Community Experience Distilled

Programming MapReduce with Scalding

A practical guide to designing, testing, and implementing complex MapReduce applications in Scala

Antonios Chalkiopoulos

[PACKT] PUBLISHING
open source*
community experience distilled

Programming MapReduce with Scalding

A practical guide to designing, testing, and implementing complex MapReduce applications in Scala

Antonios Chalkiopoulos

[PACKT] PUBLISHING open source community experience distilled

BIRMINGHAM - MUMBAI

Programming MapReduce with Scalding

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: June 2014

Production reference: 1190614

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78328-701-7

www.packtpub.com

Credits

Author

Antonios Chalkiopoulos

Reviewers

Ahmad Alkilani

Włodzimierz Bzyl

Tanin Na Nakorn

Sen Xu

Commissioning Editor

Owen Roberts

Acquisition Editor

Llewellyn Rozario

Content Development Editor

Sriram Neelakantan

Technical Editor

Kunal Anil Gaikwad

Copy Editors

Sayanee Mukherjee

Alfida Paiva

Project Coordinator

Aboli Ambardekar

Proofreaders

Mario Cecere

Maria Gould

Indexers

Mehreen Deshmukh

Rekha Nair

Tejal Soni

Graphics

Sheetal Aute

Ronak Dhruv

Valentina Dsilva

Disha Haria

Production Coordinator

Conidon Miranda

Cover Work

Conidon Miranda

Cover Image

Sheetal Aute

About the Author

Antonios Chalkiopoulos is a developer living in London and a professional working with Hadoop and Big Data technologies. He completed a number of complex MapReduce applications in Scalding into 40-plus production nodes HDFS Cluster. He is a contributor to Scalding and other open source projects, and he is interested in cloud technologies, NoSQL databases, distributed real-time computation systems, and machine learning.

He was involved in a number of Big Data projects before discovering Scala and Scalding. Most of the content of this book comes from his experience and knowledge accumulated while working with a great team of engineers.

> I would like to thank Rajah Chandan for introducing Scalding to the team and being the author of SpyGlass and Stefano Galarraga for co-authoring chapters 5 and 6 and being the author of ScaldingUnit. Both these libraries are presented in this book.
>
> Saad, Gracia, Deepak, and Tamas, I've learned a lot working next to you all, and this book wouldn't be possible without all your discoveries. Finally, I would like to thank Christina for bearing with my writing sessions and supporting all my endeavors.

About the Reviewers

Ahmad Alkilani is a data architect specializing in the implementation of high-performance distributed systems, data warehouses, and BI systems. His career has been split between building enterprise applications and products using a variety of web and database technologies, including .NET, SQL Server, Hadoop, Hive, Scala, and Scalding. His recent interests include building real-time web and predictive analytics and streaming and sketching algorithms.

Currently, Ahmad works at Move.com (`http://www.realtor.com`) and enjoys speaking at various user groups and national conferences, and he is an author on Pluralsight with courses focused on Hadoop and Big Data, SQL Server 2014, and more, targeting the Big Data and streaming spaces.

You can find more information on Ahmad on his LinkedIn profile (`http://www.linkedin.com/in/ahmadalkilani`) or his Pluralsight author page (`http://pluralsight.com/training/Authors/Details/ahmad-alkilani`).

> I would like to thank my family, especially my wonderful wife, Farah, and my beautiful son Maher for putting up with my long working hours and always being there for me.

Włodzimierz Bzyl works at the University of Gdańsk. His current interests include web-related technologies and NoSQL databases.

He has a passion for new technologies and introducing his students to them.

He enjoys contributing to open source software and spending time trekking in the Tatra mountains.

Tanin Na Nakorn is a software engineer who is enthusiastic about building consumer products and open source projects that make people's lives easier. He cofounded Thaiware, a software portal in Thailand and GiveAsia, a donation platform in Singapore; he currently builds products at Twitter. You may find him expressing himself on his Twitter handle `@tanin` and helping on various open source projects at `http://www.github.com/tanin47`.

Sen Xu is a software engineer in Twitter; he was previously a data scientist in Inome Inc.

He worked on designing and building data pipelines on top of traditional RDBMS (MySQL, PostgreSQL, and so on) and key-value store solutions (Hadoop). His interests include Big Data analytics, text mining, record linkage, machine learning, and spatial data handling.

www.PacktPub.com

Support files, eBooks, discount offers, and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

PACKTLIB™

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Scalding is a relatively new Scala DSL that builds on top of the Cascading pipeline framework, offering a powerful and expressive architecture for MapReduce applications. Scalding provides a highly abstracted layer for design and implementation in a componentized fashion, allowing code reuse and development with the Test Driven Methodology.

Similar to other popular MapReduce technologies such as Pig and Hive, Cascading uses a tuple-based data model, and it is a mature and proven framework that many dynamic languages have built technologies upon. Instead of forcing developers to write raw map and reduce functions while mentally keeping track of key-value pairs throughout the data transformation pipeline, Scalding provides a more natural way to express code.

In simpler terms, programming raw MapReduce is like developing in a low-level programming language such as assembly. On the other hand, Scalding provides an easier way to build complex MapReduce applications and integrates with other distributed applications of the Hadoop ecosystem.

This book aims to present MapReduce, Hadoop, and Scalding, it suggests design patterns and idioms, and it provides ample examples of real implementations for common use cases.

What this book covers

Chapter 1, Introduction to MapReduce, serves as an introduction to the Hadoop platform, MapReduce and to the concept of the pipeline abstraction that many Big Data technologies use. The first chapter outlines Cascading, which is a sophisticated framework that empowers developers to write efficient MapReduce applications.

