Addison Wesley Data & Analytics Series

Practical **Data Science** with Hadoop® and Spark

Designing and Building **Effective Analytics** at Scale

Ofer Mendelevitch
Casey Stella
Douglas Eadline

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

Practical Data Science with Hadoop® and Spark

Practical Data Science with Hadoop® and Spark

Designing and Building Effective Analytics at Scale

Ofer Mendelevitch
Casey Stella
Douglas Eadline

Addison-Wesley

Boston • Columbus • Indianapolis • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and the publisher was aware of a trademark claim, the designations have been printed with initial capital letters or in all capitals.

The authors and publisher have taken care in the preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2016955465

Copyright © 2017 Pearson Education, Inc.

All rights reserved. Printed in the United States of America. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. For information regarding permissions, request forms and the appropriate contacts within the Pearson Education Global Rights & Permissions Department, please visit www.pearsoned.com/permissions/.

ISBN-13: 978-0-13-402414-1
ISBN-10: 0-13-402414-1

1 16

Contents

This page intentionally left blank

Foreword

Hadoop and data science have been sought after skillsets respectively over the last five years. However, few publications have attempted to bring the two together, teaching data science within the Hadoop context. For practitioners looking for an introduction to data science combined with solving those problems at scale using Hadoop and related tools, this book will prove to be an excellent resource.

The topic of data science is introduced with topics covered including data ingest, munging, feature extraction, machine learning, predictive modeling, anomaly detection, and natural language processing. The platform of choice for the examples and implementation of these topics is Hadoop, Spark, and the other parts of the Hadoop ecosystem. Its coverage is broad, with specific examples keeping the book grounded in an engineer's need to solve real-world problems. For those already familiar with data science, but looking to expand their skillsets to very large datasets and Hadoop, this book is a great introduction.

Throughout the text it focuses on concrete examples and providing insight into business value with each approach. Chapter 5, "Data Munging with Hadoop," provides particularly useful real-world examples on using Hadoop to prepare large datasets for common machine learning and data science tasks. Chapter 10 on anomaly detection is particularly useful for large datasets where monitoring and alerting are important. Chapter 11 on natural language processing will be of interest to those attempting to make chatbots.

Ofer Mendelevitch is the VP of Data Science at Lendup.com and was previously the Director of Data Science at Hortonworks. Few others are as qualified to be the lead author on a book combining data science and Hadoop. Joining Ofer is his former colleague, Casey Stella, a Principal Data Scientist at Hortonworks. Rounding out these experts in data science and Hadoop is Doug Eadline, frequent contributor to the Addison-Wesley Data & Analytics Series with the titles *Hadoop Fundamentals Live Lessons*, *Apache Hadoop 2 Quick-Start Guide*, and *Apache Hadoop YARN*. Collectively, this team of authors brings over a decade of Hadoop experience. I can imagine few others that have as much knowledge on the subject of data science and Hadoop.

I'm excited to have this addition to the Data & Analytics Series. Creating data science solutions at scale in production systems is an in-demand skillset. This book will help you come up to speed quickly to deploy and run production data science solutions at scale.

—*Paul Dix*
Series Editor

Preface

Data science and machine learning are at the core of many innovative technologies and products and are expected to continue to disrupt many industries and business models across the globe for the foreseeable future. Until recently though, most of this innovation was constrained by the limited availability of data.

With the introduction of Apache Hadoop, all of that has changed. Hadoop provides a platform for storing, managing, and processing large datasets inexpensively and at scale, making data science analysis of large datasets practical and feasible. In this new world of large-scale advanced analytics, data science is a core competency that enables organizations to remain competitive and innovate beyond their traditional business models. During our time at Hortonworks, we have had a chance to see how various organizations tackle this new set of opportunities and help them on their journey to implementing data science at scale with Hadoop and Spark. In this book we would like to share some of this learning and experiences.

Another issue we also wish to emphasize is the evolution of Apache Hadoop from its early incarnation as a monolithic MapReduce engine (Hadoop version 1) to a versatile data analytics platform that runs on YARN and supports not only MapReduce but also Tez and Spark as processing engines (Hadoop version 2). The current version of Hadoop provides a robust and efficient platform for many data science applications and opens up a universe of opportunities to new business use cases that were previously unthinkable.

Focus of the Book

This book focuses on real-world practical aspects of data science with Hadoop and Spark. Since the scope of data science is very broad, and every topic therein is deep and complex, it is quite difficult to cover the topic thoroughly. We approached this problem by attempting a good balance between the theoretical coverage of each use case and the example-driven treatment of practical implementation.

This book is not designed to dig deep into many of the mathematical details of each machine learning or statistical approach but rather provide a high-level description of the main concepts along with guidelines for its practical use in the context of the business problem. We provide some references that offer more in-depth treatment of the mathematical details of these techniques in the text and have compiled a list of relevant resources in Appendix C, "Additional Background on Data Science and Apache Hadoop and Spark."

When learning about Hadoop, access to a Hadoop cluster environment can become an issue. Finding an effective way to "play" with Hadoop and Spark can be challenging for some individuals. At a minimum, we recommend the Hortonworks virtual machine sandbox for those that would like an easy way to get started with Hadoop. The sandbox is a full single-node Hadoop installation running inside a virtual machine. The virtual machine can be run under Windows, Mac OS, and Linux. Please see http://hortonworks.com/products/sandbox for more information on how to download and install the sandbox. For further help with Hadoop we recommend *Hadoop 2 Quick-Start Guide: Learn the Essentials of Big Data Computation in the Apache Hadoop 2 Ecosystem* (and supporting videos), all mentioned in Appendix C.

Who Should Read This Book

This book is intended for those readers who are interested to learn more about what data science is and some of the practical considerations of its application to large-scale datasets. It provides a strong technical foundation for readers who want to learn more about how to implement various use cases, the tools that are best suited for the job, and some of the architectures that are common in these situations. It also provides a business-driven viewpoint on when application of data science to large datasets is useful to help stakeholders understand what value can be derived for their organization and where to invest their resources in applying large-scale machine learning.

There is also a level of experience assumed for this book. For those not versed in data science, some basic competencies are important to have to understand the different methods, including statistical concepts (for example, mean and standard deviation), and a bit of background in programming (mostly Python and a bit of Java or Scala) to understand the examples throughout the book.

For those with a data science background, you should generally be comfortable with the material, although there may be some practical issues such as understanding the numerous Apache projects. In addition, all examples are text-based, and some familiarity with the Linux command line is required. It should be noted that we did not use (or test) a Windows environment for the examples. However, there is no reason to assume they will not work in that and other environments (Hortonworks supports Windows).

In terms of a specific Hadoop environment, all the examples and code were run under Hortonworks HDP Linux Hadoop distribution (either laptop or cluster). Your environment may differ in terms of distribution (Cloudera, MapR, Apache Source) or operating systems (Windows). However, all the tools (or equivalents) are available in both environments.

How to Use This Book

We anticipate several different audiences for the book:

- data scientists
- developers/data engineers
- business stakeholders

While these readers come at the Hadoop analytics from different backgrounds, their goal is certainly the same—running data analytics with Hadoop and Spark at scale. To this end, we have designed the chapters to meet the needs of all readers, and as such readers may find that they can skip areas where they may have a good practical understanding. Finally, we also want to invite novice readers to use this book as a first step in their understanding of data science at scale. We believe there is value in "walking" through the examples, even if you are not sure what is actually happening, and then going back and buttressing your understanding with the background material.

Part I, "Data Science with Hadoop—An Overview," spans the first three chapters.

Chapter 1, "Introduction to Data Science," provides an overview of data science and its history and evolution over the years. It lays out the journey people often take to become a data scientist. For those not versed in data science, this chapter will help you understand why it has evolved into a powerful discipline and provide some insight into how a data scientist designs and refines projects. There is also some discussion about what makes a data scientist and how to best plan your career in that direction.

Chapter 2, "Use Cases for Data Science," provides a good overview of how business use cases are impacted by the volume, variety, and velocity of modern data streams. It also covers some real-world data science use cases in order to help you gain an understanding of its benefits in various industries and applications.

Chapter 3, "Hadoop and Data Science," provides a quick overview of Hadoop, its evolution over the years, and the various tools in the Hadoop ecosystem. For first-time Hadoop users this chapter can be a bit overwhelming. There are many new concepts introduced including the Hadoop file system (HDFS), MapReduce, the Hadoop resource manager (YARN), and Spark. While the number of sub-projects (and weird names) that make up the Hadoop ecosystem may seem daunting, not every project is used at the same time, and the applications in the later chapters usually focus on only a few tools at a time.

Part II, "Preparing and Visualizing Data with Hadoop," includes the next three chapters.

Chapter 4, "Getting Data into Hadoop," focuses on data ingestion, discussing various tools and techniques to import datasets from external sources into Hadoop. It is useful for many subsequent chapters. We begin with describing the Hadoop data lake concept and then move into the various ways data can be used by the Hadoop platform. The ingestion targets two of the more popular Hadoop tools—Hive and Spark. This chapter focuses on code and hands-on solutions—if you are new to Hadoop, its best to also consult Appendix B, "HDFS Quick Start," to get you up to speed on the HDFS file system.

Chapter 5, "Data Munging with Hadoop," focuses on data munging with Hadoop or how to identify and handle data quality issues, as well as pre-process data and prepare it for modeling. We introduce the concepts of data completeness, validity, consistency, timeliness, and accuracy. Examples of feature generation using a real data set are provided. This chapter is useful for all types of subsequent analysis and, like Chapter 4, is a precursor to many of the techniques mentioned in later chapters.

An important tool in the process of data munging is visualization. Chapter 6, "Exploring and Visualizing Data," discusses what it means to do visualization with big data. As background, this chapter is useful for reinforcing some of the basic concepts behind data visualization. The charts presented in the chapter were generated using R. Source code for all the plots is available so readers can try these charts with their own data.

Part III, "Applying Data Modeling with Hadoop," encompasses the final six chapters.

