performs actions accordingly.
Whenever a new event is detected, it triggers a call to `on_data()`. In this method, we extract the text field from a tweet object and print it to standard output with UTF-8 encoding. Alternatively, if the script is invoked with `-j`, we print the whole JSON tweet. When the script is executed, we instantiate a `tweepy.OAuthHandler` object with the OAuth credentials that identify our Twitter account, and then we use this object to authenticate with the application access and secret key. We then use the `auth` object to create an instance of the `tweepy.API` class (`api`)

Upon successful authentication, we tell Python to listen for events on the real-time stream using `EchoStreamListener`.

An http GET request to the `statuses/sample` endpoint is performed by `sample()`. The request returns a random sample of all public statuses.

Beware! By default, `sample()` will run indefinitely. Remember to explicitly terminate the method call by pressing `Ctrl + C`.

**Summary**

This chapter gave a whirlwind tour of where Hadoop came from, its evolution, and why the version 2 release is such a major milestone. We also described the emerging market in Hadoop distributions and how we will use a combination of local and cloud distributions in the book.

Finally, we described how to set up the needed software, accounts, and environments required in subsequent chapters and demonstrated how to pull data from the Twitter stream that we will use for examples.

With this background out of the way, we will now move on to a detailed examination of the storage layer within Hadoop.
Where to buy this book
You can buy Learning Hadoop 2 from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.
Click here for ordering and shipping details.