Chapter 2, Get Ready for Scalding, lays the foundation for working with Scala, using build tools and an IDE, and setting up a local-development Hadoop system. It is a hands-on chapter that completes packaging and executing a Scalding application in local mode and submitting it in our Hadoop mini-cluster.

Chapter 3, Scalding by Example, teaches us how to perform map-like operations, joins, grouping, pipe, and composite operations by providing examples of the Scalding API.

Chapter 4, Intermediate Examples, illustrates how to use the Scalding API for building real use cases, one for log analysis and another for ad targeting. The complete process, beginning with data exploration and followed by complete implementations, is expressed in a few lines of code.

Chapter 5, Scalding Design Patterns, presents how to structure code in a reusable, structured, and testable way following basic principles in software engineering.

Chapter 6, Testing and TDD, focuses on a test-driven methodology of structuring projects in a modular way for maximum testability of the components participating in the computation. Following this process, the number of bugs is reduced, maintainability is enhanced, and productivity is increased by testing every layer of the application.

Chapter 7, Running Scalding in Production, discusses how to run our jobs on a production cluster and how to schedule, configure, monitor, and optimize them.

Chapter 8, Using External Data Stores, goes into the details of accessing external NoSQL- or SQL-based data stores as part of a data processing workflow.

Chapter 9, Matrix Calculations and Machine Learning, guides you through the process of applying machine learning algorithms, matrix calculations, and integrating with Mahout algorithms. Concrete examples demonstrate similarity calculations on documents, items, and sets.

What you need for this book

Prior knowledge about Hadoop or Scala is not required to follow the topics and techniques, but it is certainly beneficial. You will need to set up your environment with the JDK, an IDE, and Maven as a build tool. As this is a practical guide you will need to set up a mini Hadoop cluster for development purposes.

Who this book is for

This book is structured in such a way as to introduce Hadoop and MapReduce to a developer who has a basic understanding of these technologies and to leverage existing and well-known tools in order to become highly productive. A more experienced Scala developer will benefit from the Scalding design patterns, and an experienced Hadoop developer will be enlightened by this alternative methodology of developing MapReduce applications with Scalding.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, and user input are shown as follows: "A Map class to map lines into `<key,value>` pairs; for example, `<"INFO",1>`."

A block of code is set as follows:

```
LogLine    = load 'file.logs' as (level, message);
LevelGroup = group LogLine by level;
Result     = foreach LevelGroup generate group, COUNT(LogLine);
store Result into 'Results.txt';
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import com.twitter.scalding._

class CalculateDailyAdPoints (args: Args) extends Job(args) {

  val logSchema = List ('datetime, 'user, 'activity, 'data,
   'session, 'location, 'response, 'device, 'error, 'server)

  val logs = Tsv("/log-files/2014/07/01", logSchema )
   .read
   .project('user,'datetime,'activity,'data)
   .groupBy('user) { group => group.sortBy('datetime) }
   .write(Tsv("/analysis/log-files-2014-07-01"))
}
```

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

```
$ echo "This is a happy day. A day to remember" > input.txt
$ hadoop fs -mkdir -p hdfs:///data/input hdfs:///data/output
$ hadoop fs -put input.txt hdfs:///data/input/
```

New terms and **important words** are shown in bold.

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

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

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

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

Downloading the example code

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

Also you can access the latest code from GitHub at `https://github.com/scalding-io/ProgrammingWithScalding` or `http://scalding.io`.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

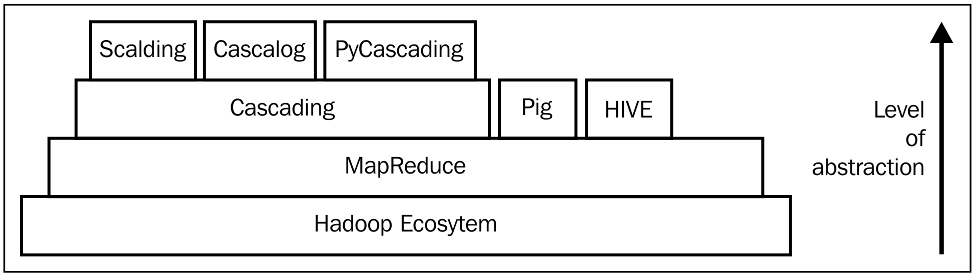
You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Introduction to MapReduce

In this first chapter, we will take a look at the core technologies used in the distributed model of Hadoop; more specifically, we cover the following:

- The Hadoop platform and the framework it provides
- The MapReduce programming model
- Technologies built on top of MapReduce that provide an abstraction layer and an API that is easier to understand and work with

In the following diagram, Hadoop stands at the base, and MapReduce as a design pattern enables the execution of distributed jobs. MapReduce is a low-level programming model. Thus, a number of libraries such as Cascading, Pig, and Hive provide alternative APIs and are compiled into MapReduce. Cascading, which is a Java application framework, has a number of extensions in functional programming languages, with Scalding being the one presented in this book.



The Hadoop platform

Hadoop can be used for a lot of things. However, when you break it down to its core parts, the primary features of Hadoop are **Hadoop Distributed File System** (**HDFS**) and MapReduce.

HDFS stores read-only files by splitting them into large blocks and distributing and replicating them across a Hadoop cluster. Two services are involved with the filesystem. The first service, the **NameNode** acts as a master and keeps the directory tree of all file blocks that exist in the filesystem and tracks where the file data is kept across the cluster. The actual data of the files is stored in multiple **DataNode** nodes, the second service.

MapReduce is a programming model for processing large datasets with a parallel, distributed algorithm in a cluster. The most prominent trait of Hadoop is that it brings processing to the data; so, MapReduce executes tasks closest to the data as opposed to the data travelling to where the processing is performed. Two services are involved in a job execution. A job is submitted to the service **JobTracker**, which first discovers the location of the data. It then orchestrates the execution of the *map* and *reduce* tasks. The actual tasks are executed in multiple **TaskTracker** nodes.