Chapter 7, "Machine Learning with Hadoop," provides an overview of machine learning at a high level, covering the main tasks in machine learning such as classification and regression, clustering, and anomaly detection. For each task type, we explore the problem and the main approaches to solutions.

Chapter 8, "Predictive Modeling," covers the basic algorithms and various Hadoop tools for predictive modeling. The chapter includes an end-to-end example of building a predictive model for sentiment analysis of Twitter text using Hive and Spark.

Chapter 9, "Clustering," dives into cluster analysis, a very common technique in data science. It provides an overview of various clustering techniques and similarity functions, which are at the core of clustering. It then demonstrates a real-world example of using topic modeling on a large corpus of documents using Hadoop and Spark.

Chapter 10, "Anomaly Detection with Hadoop," covers anomaly detection, describing various types of approaches and algorithms as well as how to perform large-scale anomaly detection on various datasets. It then demonstrates how to build an anomaly detection system with Spark for the KDD99 dataset.

Chapter 11, "Natural Language Processing," covers applications of data science to the specific area of human language, using a set of techniques commonly called natural language processing (NLP). It discusses various approaches to NLP, open-source tools that are effective at various NLP tasks, and how to apply NLP to large-scale corpuses using Hadoop, Pig, and Spark. An end-to-end example shows an advanced approach to sentiment analysis that uses NLP at scale with Spark.

Chapter 12, "Data Science with Hadoop—The Next Frontier," discusses the future of data science with Hadoop, covering advanced data discovery techniques and deep learning.

Consult Appendix A, "Book Webpage and Code Download," for the book web page and code repository (the web page provides a question and answer forum). Appendix B, as mentioned previously, provides a quick overview of HDFS for new users and the aforementioned Appendix C provides further references and background on Hadoop, Spark, HDFS, machine learning, and many other topics.

Book Conventions

Code and file references are displayed in a `monospaced` font. Code input lines that wrap because they are too long to fit on one line in this book are denoted with this symbol ➥ at the start of the next line. Long output lines are wrapped at page boundaries without the symbol.

Accompanying Code

Again, please see Appendix A, "Book Web Page and Code Download," for the location of all code used in this book.

Register your copy of *Practical Data Science with Hadoop® and Spark* at informit.com for convenient access to downloads, updates, and corrections as they become available. To start the registration process, go to informit.com/register and log in or create an account. Enter the product ISBN (9780134024141) and click Submit. Once the process is complete, you will find any available bonus content under "Registered Products."

This page intentionally left blank

Acknowledgments

Some of the figures and examples were inspired and copied from Yahoo! (yahoo.com), the Apache Software Foundation (http://www.apache.org), and Hortonworks (http://hortonworks.com). Any copied items either had permission from the author or were available under an open sharing license.

Many people have worked behind the scenes to make this book possible. Thank you to the reviewers who took the time to carefully read the rough drafts: Fabricio Cannini, Brian D. Davison, Mark Fenner, Sylvain Jaume, Joshua Mora, Wendell Smith, and John Wilson.

Ofer Mendelevitch

I want to thank Jeff Needham and Ron Lee who encouraged me to start this book, many others at Hortonworks who helped with constructive feedback and advice, John Wilson who provided great constructive feedback and industry perspective, and of course Debra Williams Cauley for her vision and support in making this book a reality. Last but not least, this book would not have come to life without the loving support of my beautiful wife, Noa, who encouraged and supported me every step of the way, and my boys, Daniel and Jordan, who make all this hard work so worthwhile.

Casey Stella

I want to thank my patient and loving wife, Leah, and children, William and Sylvia, without whom I would not have the time to dedicate to such a time-consuming and rewarding venture. I want to thank my mother and grandmother, who instilled a love of learning that has guided me to this day. I want to thank the taxpayers of the State of Louisiana for providing a college education and access to libraries, public radio, and television; without which I would have neither the capability, the content, nor the courage to speak. Finally, I want to thank Debra Williams Cauley at Addison-Wesley who used the carrot far more than the stick.

Douglas Eadline

To Debra Williams Cauley at Addison-Wesley, your kind efforts and office at the GCT Oyster Bar made the book-writing process almost easy (again!). Thanks to my support crew, Emily, Carla, and Taylor—yet another book you know nothing about. Of course, I cannot forget my office mate, Marlee, and those two boys. And, finally, another big thank you to my wonderful wife, Maddy, for her constant support.

This page intentionally left blank

About the Authors

Ofer Mendelevitch is Vice President of Data Science at Lendup, where he is responsible for Lendup's machine learning and advanced analytics group. Prior to joining Lendup, Ofer was Director of Data Science at Hortonworks, where he was responsible for helping Hortonwork's customers apply Data Science with Hadoop and Spark to big data across various industries including healthcare, finance, retail, and others. Before Hortonworks, Ofer served as Entrepreneur in Residence at XSeed Capital, Vice President of Engineering at Nor1, and Director of Engineering at Yahoo!.

Casey Stella is a Principal Data Scientist at Hortonworks, which provides an open source Hadoop distribution. Casey's primary responsibility is leading the analytics/data science team for the Apache Metron (Incubating) Project, an open source cybersecurity project. Prior to Hortonworks, Casey was an architect at Explorys, which was a medical informatics startup spun out of the Cleveland Clinic. In the more distant past, Casey served as a developer at Oracle, Research Geophysicist at ION Geophysical, and as a poor graduate student in Mathematics at Texas A&M.

Douglas Eadline, PhD, began his career as an analytical chemist with an interest in computer methods. Starting with the first Beowulf how-to document, Doug has written hundreds of articles, white papers, and instructional documents covering many aspects of HPC and Hadoop computing. Prior to starting and editing the popular ClusterMonkey.net website in 2005, he served as editor in chief for *ClusterWorld Magazine* and was senior HPC editor for *Linux Magazine*. He has practical hands-on experience in many aspects of HPC and Apache Hadoop, including hardware and software design, benchmarking, storage, GPU, cloud computing, and parallel computing. Currently, he is a writer and consultant to the HPC/analytics industry and leader of the Limulus Personal Cluster Project (http://limulus.basement-supercomputing.com). He is author of the *Hadoop Fundamentals LiveLessons* and *Apache Hadoop YARN Fundamentals LiveLessons* videos from Pearson, and is co-author of *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2* and author *of Hadoop 2 Quick Start Guide: Learn the Essentials of Big Data Computing in the Apache Hadoop 2 Ecosystem*, also from Addison-Wesley, and *High Performance Computing for Dummies.*

II

Preparing and Visualizing Data with Hadoop

This page intentionally left blank

4

Getting Data into Hadoop

You can have data without information,
but you cannot have information without data.

Daniel Keys Moran

In This Chapter:

- The data lake concept is presented as a new data processing paradigm.
- Basic methods for importing CSV data into HDFS and Hive tables are presented.
- Additional methods for using Spark to import data into Hive tables or directly for a Spark job are presented.
- Apache Sqoop is introduced as a tool for exporting and importing relational data into and out of HDFS.
- Apache Flume is introduced as a tool for transporting and capturing streaming data (e.g., web logs) into HDFS.
- Apache Oozie is introduced as workflow manager for Hadoop ingestion jobs.
- The Apache Falcon project is described as a framework for data governance (organization) on Hadoop clusters.

No matter what kind of data needs processing, there is often a tool for importing such data from or exporting such data into the Hadoop Distributed File System (HDFS). Once stored in HDFS the data may be processed by any number of tools available in the Hadoop ecosystem.

This chapter begins with the concept of the Hadoop data lake and then follows with a general overview of each of the main tools for data ingestion into Hadoop —Spark, Sqoop, and Flume—along with some specific usage examples. Workflow tools such as Oozie and Falcon are presented as tools that aid in managing the ingestion process.

Hadoop as a Data Lake

Data is ubiquitous, but that does not always mean that it's easy to store and access. In fact, many existing pre-Hadoop data architectures tend to be rather strict and therefore difficult to work with and make changes to. The data lake concept changes all that.

So what is a data lake?

With the more traditional database or data warehouse approach, adding data to the database requires data to be transformed into a *pre-determined* schema before it can be loaded into the database. This step is often called "extract, transform, and load" (ETL) and often consumes a lot of time, effort, and expense before the data can be used for downstream applications. More importantly, decisions about how the data will be used must be made during the ETL step, and later changes are costly. In addition, data are often discarded in the ETL step because they do not fit into the data schema or are deemed un-needed or not valuable for downstream applications.

One of the basic features of Hadoop is a central storage space for all data in the Hadoop Distributed File Systems (HDFS), which make possible inexpensive and redundant storage of large datasets at a much lower cost than traditional systems.

This enables the Hadoop data lake approach, wherein all data are often stored in raw format, and what looks like the ETL step is performed when the data are processed by Hadoop applications. This approach, also known as schema on read, enables programmers and users to enforce a structure to suit their needs when they access data. The traditional data warehouse approach, also known as schema on write, requires more upfront design and assumptions about how the data will eventually be used.

For data science purposes, the capability to keep all the data in raw format is extremely beneficial since often it is not clear up front which data items may be valuable to a given data science goal.

With respect to big data, the data lake offers three advantages over a more traditional approach:

- All data are **available**. There is no need to make any assumptions about future data use.
- All data are **sharable**. Multiple business units or researchers can use all available data[1], some of which may not have been previously available due to data compartmentalization on disparate systems.
- All **access methods** are available. Any processing engine (MapReduce, Tez, Spark) or application (Hive, Spark-SQL, Pig) can be used to examine the data and process it as needed.

1. The capability to use all available data is, of course, governed, as you might expect, by the appropriate security policy with Hadoop tools such as Apache Ranger. The point here is that there is no technical hurdle to data sharing, as is often the case with traditional data architectures.

To be clear, data warehouses are valuable business tools, and Hadoop is designed to complement them, not replace them. Nonetheless, the traditional data warehouse technology was developed before the data lake began to fill with such large quantities of data. The growth of new data from disparate sources including social media, click streams, sensor data, and others is such that we are starting to quickly fill the data lake. Traditional ETL stages may not be able to keep up with the rate at which data are entering the lake. There will be overlap, and each tool will address the need for which it was designed.

The difference between a traditional data warehouse and Hadoop is depicted in Figure 4.1.

Different data sources (A, B, C) can be seen entering either an ETL process or a data lake. The ETL process places the data in a schema as it stores (writes) the data to the relational database. The data lake stores the data in raw form. When a Hadoop application

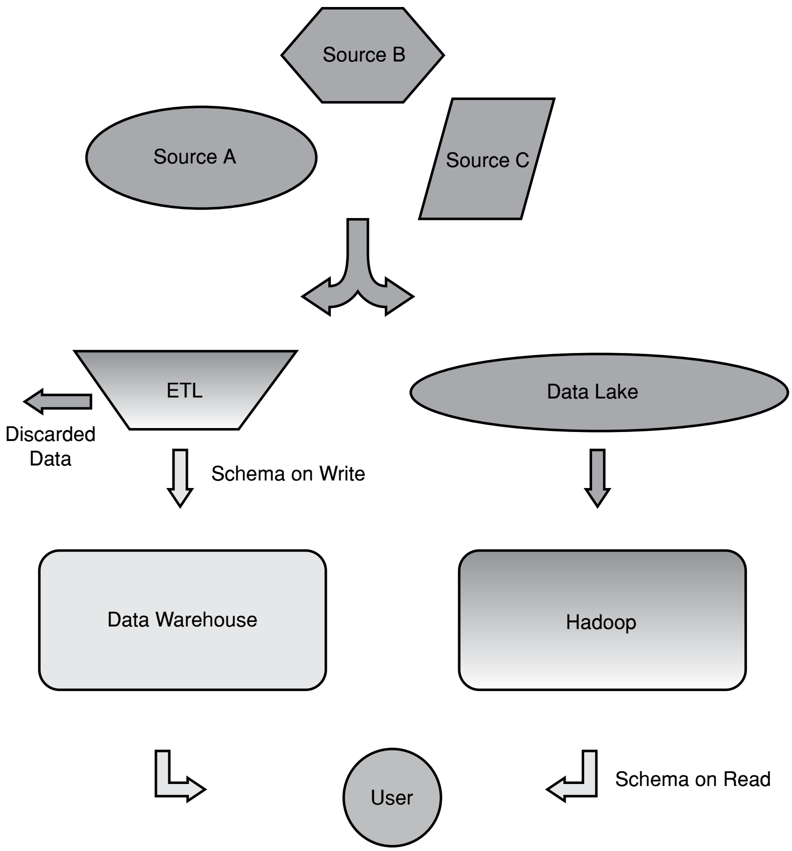

Figure 4.1 The data warehouse versus the Hadoop data lake.

uses the data, the schema is applied to data as they are read from the lake. Note that the ETL step often discards some data as part of the process. In both cases the user accesses the data they need. However, in the Hadoop case it can happen as soon as the data are available in the lake.

The Hadoop Distributed File System (HDFS)

Virtually all Hadoop applications operate on data that are stored in HDFS. The operation of HDFS is separate from the local file system that most users are accustomed to using. That is, the user must explicitly copy to and from the HDFS file system. HDFS is not a general file system and as such cannot be used as a substitute for existing POSIX (or even POSIX-like) file systems.

In general, HDFS is a specialized streaming file system that is optimized for reading and writing of large files. When writing to HDFS, data are "sliced" and replicated across the servers in a Hadoop cluster. The slicing process creates many small sub-units (blocks) of the larger file and *transparently* writes them to the cluster nodes. The various slices can be processed in parallel (at the same time) enabling faster computation. The user does not see the file slices but interacts with whole files in HDFS like a normal file system (i.e., files can be moved, copied, deleted, etc.). When transferring files out of HDFS, the slices are assembled and written as one file on the host file system.

The slices or sub-units are also replicated across different servers so that the failure of any single server will not result in lost data. Due to its design, HDFS does not support random reads or writes to files but does support appending a file. Note that for testing purposes it is also possible to create a single instance of HDFS on a single hard drive (i.e., a laptop or desktop computer), and in this situation there is no file slicing or replication performed on the file.

Direct File Transfer to Hadoop HDFS

The easiest way to move data into and out of HDFS is to use the native HDFS commands. These commands are wrappers that interact with the HDFS file system. Local commands, such as `cp`, `ls`, or `mv` will only work on local files. To copy a file (`test`) from your local file system to HDFS, the following `put` command can be used:

```
$ hdfs dfs -put test
```

To view files in HDFS use the following command. The result is a full listing similar to a locally executed `ls -l` command:

```
$ hdfs dfs -ls
-rw-r--r--   2 username hdfs        497 2016-05-11 14:32 test
```

To copy a file (`another-test`) from HDFS to your local file system, use the following `get` command:

```
$ hdfs dfs -get another-test
```

Other HDFS commands will be introduced in the examples. Appendix B "HDFS Quick Start," provides basic command examples including listing, copying, and removing files in HDFS.

Importing Data from Files into Hive Tables

Apache Hive is an SQL-like tool for analyzing data in HDFS. Data scientists often want to import data into Hive from existing text-based files exported from spreadsheets or databases. These file formats often include tab-separated values (TSV), comma-separated values (CSV), raw text, JSON, and others. Having the data in Hive tables enables easy access to it for subsequent modeling steps, the most common of which is feature generation, which we discuss in Chapter 5, "Data Munging with Hadoop."

Once data are imported and present as a Hive table, it is available for processing using a variety of tools including Hive's SQL query processing, Pig, or Spark.

Hive supports two types of tables. The first type of table is an *internal table* and is fully managed by Hive. If you delete an internal table, both the definition in Hive *and* the data will be deleted. Internal tables are stored in an optimized format such as ORC and thus provide a performance benefit. The second type of table is an *external table* that is not managed by Hive. External tables use only a metadata description to access the data in its raw form. If you delete an external table, only the definition (metadata about the table) in Hive is deleted and the actual data remain intact. External tables are often used when the data resides outside of Hive (i.e., some other application is also using/creating/managing the files), or the original data need to remain in the underlying location even after the table is deleted.

Due to the large number of use cases, we do not cover all the input methods available to Hive, and instead just a basic example of CSV file import is described. Interested readers can consult the Hive project page, https://hive.apache.org, for more information.

Import CSV Files into Hive Tables

The following example illustrates how a comma delimited text file (CSV file) can be imported into a Hive table. The input file (names.csv) has five fields (Employee ID, First Name, Title, State, and type of Laptop). The first five lines of the file are as follows:

```
10,Andrew,Manager,DE,PC
11,Arun,Manager,NJ,PC
12,Harish,Sales,NJ,MAC
13,Robert,Manager,PA,MAC
14,Laura,Engineer,PA,MAC
```

The first input step is to create a directory in HDFS to hold the file. Note that, like most Hadoop tools, Hive input is directory-based. That is, input for an operation is taken as all files in a given directory. The following command creates a `names` directory in the users HDFS directory.

```
$ hdfs dfs -mkdir names
```

In this example, one file is used. However, any number of files could be placed in the input directory. Next the names.csv file is moved into the HDFS names directory.

```
$ hdfs dfs -put name.csv names
```

Once the file is in HDFS, we first load the data as an external Hive table. Start a Hive shell by typing `hive` at the command prompt and enter the following commands. Note, to cut down on clutter, some of the non-essential Hive output (run times, progress bars, etc.) have been removed from the Hive output.

```
hive> CREATE EXTERNAL TABLE IF NOT EXISTS Names_text(
    > EmployeeID INT,FirstName STRING, Title STRING,
    > State STRING, Laptop STRING)
    > COMMENT 'Employee Names'
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY ','
    > STORED AS TEXTFILE
    > LOCATION '/user/username/names';
OK
```

If the command worked, an `OK` will be printed. The various fields and the comma delimiter are declared in the command. The final `LOCATION` statement in the command tells Hive where to find the input files. The import can be verified by listing the first five rows in the table:

```
hive> Select * from Names_text limit 5;
OK
10      Andrew  Manager DE      PC
11      Arun    Manager NJ      PC
12      Harish  Sales   NJ      MAC
13      Robert  Manager PA      MAC
14      Laura   Engineer PA     MAC
```

The next step is to move the external table to an internal Hive table. The internal table must be created using a similar command. However, the `STORED AS` format offers new options. There are four main file formats for Hive tables in addition to the basic text format. The choice of format depends on the type of data and analysis, but in most cases either ORC or Parquet are used as they provide the best compression and speed advantages for most data types.

- Text file—All data are stored as raw text using the Unicode standard.
- Sequence file—The data are stored as binary key/value pairs.
- RCFile—All data are stored in a column optimized format (instead of row optimized).
- ORC—An optimized row columnar format that can significantly improve Hive performance.
- Parquet—A columnar format that provides portability to other Hadoop tools including Hive, Drill, Impala, Crunch, and Pig.

The following command creates an internal Hive table that uses the ORC format:

```
hive> CREATE TABLE IF NOT EXISTS Names(
    > EmployeeID INT,FirstName STRING, Title STRING,
    > State STRING, Laptop STRING)
    > COMMENT 'Employee Names'
    > STORED AS ORC;
OK
```

To create a table using one of the other formats, change the `STORED AS` command to reflect the new format. Once the table is created, the data from the external table can be moved to the internal table using the command,

```
hive> INSERT OVERWRITE TABLE Names SELECT * FROM Names_text;
```

As with the external table, the contents can be verified using the following command:

```
hive> Select * from Names limit 5;
OK
10       Andrew   Manager  DE       PC
11       Arun     Manager  NJ       PC
12       Harish   Sales    NJ       MAC
13       Robert   Manager  PA       MAC
14       Laura    Engineer PA       MAC
```

Hive also supports partitions. With partitions, tables can be separated into logical parts that make it more efficient to query a portion of the data. For example, the internal Hive table created previously can also be created with a partition based on the state field. The following command creates a partitioned table:

```
hive> CREATE TABLE IF NOT EXISTS Names_part(
    > EmployeeID INT,
    > FirstName STRING,
    > Title STRING,
    > Laptop STRING)
    > COMMENT 'Employee names partitioned by state'
    > PARTITIONED BY (State STRING)
    > STORED AS ORC;
OK
```

To fill the internal table from the external table for those employed from PA, the following command can be used:

```
hive> INSERT INTO TABLE Names_part PARTITION(state='PA')
    > SELECT EmployeeID, FirstName, Title, Laptop FROM Names_text WHERE
➥ state='PA';
...
OK
```

This method requires each partition key to be selected and loaded individually. When the number of potential partitions is large, this can make data entry inconvenient. To address this issue Hive now supports **dynamic-partition insert** (or multi-partition insert) that is designed to solve this problem by dynamically determining which partitions should be created and populated while scanning the input table.