Hadoop handles infrastructure failures such as network issues, node, or disk failures automatically. Overall, it provides a framework for distributed storage within its distributed file system and execution of jobs. Moreover, it provides the service **ZooKeeper** to maintain configuration and distributed synchronization.

Many projects surround Hadoop and complete the ecosystem of available Big Data processing tools such as utilities to import and export data, NoSQL databases, and event/real-time processing systems. The technologies that move Hadoop beyond batch processing focus on in-memory execution models. Overall multiple projects, from batch to hybrid and real-time execution exist.

MapReduce

Massive parallel processing of large datasets is a complex process. MapReduce simplifies this by providing a design pattern that instructs algorithms to be expressed in map and reduce phases. *Map* can be used to perform simple transformations on data, and *reduce* is used to group data together and perform aggregations.

By chaining together a number of map and reduce phases, sophisticated algorithms can be achieved. The *shared nothing* architecture of MapReduce prohibits communication between map tasks of the same phase or reduces tasks of the same phase. Communication that's required happens at the end of each phase.

The simplicity of this model allows Hadoop to translate each phase, depending on the amount of data that needs to be processed into tens or even hundreds of tasks being executed in parallel, thus achieving scalable performance.

Internally, the map and reduce tasks follow a simplistic data representation. Everything is a key or a value. A map task receives key-value pairs and applies basic transformations emitting new key-value pairs. Data is then partitioned and different partitions are transmitted to different reduce tasks. A reduce task also receives key-value pairs, groups them based on the key, and applies basic transformation to those groups.

A MapReduce example

To illustrate how MapReduce works, let's look at an example of a log file of total size 1 GB with the following format:

```
INFO      MyApp  - Entering application.
WARNING   com.foo.Bar - Timeout accessing DB - Retrying
ERROR     com.foo.Bar  - Did it again!
INFO      MyApp  - Exiting application
```

Downloading the example code

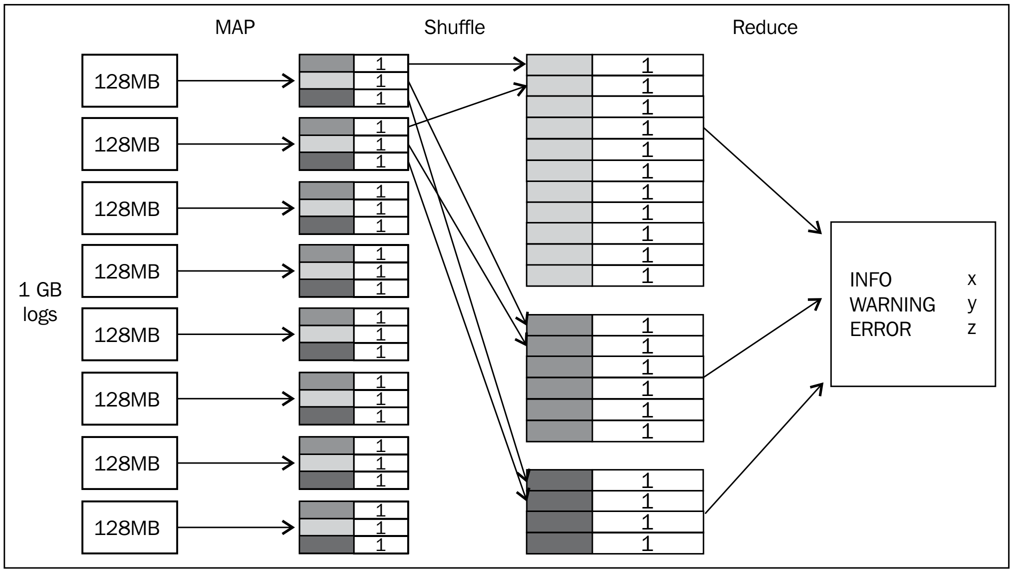
You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

Once this file is stored in HDFS, it is split into eight 128 MB blocks and distributed in multiple Hadoop nodes. In order to build a MapReduce job to count the amount of INFO, WARNING, and ERROR log lines in the file, we need to think in terms of map and reduce phases.

In one map phase, we can read local blocks of the file and map each line to a key and a value. We can use the log level as the key and the number `1` as the value. After it is completed, data is partitioned based on the key and transmitted to the reduce tasks.

MapReduce guarantees that the input to every reducer is sorted by key. *Shuffle* is the process of sorting and copying the output of the map tasks to the reducers to be used as input. By setting the value to `1` on the map phase, we can easily calculate the total in the reduce phase. Reducers receive input sorted by key, aggregate counters, and store results.

In the following diagram, every green block represents an **INFO** message, every yellow block a **WARNING** message, and every red block an **ERROR** message:



Implementing the preceding MapReduce algorithm in Java requires the following three classes:

- A Map class to map lines into `<key,value>` pairs; for example, `<"INFO",1>`
- A Reduce class to aggregate counters
- A Job configuration class to define input and output types for all `<key,value>` pairs and the input and output files

MapReduce abstractions

This simple MapReduce example requires more than 50 lines of Java code (mostly because of infrastructure and boilerplate code). In SQL, a similar implementation would just require the following:

```
SELECT level, count(*) FROM table GROUP BY level
```

Hive is a technology originating from Facebook that translates SQL commands, such as the preceding one, into sets of map and reduce phases. SQL offers convenient ubiquity, and it is known by almost everyone.

However, SQL is declarative and expresses the logic of a computation without describing its control flow. So, there are use cases that will be unusual to implement in SQL, and some problems are too complex to be expressed in relational algebra. For example, SQL handles joins naturally, but it has no built-in mechanism for splitting data into streams and applying different operations to each substream.

Pig is a technology originating from Yahoo that offers a relational data-flow language. It is procedural, supports splits, and provides useful operators for joining and grouping data. Code can be inserted anywhere in the data flow and is appealing because it is easy to read and learn.

However, Pig is a purpose-built language; it excels at simple data flows, but it is inefficient for implementing non-trivial algorithms.

In Pig, the same example can be implemented as follows:

```
LogLine    = load 'file.logs' as (level, message);
LevelGroup = group LogLine by level;
Result     = foreach LevelGroup generate group, COUNT(LogLine);
store Result into 'Results.txt';
```

Both Pig and Hive support extra functionality through loadable **user-defined functions** (**UDF**) implemented in Java classes.

Cascading is implemented in Java and designed to be expressive and extensible. It is based on the design pattern of *pipelines* that many other technologies follow. The pipeline is inspired from the original chain of responsibility design pattern and allows ordered lists of actions to be executed. It provides a Java-based API for data-processing flows.

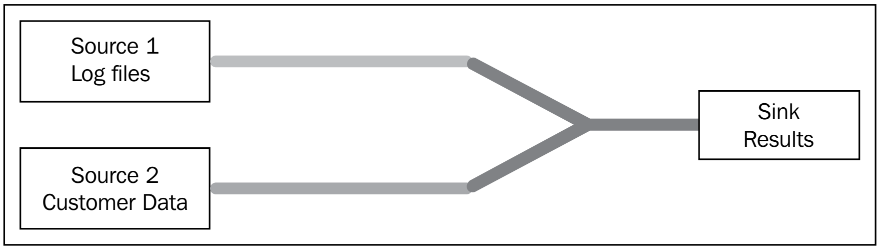
Developers with functional programming backgrounds quickly introduced new domain specific languages that leverage its capabilities. *Scalding*, *Cascalog*, and *PyCascading* are popular implementations on top of Cascading, which are implemented in programming languages such as Scala, Clojure, and Python.

Introducing Cascading

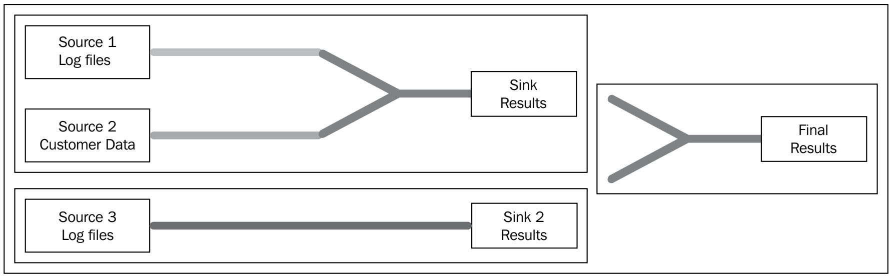
Cascading is an abstraction that empowers us to write efficient MapReduce applications. The API provides a framework for developers who want to think in higher levels and follow **Behavior Driven Development** (**BDD**) and **Test Driven Development** (**TDD**) to provide more value and quality to the business.

Cascading is a mature library that was released as an open source project in early 2008. It is a paradigm shift and introduces new notions that are easier to understand and work with.

In Cascading, we define reusable *pipes* where operations on data are performed. Pipes connect with other pipes to create a *pipeline*. At each end of a pipeline, a *tap* is used. Two types of taps exist: *source*, where input data comes from and *sink*, where the data gets stored.



In the preceding image, three pipes are connected to a pipeline, and two input sources and one output sink complete the flow. A complete pipeline is called a *flow*, and multiple flows bind together to form a *cascade*. In the following diagram, three flows form a cascade:



The Cascading framework translates the pipes, flows, and cascades into sets of map and reduce phases. The flow and cascade planner ensure that no flow or cascade is executed until all its dependencies are satisfied.

The preceding abstraction makes it easy to use a whiteboard to design and discuss data processing logic. We can now work on a productive higher level abstraction and build complex applications for ad targeting, logfile analysis, bioinformatics, machine learning, predictive analytics, web content mining, and for extract, transform and load (ETL) jobs.

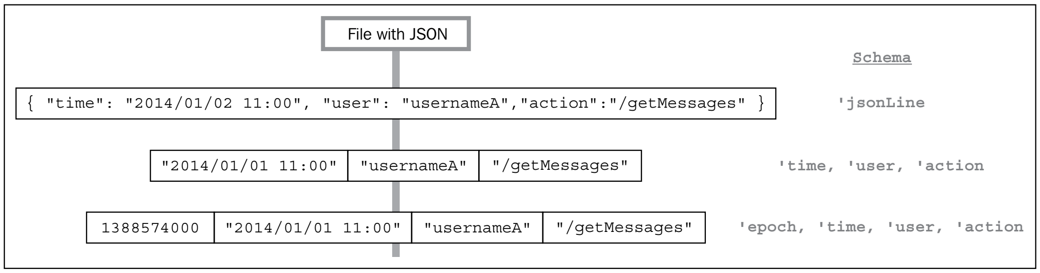
By abstracting from the complexity of key-value pairs and map and reduce phases of MapReduce, Cascading provides an API that so many other technologies are built on.

What happens inside a pipe

Inside a pipe, data flows in small containers called *tuples*. A tuple is like a fixed size ordered list of elements and is a base element in Cascading. Unlike an array or list, a tuple can hold objects with different types.

Tuples stream within pipes. Each specific stream is associated with a *schema*. The schema evolves over time, as at one point in a pipe, a tuple of size one can receive an operation and transform into a tuple of size three.