Importing Data into Hive Tables Using Spark

Apache Spark is a modern processing engine that is focused on in-memory processing. Spark's primary data abstraction is an immutable distributed collection of items called a resilient distributed dataset (RDD). RDDs can be created from Hadoop input formats (such as HDFS files) or by transforming other RDDs. Each dataset in an RDD is divided into logical partitions, which may be transparently computed on different nodes of the cluster.

The other important data abstraction is Spark's DataFrame. A DataFrame is built on top of an RDD, but data are organized into named columns similar to a relational database table and similar to a data frame in R or in Python's Pandas package.

Spark DataFrames can be created from different data sources such as the following:

- Existing RDDs
- Structured data files
- JSON datasets
- Hive tables
- External databases

Due to its flexibility and friendly developer API, Spark is often used as part of the process of ingesting data into Hadoop. With Spark, you can read data from a CSV file, external SQL or NO-SQL data store, or another data source, apply certain transformations to the data, and store it onto Hadoop in HDFS or Hive. Similar to the Hive examples, a full treatment of all Spark import scenarios is beyond the scope of this book. Consult the Apache Spark project page, http://spark.apache.org, for more information.

The following sections provide some basic usage examples of data import using PySpark (Spark via the Python API), although these steps can also be performed using the Scala or Java interfaces to Spark. Each step is explained. However, a full description of the Spark commands and API are beyond the scope of this book.

All the examples assume the PySpark shell (version 1.6) has been started using the following command:

```
$ pyspark
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.2
      /_/
```

```
Using Python version 2.7.9 (default, Apr 14 2015 12:54:25)
SparkContext available as sc, HiveContext available as sqlContext.
>>>
```

Import CSV Files into HIVE Using Spark

Comma-separated value (CSV) files and, by extension, other text files with separators can be imported into a Spark DataFrame and then stored as a HIVE table using the steps described. Note that in this example we show how to use an RDD, translate it into a DataFrame, and store it in HIVE. It is also possible to load CSV files directly into DataFrames using the spark-csv package.

1. The first step imports functions necessary for Spark DataFrame operations:

   ```
   >>> from pyspark.sql import HiveContext
   >>> from pyspark.sql.types import *
   >>> from pyspark.sql import Row
   ```

2. Next, the raw data are imported into a Spark RDD. The input file, names.csv, is located in the users local file system and does not have to be moved into HDFS prior to use. (Assuming the local path to the data is /home/username.)

   ```
   >>> csv_data = sc.textFile("file:///home/username/names.csv")
   ```

3. The RDD can be confirmed by using the `type()` command:

   ```
   >>> type(csv_data)
   <class 'pyspark.rdd.RDD'>
   ```

4. The comma-separated data are then split using Spark's map( ) function that creates a new RDD:

   ```
   >>> csv_data  = csv_data.map(lambda p: p.split(","))
   ```

   Most CSV files have a header with the column names. The following steps remove this from the RDD,

   ```
   >>> header = csv_data.first()
   >>> csv_data = csv_data.filter(lambda p:p != header)
   ```

5. The data in the `csv_data` RDD are put into a Spark SQL DataFrame using the `toDF()` function. First, however, the data are mapped using the map() function so that every RDD item becomes a Row object which represents a row in the new DataFrame. Note the use of the `int()` to cast for the employee ID as an integer. All other columns default to a string type.

   ```
   >>> df_csv = csv_data.map(lambda p: Row(EmployeeID = int(p[0]),
   ➥FirstName = p[1], Title=p[2], State=p[3], Laptop=p[4])).toDF()
   ```

   The `Row()` class captures the mapping of the single values into named columns in a row and subsequently transforms the complete data into a DataFrame.

6. The structure and data of the first five rows of the `df_csv` DataFrame are viewed using the following command:

```
>>> df_csv.show(5)
+----------+---------+------+-----+--------+
|EmployeeID|FirstName|Laptop|State|   Title|
+----------+---------+------+-----+--------+
|        10|   Andrew|    PC|   DE| Manager|
|        11|     Arun|    PC|   NJ| Manager|
|        12|   Harish|   MAC|   NJ|   Sales|
|        13|   Robert|   MAC|   PA| Manager|
|        14|    Laura|   MAC|   PA|Engineer|
+----------+---------+------+-----+--------+
only showing top 5 rows
```

7. Similarly, if you'd like to inspect the DataFrame schema, use the `printSchema()` command:

```
>>> df_csv.printSchema()
root
 |-- EmployeeID: long (nullable = true)
 |-- FirstName: string (nullable = true)
 |-- Laptop: string (nullable = true)
 |-- State: string (nullable = true)
 |-- Title: string (nullable = true)
```

8. Finally, to store the DataFrame into a Hive table, use `saveAsTable()`:

```
>>> from pyspark.sql import HiveContext
>>> hc = HiveContext(sc)
>>> df_csv.write.format("orc").saveAsTable("employees")
```

Here we create a HiveContext that is used to store the DataFrame into a Hive table (in ORC format), by using the `saveAsTable()` command.

Import a JSON File into HIVE Using Spark

Spark can import JSON files directly into a DataFrame. The following is a JSON formatted version of the names.csv file used in the previous examples. Note that by entering the `EmployeeID` as an un-quoted integer, it will be input as an integer.

```
{"EmployeeID":10,"FirstName":"Andrew","Title":"Manager","State":"DE",
➥ "Laptop":"PC"}
{"EmployeeID":11,"FirstName":"Arun","Title":"Manager","State":"NJ",
➥ "Laptop":"PC"}
{"EmployeeID":12,"FirstName":"Harish","Title":"Sales","State":"NJ",
➥ "Laptop":"MAC"}
```

Also note that Spark expects each line to be a separate JSON object, so it will fail if you try to load a fully formatted JSON file.

1. The first step imports the needed functions and creates a `HiveContext`.

   ```
   >>> from pyspark.sql import HiveContext
   >>> hc = HiveContext(sc)
   ```

   Similar to the CSV example, the data file is located in the users local file system.

   ```
   >>> df_json = hc.read.json("file:///home/username/names.json")
   ```

2. The first five rows of the DataFrame can be viewed using the `df_json.show(5)` command:

   ```
   >>> df_json.show(5)
   +----------+---------+------+-----+--------+
   |EmployeeID|FirstName|Laptop|State|   Title|
   +----------+---------+------+-----+--------+
   |        10|   Andrew|    PC|   DE| Manager|
   |        11|     Arun|    PC|   NJ| Manager|
   |        12|   Harish|   MAC|   NJ|   Sales|
   |        13|   Robert|   MAC|   PA| Manager|
   |        14|    Laura|   MAC|   PA|Engineer|
   +----------+---------+------+-----+--------+
   only showing top 5 rows
   ```

3. To confirm that the `EmployeeID` was indeed cast as an integer, the `df_json.printSchema()` command can be used to inspect the DataFrame schema:

   ```
   >>> df_json.printSchema()

   root
    |-- EmployeeID: long (nullable = true)
    |-- FirstName: string (nullable = true)
    |-- Laptop: string (nullable = true)
    |-- State: string (nullable = true)
    |-- Title: string (nullable = true)
   ```

4. Similar to the CSV example, storing this DataFrame back to Hive is simple:

   ```
   >>> df_json.write.format("orc").saveAsTable("employees")
   ```

Using Apache Sqoop to Acquire Relational Data

In many enterprise environments, a lot of data that is required for data science applications resides inside of database management systems such as Oracle, MySQL, PosgreSQL, or DB2. Before we can use this data in the context of a data science application, we need to ingest such data into Hadoop.

Sqoop is a tool designed to transfer data between Hadoop and relational databases. You can use Sqoop to import data from a relational database management system (RDBMS) into the Hadoop Distributed File System (HDFS) or export data from Hadoop back into an RDBMS.

Sqoop can be used with any JDBC-compliant database and has been tested on Microsoft SQL Server, PostgreSQL, MySQL, and Oracle. In the remainder of this section, a brief overview of how Sqoop works with Hadoop is provided. In addition, a basic Sqoop example walk-through is demonstrated. To fully explore Sqoop, more information can found by consulting the Sqoop project website at http://sqoop.apache.org.

Data Import and Export with Sqoop

Figure 4.2 describes the process of importing data into HDFS using Sqoop, which includes two steps. In the first step, Sqoop examines the database to gather the necessary metadata for the data that are to be imported. The second step is a map-only[2] (no reduce step) Hadoop job that Sqoop submits to the cluster. This is the job that does the actual data transfer using the metadata captured in the previous step. Note that each node doing the import must have access to the database.