To illustrate this concept, we will use a JSON transformation job. Each line is originally stored in tuples of size one with a schema: `'jsonLine`. An operation transforms these tuples into new tuples of size three: `'time`, `'user`, and `'action`. Finally, we extract the epoch, and then the pipe contains tuples of size four: `'epoch`, `'time`, `'user`, and `'action`.

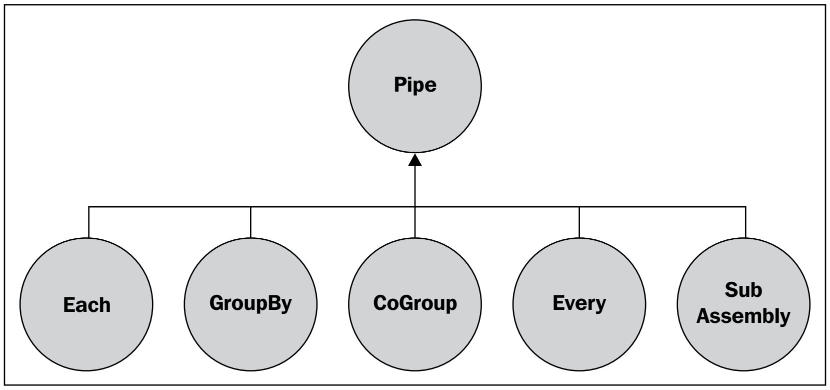


Pipe assemblies

Transformation of tuple streams occurs by applying one of the five types of operations, also called pipe assemblies:

- **Each**: To apply a function or a filter to each tuple
- **GroupBy**: To create a group of tuples by defining which element to use and to merge pipes that contain tuples with similar schemas
- **Every**: To perform aggregations (count, sum) and buffer operations to every group of tuples
- **CoGroup**: To apply SQL type joins, for example, Inner, Outer, Left, or Right joins

- **SubAssembly**: To chain multiple pipe assemblies into a pipe



To implement the pipe for the logfile example with the INFO, WARNING, and ERROR levels, three assemblies are required: The *Each* assembly generates a tuple with two elements (level/message), the *GroupBy* assembly is used in the level, and then the *Every* assembly is applied to perform the count aggregation.

We also need a source tap to read from a file and a sink tap to store the results in another file. Implementing this in Cascading requires 20 lines of code; in Scala/Scalding, the boilerplate is reduced to just the following:

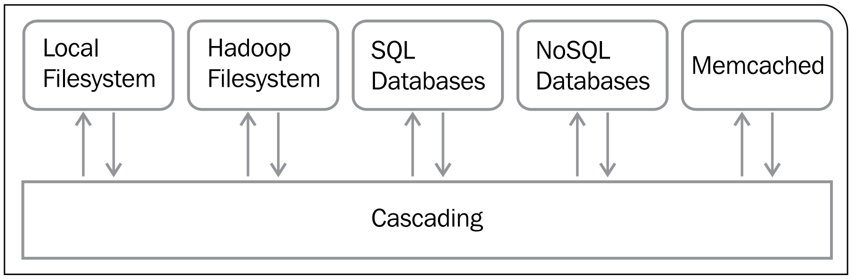
```
TextLine(inputFile)
.mapTo('line->'level,'message) { line:String => tokenize(line) }
.groupBy('level)  { _.size }
.write(Tsv(outputFile))
```

Cascading is the framework that provides the notions and abstractions of tuple streams and pipe assemblies. Scalding is a **domain-specific language** (**DSL**) that specializes in the particular domain of pipeline execution and further minimizes the amount of code that needs to be typed.

Cascading extensions

Cascading offers multiple extensions that can be used as taps to either read from or write data to, such as SQL, NoSQL, and several other distributed technologies that fit nicely with the MapReduce paradigm.

A data processing application, for example, can use taps to collect data from a SQL database and some more from the Hadoop file system. Then, process the data, use a NoSQL database, and complete a machine learning stage. Finally, it can store some resulting data into another SQL database and update a mem-cache application.



Summary

The pipelining abstraction works really well with the Hadoop ecosystem and other state-of-the-art messaging technologies. Cascading provides the blueprints to pipeline for MapReduce. As a framework, it offers a frame to build applications. It comes with several decisions that are already made, and it provides a foundation, including support structures that allow us to get started and deliver results quickly.

Unlike Hive and Pig, where user-defined functionality is separated from the query language, Cascading integrates everything into a single language. Functional and scalable languages follow lightweight, modular, high performance, and testable principles. Scalding combines functional programming with Cascading and brings the best of both worlds by providing an unmatchable way of developing distributed applications.

In the next chapter, we will introduce Scala, set up our environment, and demonstrate the power and expressiveness of Scalding when building MapReduce applications.

2
Get Ready for Scalding

Scalding is a domain-specific language built on top of the capabilities provided by Cascading. It was developed and open-sourced in Twitter and offers a higher level of abstraction by leveraging the power of Scala. In this chapter, we will:

- Get familiar with Scala
- Set up Hadoop and our development environment
- Execute our first Scalding application

Why Scala?

Development has evolved a lot since Java was originally invented 20 years ago. Java, as an imperative language, was designed for the Von-Neumann architecture, where a computer consists of a processor, a memory, and a bus that reads both instructions and data from the memory into the processor. In that architecture, it is safe to store values in variables, and then mutate them by assigning new values. Loop controls are thus normal to use, as shown in the following code:

```
for ( int i=0; i < 1000000;  i++) {
  a=a+1;
}
```

However, over the past decade, hardware engineers have been stressing that the Von-Neumann model is no longer sustainable. Since processors hit physical limitations at high frequencies, engineers look for evolution beyond the single-processor model. Nowadays, manufacturers integrate multiple cores onto a single integrated circuit die—a multiprocessor chip. Similarly, the emergence of cloud computing and Hadoop clusters bring into play another dimension in computing, where resources are distributed across different nodes.

The imperative programming style dictates thinking in terms of time. In distributed programming, we need to think in terms of space: build one block, then another, and then build another block—like building in Lego. When building in space, it is easier to build each block on a different process and parallelize the execution of the required blocks.

Unfortunately, the imperative logic is not compatible with modern distributed systems, cloud applications, and scalable systems. In practice, in parallelized systems, it is unsafe to assign a new value to a variable as this happens in a single node and other nodes are not aware of the local change. For this reason, the simple `for` loop cannot be parallelized into 10 or 100 nodes.

Effective software development techniques and language design evolved over the past decade, and as such, Scala is an advanced scalable language that restricts imperative features and promotes the development of functional and parallelized code blocks. Scala keeps the object-oriented model and provides functional capabilities and other cool features.

Moreover, Scala significantly reduces boilerplate code. Consider a simple Java class, as shown in the following code:

```
public class Person {
  public final String name;
  public final int age;
  Person(String name, int age) {
    this.name=name;
    this.age=age;
  }
}
```

The preceding code can be expressed in Scala with just a single line, as shown:

```
case class Person(val name: String, val age: Int)
```

For distributed computations and parallelism, Scala offers collections. Splitting an array of objects into two separate arrays can be achieved using distributed collections, as shown in the following code:

```
val people: Array[Person]
val (minors,adults) = people partition (_.age < 18)
```

For concurrency, the Actor model provides actors that are similar to objects but inherently concurrent, uses message-passing for communication, and is designed to create an infinite number of new actors. In effect, an actor under stress from a number of asynchronous requests can generate more actors that live in different computers and JVMs and have the network topology updated to achieve dynamic autoscaling through load balancing.

Scala basics

Scala modernizes Java's object-oriented approach while adding in the mix functional programming. It compiles into byte-code, and it can be executed on any Java virtual machine; thus, libraries and classes of Java and Scala communicate seamlessly.

Scala, similar to Java, is a statically typed programming language but can infer type information. It can infer that `t` is a `String` type in the following example:

```
val t = "Text"
```

Semicolons are not required when terminating commands. Variables are declared, with `var` and constants with `val`, and Scala favors immutability, which means that we should try to minimize the usage of variables.

Scala is fully object-oriented and functional. There are no primitives, like float or int only objects such as Int, Long, Double, String, Boolean, Float. Also there is no null.

The Scala equivalent of Java interfaces is called *trait*. Scala allows traits to be partially implemented, that is, it is possible to define default implementations for some methods. A Scala class can extend another class and implement multiple traits using the `with` keyword.

Lists are the most valuable data type in any functional language. Lists are immutable and homogeneous, which means that all elements of a list are of the same type. Lists provide methods and higher-order functions. Some notable ones are as follows:

- `list1 ::: list2`: This is used to append the two lists
- `list.reverse`: This is used to return a list in reverse order
- `list.mkString(string)`: This is used to concatenate the list elements using a string in between them
- `list.map(function)`: This is used to return a new list with a function applied to each element
- `list.filter(predicate)`: This is used to return a list with elements for which the predicate is true
- `list.sortWith(comparisonFunction)`: This is used to return a sorted list using a two parameter comparison function

Understanding the higher order functions of Scala Lists is very beneficial for developing in Scalding. In the next chapter, we will see that Scalding provides implementations of the same functions with similar functionality which work on pipes that contain tuples.

For example, the Scala function flatMap removes one level of nesting by applying a function to each element of each sublist. The same function in Scalding, also removes one level of nesting by iterating through a collection to generate new rows.

Another interesting Scala function is groupBy, which returns a Map of *key* → *values*, where the keys are the results of applying a function to each element of the list, and the values are a List of values so that applying the function to each value yields that key:

```
List("one", "two", "three").groupBy(x => x.length) gives
Map(5 -> List(three), 3 -> List(one, two))
```

Tuples are containers that, unlike an array or a list, can hold objects with different types. A Scala tuple consists of 2 to 22 comma-separated objects enclosed in parentheses and is immutable. To access the nth value in a tuple t, we can use the notation t._n, where n is a literal integer in the range 1 (not 0!) to 22.

To avoid the primitive null that causes many issues, Scala provides the *Option*. Options are parameterized types. For example, one may have an Option[String] type with possible values Some(value) (where the value is of correct type) or None (when no value has been found).

Methods in Scala are public by default and can have private or protected access similar to Java. The syntax is:

```
def methodName(arg1: type, argN:type) { body } // returns Unit
def methodName(arg1: type, .. , argN:type) : returnType = { body }
```

Another aspect of Scala is *function literals*. A function literal (also called anonymous function) is an alternate syntax for defining a function. It is useful to define one-liners and pass a function as an argument to a method. The syntax is (arg1: Type1, ..., argN:TypeN) => expression. Thus, when implementing the function in string.map(function), we can avoid defining an external function by using the following:

```
"aBcDeF".map(x => x toLower) // or for a single parameter, just _
"aBcDeF".map(_.toLower)
```

Scala build tools

There are many build tools we can use to compile and build Scala or Scalding applications. They provide support for mixed Java/Scala projects, dependency management and useful plugins.

We can use **Simple Build Tool** (**sbt**) that allows incremental compilation and is itself implemented in Scala, or Maven, which is popular among developers, is mature, and provides a large range of plugins.

There are other build tools, such as Gradle and buildr, that support Scala, but in this book, we will use Maven and a number of plugins for project dependencies and assembly descriptors due to its high compatibility with continuous integration systems and most developers being familiar with this tool.