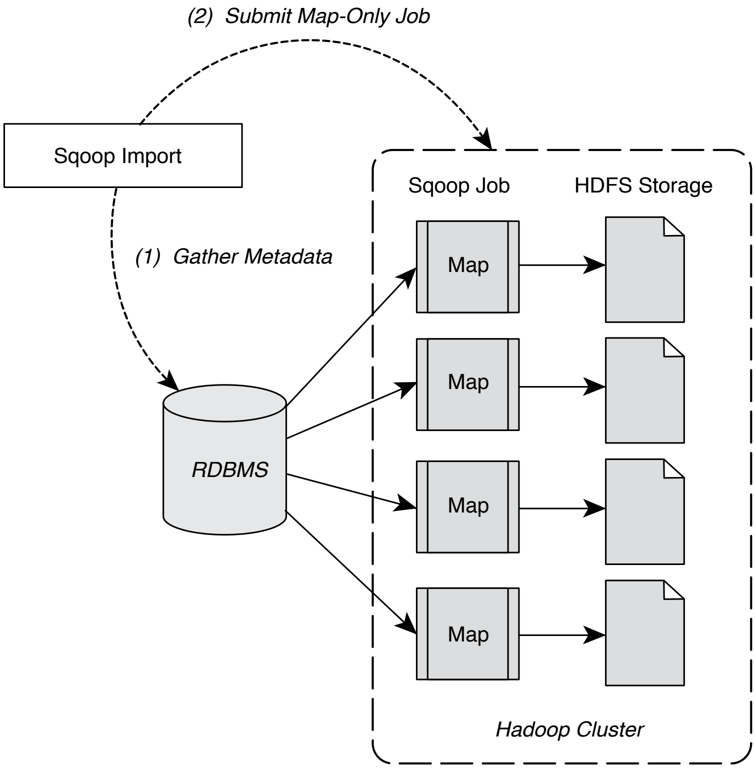

Figure 4.2 Two-step Apache Sqoop data import method.

2. A **map-only job** is a term used in the Hadoop ecosystem to refer to a map-reduce job that has some logic implemented in the map stage, and nothing (no-op) in the reduce job.

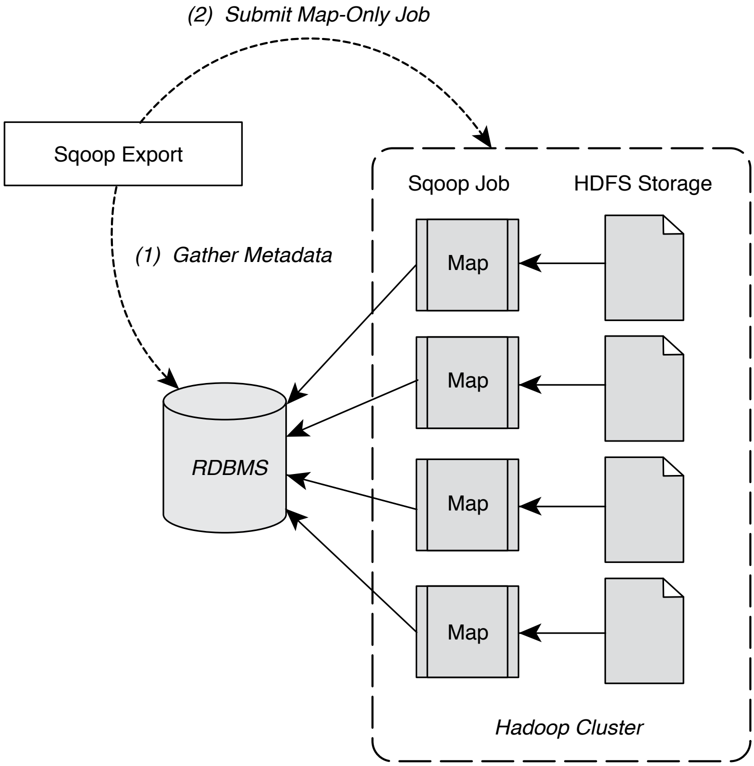

Figure 4.3 Two-step Sqoop data export method.

The imported data is saved in an HDFS directory. Sqoop will use the database name for the directory or the user can specify any alternative directory where the files should be populated. By default, these files contain comma-delimited fields, with new lines separating different records. You can easily override the format in which data is copied over by explicitly specifying the field separator and record terminator characters. Once placed in HDFS, the data are ready for further processing.

Data export from the cluster works in a similar fashion. The export is done in two steps as shown in Figure 4.3. Like the import process, the first step is to examine the database for metadata, followed by the export step that is again a map-only Hadoop job to write the data to the target database. Sqoop divides the input dataset into splits and then uses individual map tasks to push the splits to the database. Again, this process assumes the map tasks have access to the database.

Apache Sqoop Version Changes

Two versions of Sqoop are in general use within the Hadoop ecosystem. Many users have found the features removed in version 2 to be useful and continue to use the first version. Sqoop version 2 will be used for the examples.

Table 4.1 **Apache Sqoop version comparison.**

Feature	Sqoop Version 1	Sqoop Version 2
Connectors for all major RDBMS	Supported	Not supported. Use the generic JDBC Connector.
Kerberos Security Integration	Supported	Not supported
Data transfer from RDBMS to Hive or HBase	Supported	Not supported. First import data from RDBMS into HDFS, then load data into Hive or HBase manually.
Data transfer from Hive or HBase to RDBMS	Not supported. First export data from Hive or HBase into HDFS, and then use Sqoop for export.	Not supported. First export data from Hive or HBase into HDFS, and then use Sqoop for export.

Sqoop version 1 uses specialized connectors to access external database systems. These are often optimized for various RDBMS systems or those that do not support JDBC (Java Database Connectivity). Connectors are plug-in components based on Sqoop's extension framework and can be added to any existing Sqoop installation. Once a connector is installed, Sqoop can use it to efficiently transfer data between Hadoop and the external store supported by the connector. By default, Sqoop version 1 includes connectors for various popular databases such as MySQL, PostgreSQL, Oracle, SQL Server, and DB2. Sqoop version 1 also supports direct transfer to and from the RDBMS for HBase or Hive.

In order to streamline the Sqoop input methods (the issues cited were increasingly complex command lines, security, and the need to understand too many low-level issues), Sqoop version 2 no longer supports specialized connectors or direct import into HBase or Hive or direct data transfer from Hive or HBase to your RDBMS. There are more generalized ways to accomplish these tasks in version 2. All import and export is done through the JDBC interface. Table 4.1 summarizes the changes. Due to these changes, any new development should be done with attention to Sqoop version 2 capabilities.

Using Sqoop V2: A Basic Example

To better understand how to use Sqoop in practice, we're going to demonstrate how to configure and use Sqoop version 2 via a simple example. The example can then be extended as needed to explore the other capabilities offered by Apache Sqoop. More detailed information can be found at the Sqoop website at http://sqoop.apache.org.

The following steps will be performed:

1. Download and load sample MySQL data
2. Add Sqoop user permissions for local machine and cluster
3. Import data from MySQL to HDFS
4. Export data from HDFS to MySQL

Step 1: Download a Sample MySQL Database

For this example, we assume MySQL is installed on the Sqoop node and will use the world example database from the MySQL site (http://dev.mysql.com/doc/world-setup/en/index.html). The database has three tables:

- Country—Information about countries of the world.
- City—Information about some of the cities in those countries.
- CountryLanguage—Languages spoken in each country.

1. To get the database, use `wget`[3] to download and then extract the file:

```
$ wget http://downloads.mysql.com/docs/world.sql.gz
$ gunzip world.sql.gz
```

2. Next, log into MySQL (assumes you have privileges to create a database) and import that database by entering the following commands:

```
$ mysql -u root -p
mysql> CREATE DATABASE world;
mysql> USE world;
mysql> SOURCE world.sql;
mysql> SHOW TABLES;
+-----------------+
| Tables_in_world |
+-----------------+
| City            |
| Country         |
| CountryLanguage |
+-----------------+
3 rows in set (0.01 sec)
```

3. The following MySQL commands will let you see the details for each table (output omitted because of space considerations):

```
mysql> SHOW CREATE TABLE Country;
mysql> SHOW CREATE TABLE City;
mysql> SHOW CREATE TABLE CountryLanguage;
```

Step 2: Add Sqoop User Permissions for Local Machine and Cluster

Sqoop often needs to talk to MySQL from the Hadoop cluster. Thus, there needs to be permissions added to MySQL so that these conversations can take place. Depending on your installation, you may need to add several privileges for Sqoop requests based on the location (hosts or IP addresses) from where the request originates. For example, the following permissions were assigned for the example.

---

3. `wget` is a command line tool for Unix/Linux environments that directly downloads files from a valid URL. If using a Windows environment, consider `Winwget` or a browser. If using a Macintosh environment, consider using `curl -O <url>` or a browser.

```
mysql> GRANT ALL PRIVILEGES ON world.* To 'sqoop'@'localhost'
➥ IDENTIFIED BY 'sqoop';
mysql> GRANT ALL PRIVILEGES ON world.* To 'sqoop'@'_HOSTAME_'
➥ IDENTIFIED BY 'sqoop';
mysql> GRANT ALL PRIVILEGES ON world.* To 'sqoop'@'_SUBNET_'
➥ IDENTIFIED BY 'sqoop';
FLUSH PRIVILEGES;
mysql> quit
```

The `_HOSTNAME_` is the name of the host on which a user has logged in. The `_SUBNET_` is the subnet of the cluster (for example 10.0.0.%, defines 10.0.0.0/24 network). These permissions allow any node in the cluster to execute MySQL commands as user sqoop. Also, for the purposes of this example, the Sqoop password is "sqoop."

Next, log in as user sqoop to test the MySQL permissions.

```
$ mysql -u sqoop -p
mysql> USE world;
   mysql> SHOW TABLES;
   +-----------------+
   | Tables_in_world |
   +-----------------+
   | City            |
   | Country         |
   | CountryLanguage |
   +-----------------+
   3 rows in set (0.01 sec)

   mysql> quit
```

Step 3: Import Data Using Sqoop

As a check of Sqoop's capability to read the MySQL database, we can use Sqoop to list the databases in MySQL.

1. Enter the following commands. The results are after the warnings at the end of the output. Note the use of local `_HOSTNAME_` in the JDBC statement. Extra notifications have been removed from the output (represented by ...).

```
$ sqoop list-databases --connect jdbc:mysql://_HOSTNAME_/world
➥ --username sqoop --password sqoop
...
information_schema
test
world
```

2. In a similar fashion, Sqoop can connect to MySQL and list the tables in the world database.

```
$ sqoop list-tables --connect jdbc:mysql://_HOSTNAME_/world
➥ --username sqoop --password sqoop
...
City
Country
CountryLanguage
```

3. In order to import data, we need to make a directory in HDFS:

```
$ hdfs dfs -mkdir sqoop-mysql-import
```

4. The following command will import the Country table into HDFS:

```
$ sqoop import --connect jdbc:mysql://_HOSTNAME_/world  --username
➥ sqoop --password  sqoop --table Country  -m 1 --target-dir
➥ /user/username/sqoop-mysql-import/country
```

The option `--table` signifies the table to import, `--target-dir` is the directory created above, and `-m 1` tells `sqoop` to use a single map task (which is enough in our example since it is only a small table) to import the data.

5. The import can be confirmed by examining HDFS:

```
$ hdfs dfs -ls sqoop-mysql-import/country
Found 2 items
-rw-r--r--   2 username hdfs          0 2014-08-18 16:47 sqoop-mysql-
➥import/world/_SUCCESS
-rw-r--r--   2 username hdfs      31490 2014-08-18 16:47 sqoop-mysql-
➥import/world/part-m-00000
```

6. The file can be viewed using the `hdfs -cat` command:

```
$ hdfs dfs -cat sqoop-mysql-import/country/part-m-00000
ABW,Aruba,North America,Caribbean,193.0,null,103000,78.4,828.0,793.0,
➥ Aruba,Nonmetropolitan Territory of The Netherlands,Beatrix,129,AW
...
ZWE,Zimbabwe,Africa,Eastern Africa,390757.0,1980,11669000,37.8,
➥ 5951.0,8670.0,Zimbabwe,Republic,Robert G. Mugabe,4068,ZW
```

To make Sqoop commands more convenient, an options file may be created and used in the command line. This file will help you avoid having to rewrite the same options. For example, a file called world-options.txt with the following contents will include the `import` command, `--connect`, `--username`, and `--password` options:

```
import
--connect
jdbc:mysql://_HOSTNAME_/world
--username
sqoop
--password
sqoop
```

The same import command from the preceding can be performed with the following shorter line:

```
$ sqoop  --options-file world-options.txt --table City  -m 1 --target-dir
➥ /user/username/sqoop-mysql-import/city
```

It is also possible to include an SQL Query in the import step. For example, if we want just cities in Canada:

```
SELECT ID,Name from City WHERE CountryCode='CAN'
```

Then we can include the `--query` option in the Sqoop import request. In the following query example, a single mapper task is designated with the `-m 1` option:

```
sqoop  --options-file world-options.txt -m 1 --target-dir
➥ /user/username/sqoop-mysql-import/canada-city --query
➥ "SELECT ID,Name from City
➥ WHERE CountryCode='CAN' AND \$CONDITIONS"
```

Inspecting the results shows only cities from Canada are imported.

```
$ hdfs dfs -cat sqoop-mysql-import/canada-city/part-m-00000

1810,Montréal
1811,Calgary
1812,Toronto
...
1856,Sudbury
1857,Kelowna
1858,Barrie
```

Since there was only one mapper process, only one copy of the query needed to be run on the database. The results are also reported in single file (`part-m-0000`). Multiple mappers can be used to process the query if the `--split-by` option is used. The split-by option is a way to parallelize the SQL query. Each parallel task runs a subset of the main query with results partitioned by bounding conditions inferred by Sqoop. Your query must include the token `$CONDITIONS`; this is a placeholder for Sqoop to put in unique condition expression based on the `--split-by` option, and Sqoop automatically populates this with the right conditions for each mapper task. Sqoop will try to create balanced sub-queries based on a range of your primary key. However, it may be necessary to split on another column if your primary key is not uniformly distributed.

The following example will help illustrate the `-split-by` option. First, remove the results of the previous query.

```
$ hdfs dfs -rm -r -skipTrash  sqoop-mysql-import/canada-city
```

Next, run the query using four mappers (`-m 4`) where we split by the ID number (`--split-by ID`).

```
sqoop  --options-file world-options.txt -m 4 --target-dir
➥ /user/username/sqoop-mysql-import/canada-city --query "SELECT ID,
➥ Name from City WHERE CountryCode='CAN' AND \$CONDITIONS" --split-by ID
```

If we look at the number of results files, we find four files corresponding to the four mappers we requested in the command. There is no need to combine these files into one entity because all Hadoop tools can manage multiple files as input.

```
$ hdfs dfs -ls  sqoop-mysql-import/canada-city
Found 5 items
-rw-r--r--   2 username hdfs       0 2014-08-18 21:31 sqoop-mysql-import/canada-
city/_SUCCESS
-rw-r--r--   2 username hdfs     175 2014-08-18 21:31 sqoop-mysql-import/canada-
city/part-m-00000
```

```
-rw-r--r--   2 username hdfs     153 2014-08-18 21:31 sqoop-mysql-import/canada-
city/part-m-00001
-rw-r--r--   2 username hdfs     186 2014-08-18 21:31 sqoop-mysql-import/canada-
city/part-m-00002
-rw-r--r--   2 username hdfs     182 2014-08-18 21:31 sqoop-mysql-import/canada-
city/part-m-00003
```

Step 4: Export Data Using Sqoop

The first step when exporting data with Sqoop is to create tables in the target database system for the exported data. There are actually two tables needed for each exported table. The first is a table to hold the exported data (e.g., `CityExport`) and the second is a table to be used for staging the exported data (e.g., `CityExportStaging`).

1. Using the following MySQL commands, you can create the tables:

```
mysql> USE world;
mysql> CREATE TABLE `CityExport` (
       `ID` int(11) NOT NULL AUTO_INCREMENT
       `Name` char(35) NOT NULL DEFAULT '',
       `CountryCode` char(3) NOT NULL DEFAULT '',
       `District` char(20) NOT NULL DEFAULT '',
       `Population` int(11) NOT NULL DEFAULT '0',
        PRIMARY KEY (`ID`));
mysql> CREATE TABLE `CityExportStaging` (
       `ID` int(11) NOT NULL AUTO_INCREMENT,
       `Name` char(35) NOT NULL DEFAULT '',
       `CountryCode` char(3) NOT NULL DEFAULT '',
       `District` char(20) NOT NULL DEFAULT '',
       `Population` int(11) NOT NULL DEFAULT '0',
        PRIMARY KEY (`ID`));
```

2. Next, create a cities-export-options.txt file similar to the world-options.txt file created above, using the `export` instead of `import` command. The following will export the cities data we imported above back into MySQL:

```
sqoop --options-file cities-export-options.txt --table CityExport
➥ --staging-table CityExportStaging  --clear-staging-table -m 4
➥ --export-dir /user/username/sqoop-mysql-import/city
```

3. Finally, to make sure everything worked, check the table in MySQL to see if the cities are in the table.

```
$ mysql> select * from CityExport limit 10;
+----+----------------+-------------+---------------+------------+
| ID | Name           | CountryCode | District      | Population |
+----+----------------+-------------+---------------+------------+
|  1 | Kabul          | AFG         | Kabol         |    1780000 |
|  2 | Qandahar       | AFG         | Qandahar      |     237500 |
|  3 | Herat          | AFG         | Herat         |     186800 |
|  4 | Mazar-e-Sharif | AFG         | Balkh         |     127800 |
|  5 | Amsterdam      | NLD         | Noord-Holland |     731200 |
|  6 | Rotterdam      | NLD         | Zuid-Holland  |     593321 |
```

```
|  7 | Haag             | NLD          | Zuid-Holland  |     440900 |
|  8 | Utrecht          | NLD          | Utrecht       |     234323 |
|  9 | Eindhoven        | NLD          | Noord-Brabant |     201843 |
| 10 | Tilburg          | NLD          | Noord-Brabant |     193238 |
+----+-----------------+-------------+--------------+------------+
10 rows in set (0.00 sec)
```

Some Handy Clean-up Commands

If you are not real familiar with MySQL, the following commands may be helpful to clean up the examples.

To remove a table in MySQL:

```
mysql> Drop table `CityExportStaging`;
```

To remove the data in a table:

```
mysql> delete from CityExportStaging;
```

To clean up imported files:

```
$ hdfs dfs -rm -r  -skipTrash sqoop-mysql-import/{country,city,
➥ canada-city}
```

Using Apache Flume to Acquire Data Streams

In addition to structured data in databases, another common source of data is log files, which usually come in the form of continuous (streaming) incremental files often from multiple source machines. In order to use this type of data for data science with Hadoop, we need a way to ingest such data into HDFS.

Apache Flume is designed to collect, transport, and store data streams into HDFS. Often data transport involves a number of Flume agents that may traverse a series of machines and locations. Flume is often used for log files, social-media-generated data, email messages, and pretty much any continuous data source.

As shown in Figure 4.4, a Flume agent is composed of three components:

- **Source**—The source component receives data and sends it to a channel. It can send the data to more than one channel. The input data can be from a real-time source (e.g. web log) or another Flume agent.
- **Channel**—A channel is a data queue that forwards the source data to the sink destination. It can be thought of as a buffer that manages input (source) and output (sink) flow rates.
- **Sink**—The sink delivers data to destinations such as HDFS, a local file, or another Flume agent.

A Flume agent can have multiple sources, channels, and sinks but must have at least one of each of the three components defined. Sources can write to multiple channels, but a sink can only take data from a single channel. Data written to a channel remain

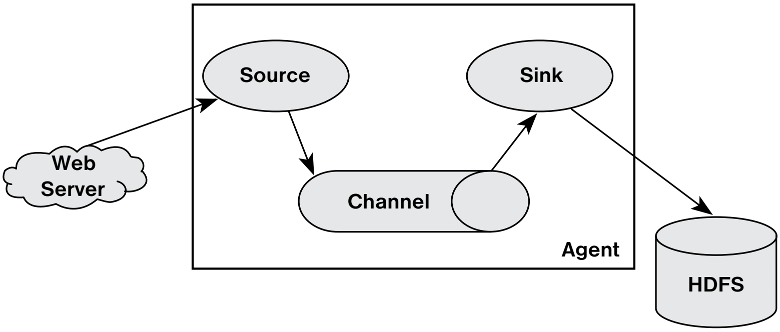

Figure 4.4 Flume Agent with Source, Channel, and Sink.

in the channel until a sink removes the data. By default, the data in a channel is kept in memory but optionally may be stored on disk to prevent data loss in the event of a network failure.