Hello World in Scala

To execute a Hello World application in Scala, we will use the latest version of Scala 2.10, the recommended JDK for Hadoop (Oracle JDK 1.6) and Maven. All we need is to add in our build tool a Scala library as a dependency and a plugin that compiles Scala:

```
<dependency>
  <groupId>org.scala-lang</groupId>
  <artifactId>scala-library</artifactId>
  <version>2.10.3</version>
</dependency>
...
<plugin>
  <groupId>net.alchim31.maven</groupId>
  <artifactId>scala-maven-plugin</artifactId>
  <version>3.1.6</version>
</plugin>
```

And the Scala source code:

```
object HelloWorld {
  def main(args: Array[String]) {
    println("Hello, world!")
  }
}
```

The preceding code can be executed with the following:

```
$ mvn package exec:java -Dexec.mainClass=HelloWorld
```

Development editors

Popular IDEs support Scala development through plugins. We should use them to enjoy autocompletion, error highlighting, code refactoring, navigation capabilities, integrated debugger, and much more.

The Scala IDE provides Eclipse prebundled with required plugins and is available at `http://scala-ide.org`. In a project, we need to add the Scala nature or facet in order to be able to execute Scala and Scalding applications.

Installing Hadoop in five minutes

A Linux operating system is the preferred environment for Hadoop. The major Hadoop distributors, MapR, Cloudera, and HortonWorks provide VMs to get started easily with Hadoop and related frameworks.

On Linux, we can also either manually install the required services or install a preconfigured bundle. BentoBox is a zero-configuration bundle that provides a suitable environment for testing and prototyping projects that use HDFS, MapReduce, and HBase with minimal setup time. The installation process requires:

```
$ cd /opt/
$ wget http://archive.kiji.org/tarballs/kiji-bento-dashi-1.4.3-release.
tar.gz
$ tar -zxvf kiji-bento-dashi-1.4.3-release.tar.gz
$ cd kiji-bento-dashi/
$ export KIJI_HOME=/opt/kiji-bento-dashi
$ source $KIJI_HOME/bin/kiji-env.sh
$ export JAVA_HOME=/usr/lib/jvm/j2sdk1.6-oracle/
$ bento start
```

Within a few minutes, we can have all the Hadoop daemons and our HDFS filesystem initiated.

```
Cluster webapps can be visited at these web addresses:
 HDFS NameNode:        http://localhost:50070
 MapReduce JobTracker: http://localhost:50030
 HBase Master:         http://localhost:60010
```

We can now access the web pages of the Hadoop services and run our first Hadoop command in the console to see the contents of the HDFS system using the following command line:

```
$ hadoop fs -ls /
```

After completing Hadoop development, the cluster can be shut down to free up resources with the following:

```
$ bento stop
```

Running our first Scalding job

After adding Scalding as a project dependency, we can now create our first Scalding job as `src/main/scala/WordCountJob.scala`:

```
import com.twitter.scalding._
class WordCountJob(args : Args) extends Job(args) {
  TextLine( args("input") )
  .flatMap('line -> 'word) { line : String =>
    line.toLowerCase.split("\\s+") }
  .groupBy('word) { _.size }
  .write( Tsv( args("output") ) )
}
```

The Scalding code above implements a cascading flow using an input file as source and stores results into another file that is used as an output tap. The pipeline tokenizes lines into words and calculates the number of times each word appears in the input text.

> Find complete project files in the code accompanying this book at `http://github.com/scalding-io/ProgrammingWithScalding`.

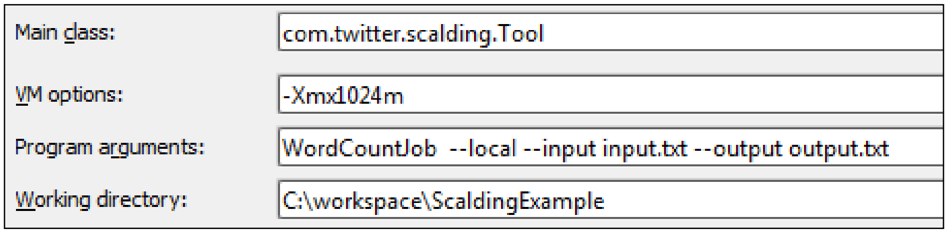
We can create a dummy file to use as input with the following command:

```
$ echo "This is a happy day. A day to remember" > input.txt
```

Scalding supports two types of execution modes: local mode and HDFS mode. The local mode uses the local filesystem and executes the algorithm in-memory, helping us build and debug an application. The HDFS mode is to be used when accessing the HDFS file-system.

To execute a Scalding application, `com.twitter.scalding.Tool` has to be specified as the **Main Class** field followed by the job's fully qualified classpath `WordCountJob`, the `--local` or `--hdfs` mode of execution, job arguments like `--input input.txt --output output.txt`, and possible VM Arguments such as `-Xmx1024m`.

In effect, this translates to the following run-configuration in IntelliJ:



Main class: com.twitter.scalding.Tool

VM options: -Xmx1024m

Program arguments: WordCountJob --local --input input.txt --output output.txt

Working directory: C:\workspace\ScaldingExample

Execute the job `WordCountJob`, and see the results in file `output.txt`.