As shown in Figure 4.5, Flume agents may be placed in a pipeline. This configuration is normally used when data is collected on one machine (e.g., a web server) and sent to another machine that has access to HDFS.

In a Flume pipeline, the sink from one agent is connected to the source of another. The data transfer format normally used by Flume is called Apache Avro[4] and provides several useful features. First, Avro is a data serialization/deserialization system that uses a compact binary format. The schema is sent as part of the data exchange and is defined using JavaScript Object Notation (JSON). Avro also uses remote procedure calls (RPC) to send data. That is, an Avro sink will contact an Avro source to send data.

Another useful Flume configuration is shown in Figure 4.6. In this configuration, Flume is used to consolidate several data sources before committing them to HDFS.

There are many possible ways to construct Flume transport networks.

The full scope of Flume functionality is beyond the scope of this book, and there are many additional features in Flume such as plug-ins and interceptors that can enhance

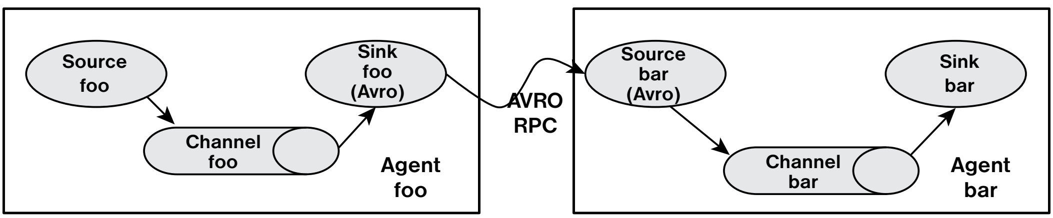

Figure 4.5 Pipeline created by connecting Flume agents.

4. https://avro.apache.org/

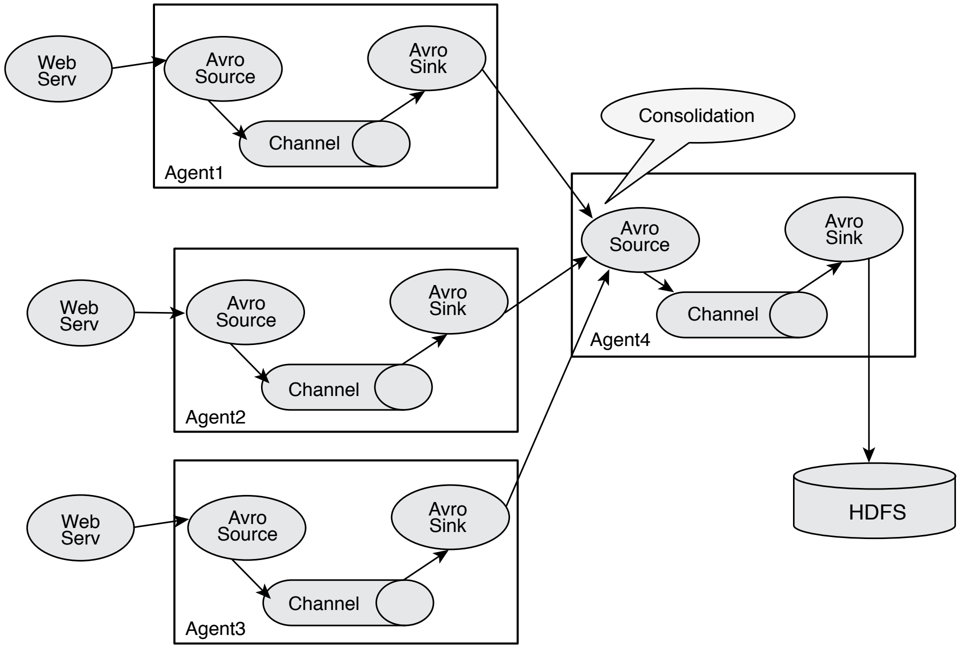

Figure 4.6 A Flume consolidation network.

Flume pipelines. For more information and example configurations, please see the Flume Users Guide at https://flume.apache.org/FlumeUserGuide.html.

Using Flume: A Web Log Example Overview

In this example web logs from the local machine will be placed into HDFS using Flume. This example is easily modified to use other web logs from different machines. The full source code and further implementation notes are available from the book web page in Appendix A, “Book Web Page and Code Download.” Two files are needed to configure Flume. (See the sidebar “Flume Configuration Files.”)

- `web-server-target-agent.conf`—The target Flume agent that writes the data to HDFS
- `web-server-source-agent.conf`—The source Flume agent that captures the web log data

The web log is also mirrored on the local file system by the agent that writes to HDFS.

1. To run the example, create the directory as root.

```
# mkdir /var/log/flume-hdfs
# chown hdfs:hadoop /var/log/flume-hdfs/
```

2. Next, as user hdfs, make a Flume data directory in HDFS.

```
$ hdfs dfs -mkdir /user/hdfs/flume-channel/
```

3. Now that the data directories are created, the Flume target agent can be started (as user hdfs).

```
$ flume-ng agent -c conf -f web-server-target-agent.conf -n collector
```

This agent writes the data into HDFS and should be started before the source agent. (The source reads the web logs.)

> **Note**
>
> In some Hadoop distributions, Flume can be started as a service when the system boots, such as "service start flume." This configuration allows for automatic use of the Flume agent. The /etc/flume/conf/{flume.conf,flume-env.sh.template} files need to be configured for this purpose. For this example, the /etc/flume/conf/flume.conf file can be the same as the web-server-target.conf file (modified for your environment).

The source agent can be started as root, which will start to feed the web log data to the target agent. Note that the source agent can be on another machine:

```
# flume-ng agent -c conf -f web-server-source-agent.conf -n source_agent
```

To see if Flume is working, check the local log by using tail. Also check to make sure the flume-ng agents are not reporting any errors (filename will vary).

```
$ tail -f /var/log/flume-hdfs/1430164482581-1
```

The contents of the local log under flume-hdfs should be identical to that written into HDFS. The file can be inspected using the hdfs -tail command. (filename will vary). Note, while running Flume, the most recent file in HDFS may have a .tmp appended to it. The .tmp indicates that the file is still being written by Flume. The target agent can be configured to write the file (and start another .tmp file) by setting some or all of the rollCount, rollSize, rollInterval, idleTimeout, and batchSize options in the configuration file.

```
$ hdfs dfs -tail flume-channel/apache_access_combined/150427/FlumeData.
➥1430164801381
```

Both files should have the same data in them. For instance, the preceding example had the following in both files:

```
10.0.0.1 - - [27/Apr/2015:16:04:21 -0400] "GET /ambarinagios/nagios/nagios_alerts
.php?q1=alerts&alert_type=all HTTP/1.1" 200 30801 "-" "Java/1.7.0_65"
10.0.0.1 - - [27/Apr/2015:16:04:25 -0400] "POST /cgi-bin/rrd.py HTTP/1.1" 200 784
"-" "Java/1.7.0_65"
10.0.0.1 - - [27/Apr/2015:16:04:25 -0400] "POST /cgi-bin/rrd.py HTTP/1.1" 200 508
"-" "Java/1.7.0_65"
```

Both the target and source file can be modified to suit your system.

Flume Configuration Files

A complete explanation of Flume configuration is beyond the scope of this chapter. The Flume website has additional information on Flume configuration at http://flume.apache.org/FlumeUserGuide.html#configuration.

The two files describe two Flume agents that have separate Source/Channel/Sink configurations. Some of the important settings used in the example above are as follows:

In `web-server-source-agent.conf`, the following lines set the source. Note that the web log is acquired by using the `tail` command to record the log file.

```
source_agent.sources = apache_server
source_agent.sources.apache_server.type = exec
source_agent.sources.apache_server.command = tail -f /etc/httpd/logs/access_log
```

Further down in the file, the sink is defined. The parameter `source_agent.sinks.avro_sink.hostname` is used to assign the Flume node that will write to HDFS. The port number is also set in the target configuration file.

```
source_agent.sinks = avro_sink
source_agent.sinks.avro_sink.type = avro
source_agent.sinks.avro_sink.channel = memoryChannel
source_agent.sinks.avro_sink.hostname =  192.168.93.24
source_agent.sinks.avro_sink.port = 4545
```

The HDFS settings are placed in the web-server-target-agent.conf file. Note the path that was used in the previous example and the data specification.

```
collector.sinks.HadoopOut.type = hdfs
collector.sinks.HadoopOut.channel = mc2
collector.sinks.HadoopOut.hdfs.path = /user/hdfs/flume-channel/%{log_type}/
%y%m%d
collector.sinks.HadoopOut.hdfs.fileType = DataStream
```

The target file also defines the port and two channels (mc1 and mc2). One of the channels writes the data to the local file system and the other writes to HDFS. The relevant lines are shown in the following:

```
collector.sources.AvroIn.port = 4545
collector.sources.AvroIn.channels = mc1 mc2

collector.sinks.LocalOut.sink.directory = /var/log/flume-hdfs
collector.sinks.LocalOut.channel = mc1
```

The HDFS file rollover counts create a new file when a threshold is exceeded. In this example, allow any file size and write a new file after 10,000 events or 600 seconds.

```
collector.sinks.HadoopOut.hdfs.rollSize = 0
collector.sinks.HadoopOut.hdfs.rollCount = 10000
collector.sinks.HadoopOut.hdfs.rollInterval = 600
```

A full discussion of Flume can be found on the website at https://flume.apache.org.

Manage Hadoop Work and Data Flows with Apache Oozie

Apache Oozie is a workflow scheduler system designed to run and manage multiple related Apache Hadoop jobs. For instance, complete data input and analysis may require several discrete Hadoop jobs to be run as a workflow where the output of one job will be the input for a successive job. Oozie is designed to construct and manage these workflows.

Oozie is not a substitute for the YARN scheduler mentioned previously. That is, YARN manages resources for individual Hadoop jobs, and Oozie provides a way to connect and control multiple Hadoop jobs on the cluster.

Oozie workflow jobs are represented as DAGs of actions. There are three types of Oozie jobs:

- **Workflow:** A specified sequence of Hadoop jobs with outcome-based decision points and control dependency. Progress from one action to another cannot happen until the first action is complete.
- **Coordinator**: A scheduled workflow job that can run at various time intervals or when data becomes available.
- **Bundle**: A higher-level Oozie abstraction that will batch a set of coordinator jobs.

Oozie is integrated with the rest of the Hadoop stack supporting several types of Hadoop jobs out of the box (such as Java MapReduce, Streaming MapReduce, Pig, Hive, Spark, and Sqoop) as well as system-specific jobs (such as Java programs and shell scripts). Oozie also provides a CLI and a Web UI for monitoring jobs. An example of a simple Oozie workflow is shown in Figure 4.7. In this example, Oozie runs a basic MapReduce operation. If the application was successful the job ends; if there was an error, the job is killed.

Oozie workflow definitions are written in Hadoop Process Definition Language (hPDL), which is an XML-based process definition language. Oozie workflows contain several types of nodes.

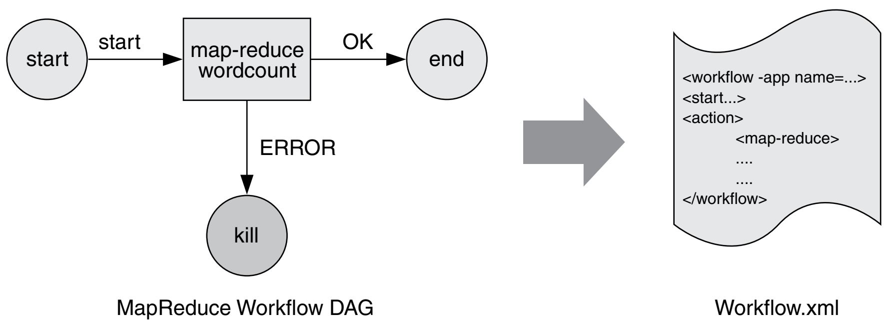

Figure 4.7 A simple Oozie DAG workflow.

- **Start/Stop control flow nodes** define the beginning and the end of a workflow. These include start, end, and optional fail nodes.
- **Action nodes** are where the actual processing tasks are defined. When an action node finishes, the remote systems notify Oozie and the next node in the workflow is executed. Action nodes can also include HDFS commands.
- **Fork/join nodes** allow parallel execution of tasks in the workflow. The fork node allows two or more tasks to run at the same time. A join node represents a rendezvous point that must wait until all forked tasks complete.
- **Control flow nodes** enable decisions to be made about the previous task. Control decisions are based on the results of the previous action (e.g. file size or file existence). Decision nodes are essentially switch-case statements that use JSP EL (Java Server Pages-Expression Language) that evaluates to either true or false.

A more complex workflow that uses all the above nodes is shown in the example workflow in Figure 4.8. More information on Oozie can be found at http://oozie.apache.org/docs/4.0.0/index.html.

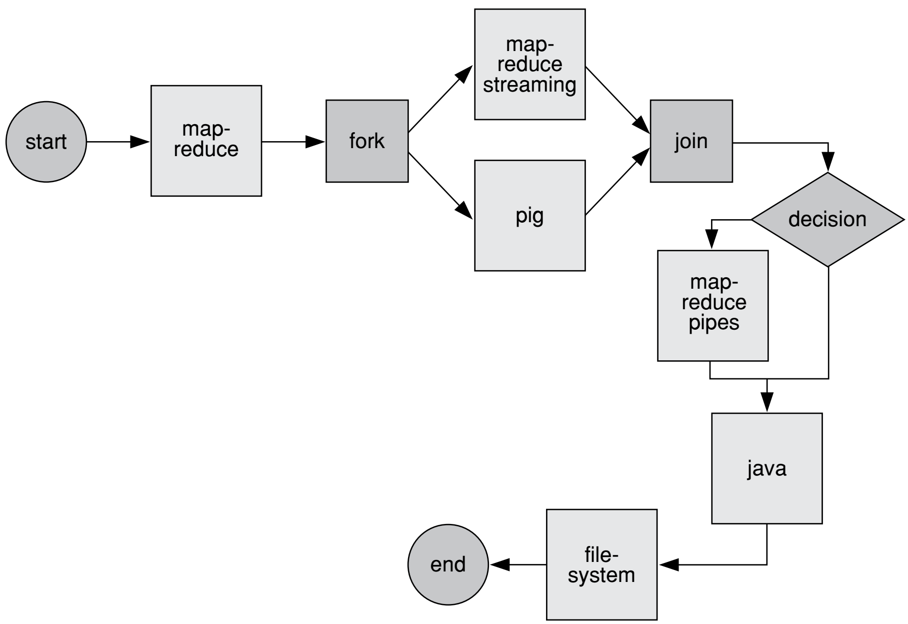

Figure 4.8 A more complex Oozie DAG workflow.

Apache Falcon

Apache Falcon simplifies the configuration of data motion by providing replication, life cycle management, lineage, and traceability. These features provide data governance consistency across Hadoop components that is not possible using Oozie. For instance, Falcon allows Hadoop administrators to centrally define their **data pipelines**, and then Falcon uses those definitions to auto-generate workflows in Apache Oozie. In simple terms, proper use of Falcon helps keep your active Hadoop cluster from becoming a confusing mess.

For example, Oozie lets you define Hadoop processing through workflow and coordinator (a recurring workflow) jobs. The input datasets for data processing are often described as part of coordinator jobs that specify properties such as path, frequency, schedule runs, and so on. If there are two coordinator jobs that depend on the same data, these details have to be defined and managed twice. If you want to add shared data deletion or movement, a separate coordinator is required. Oozie will certainly work in these situations, but there is no easy way to define and track the entire data life cycle or manage multiple independent Oozie jobs.

Oozie is useful when initially setting up and testing workflows and can be used when the workflows are independent and not expected to change often. If there are multiple dependencies between workflows or there is a need to manage the entire data life cycle, then Falcon should be considered.

As mentioned, as Hadoop's high-level workflow scheduler, Oozie may be managing hundreds to thousands of coordinator jobs and files. This situation results in some common mistakes. Processes might use the wrong copies of datasets. Datasets and processes may be duplicated, and it becomes increasingly more difficult to track down where a particular dataset originated. At that level of complexity, it becomes difficult to manage so many dataset and process definitions.

To solve these problems, Falcon allows the creation of a pipeline that is defined by three key attributes:

- A **cluster** entity that defines where data, tools, and processes live on your Hadoop cluster. A cluster entity contains things like the namenode address, Oozie URL, etc., which it uses to execute the other two entities: feeds and processes.
- A **feed** entity defines where data live on your cluster (in HDFS). The feed is designed to designate to Falcon where your new data (that's either ingested, processed, or both) live so it can retain (through retention policies) and replicate (through replication policies) the data on or from your Cluster. A feed is typically (but doesn't have to be) the output of a process.
- A **process** entity defines what action or "process" will be taking place in a pipeline. Most typically, the process links to an Oozie workflow, which contains a series of actions to execute (such as shell scripts, Java Jars, Hive actions, Pig actions, Sqoop Actions, you name it) on your cluster. A process also, by definition, takes feeds as inputs or outputs and is where you define how often a workflow should run.

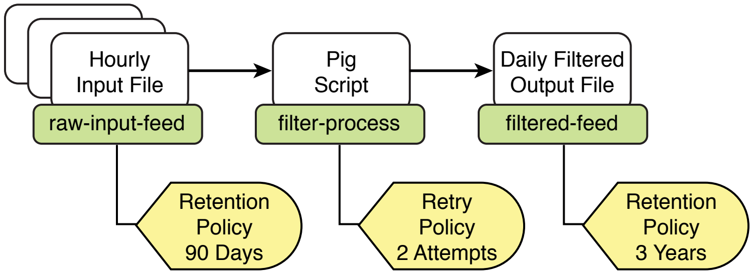

Figure 4.9 A simple Apache Falcon workflow.

The following example will help explain how Falcon is used. Assume there is raw input data that arrives every hour. These data are processed with a Pig script and the results saved for later processing. At a simple level an Oozie workflow could easily manage the task. However, high-level features, not available in Oozie, are needed to automate the process. First, the input data have a retention policy of 90 days, after which old data are discarded. Second, the processing step may have a certain number of retries should the process fail. And, finally, the output data have a retention policy of three years (and location). It is also possible to query data lineage with Falcon (i.e., Where did this data come from?). The simple job flow is shown in Figure 4.9.

What's Next in Data Ingestion?

As the Hadoop platform continues to evolve, innovation in ingestion tools continues. Two important new tools are now available to ingestion teams that we would like to mention:

- **Apache Nifi** is a recent addition to the data ingestion toolset. Originally created at the NSA and recently open sourced and added to the Apache family, Nifi provides a scalable way to define data routing, transformation, and system mediation logic. An excellent UI makes building data flows in Nifi fast and easy. Nifi provides support for lineage tracking and the security and monitoring capability that make it a great tool for data ingestion, especially for sensor data.
- **Apache Atlas** provides a set of core data governance services that enables enterprises to effectively deal with compliance requirements on Hadoop.

Summary

In this chapter

- The Hadoop data lake concept was presented as a new model for data processing.
- Various methods for making data available to several Hadoop tools were outlined. The examples included copying files directly to HDFS, importing CSV files to Apache Hive and Spark, and importing JSON files into HIVE with Spark.

- Apache Sqoop was presented as a tool for moving relational data into and out of HDFS.
- Apache Flume was presented as tool for capturing and transporting continuous data, such as web logs, into HDFS.
- The Apache Oozie workflow manager was described as a tool for creating and scheduling Hadoop workflows.
- The Apache Falcon tool enables a high-level framework for data governance (end-to-end management) by keeping Hadoop data and tasks organized and defined as pipelines.
- New tools like Apache Nifi and Atlas were mentioned as options for governance and data flow on a Hadoop cluster.

This page intentionally left blank

Index

E

F

Q

R

S

T