Submitting a Scalding job in Hadoop

Hadoop MapReduce works by submitting a fat jar (a JAR file that contains all the dependencies and the application code) to the JobTracker. We can generate this jar using the sources accompanying this book with the following command:

```
$ mvn clean package
```

We can test that file by executing it in the Cascading local mode (without using anything from Hadoop) with the following command:

```
$ java -cp target/chapter2-0-jar-with-dependencies.jar com.twitter.
scalding.Tool WordCountJob --local --input input.txt --output output.txt
-Xmx1024m
```

Then, we can start leveraging Hadoop using the command `hadoop jar` to execute the job:

```
$ hadoop jar target/chapter2-0-jar-with-dependencies.jar com.twitter.
scalding.Tool WordCountJob --local --input input.txt --output output.txt
```

Now, we are ready to submit this job into a Hadoop cluster and use the Hadoop Distributed File System. First, we have to create an HDFS folder and push the input data with the help of the following commands:

```
$ echo "This is a happy day. A day to remember" > input.txt
$ hadoop fs -mkdir -p hdfs:///data/input hdfs:///data/output
$ hadoop fs -put input.txt hdfs:///data/input/
```

We can now submit our first Scalding job in the Hadoop cluster using the parameters `--hdfs`, and use the HDFS filesystem for reading input and storing output with the following:

```
$ hadoop jar target/chapter2-0-jar-with-dependencies.jar com.twitter.
scalding.Tool WordCountJob --hdfs --input hdfs:///data/input --output
hdfs:///data/output
```

Note that the parameters for `--input` and `--output` are HDFS folders. The application will read and process all files from the input directory and store all output in the output directory. Hadoop, by default, spawns two map tasks per job, unless the number of files/blocks is more, and in that case, it spawns one map task per file/block.

A map phase reads data and is followed by a reduce phase, where the output is written to disk. Results are stored into part files in the output directory. We can see the contents of the output directory with the following:

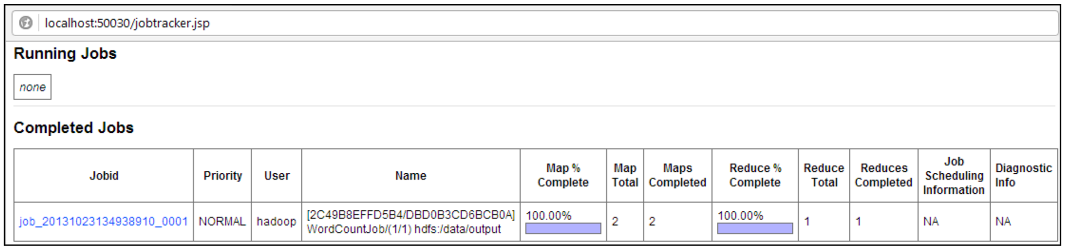
```
$ hadoop fs -ls /data/output
Found 3 items
-rw-r--r-- 3 user group  0 2013-10-21 17:15 /data/output/_SUCCESS
drwxr-xr-x - user group  0 2013-10-21 17:15 /data/output/_logs
-rw-r--r-- 3 user group 46 2013-10-21 17:15 /data/output/part-00000
```

We can see the actual results of the word count job with the following:

```
$ hadoop fs -cat /data/output/part-*
  a         2
  day       2
  happy     1
  is        1
  remember  1
  this      1
  to        1
```

> Execution on the job in a Hadoop cluster will most likely require 20-30 seconds. This is because of the extra overhead of spawning new JVMs to execute the map and reduce phases of the job.

The **JobTracker** reveals detailed information about the submitted job, and it is useful for monitoring and debugging. Note that in the following screenshot the job submitted has been successfully completed. We can also see the number of map tasks and the number of reduce tasks that were executed:



localhost:50030/jobtracker.jsp

Running Jobs

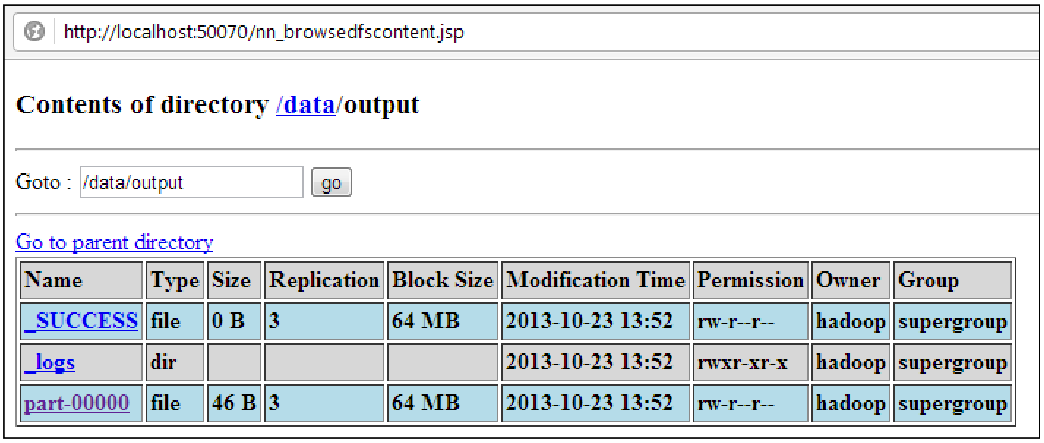
none

Completed Jobs

Jobid	Priority	User	Name	Map % Complete	Map Total	Maps Completed	Reduce % Complete	Reduce Total	Reduces Completed	Job Scheduling Information	Diagnostic Info
job_201310231349389l0_0001	NORMAL	hadoop	[2C49B8EFFD5B4/DBD0B3CD6BCB0A] WordCountJob/(1/1) hdfs:/data/output	100.00%	2	2	100.00%	1	1	NA	NA

At this stage, getting familiar with the JobTracker web interface is strongly encouraged. It provides information about the general job statistics of the Hadoop cluster, running/completed/failed jobs, and the job history. From the web interface, we can follow the job ID link to discover detailed information and logs for each phase of the job.

Another useful application is the **NameNode** web interface that listens at port 50070 by default. It shows the cluster summary, including information about total/remaining capacity and live and dead nodes; in addition, it allows us to browse HDFS and view the contents of its files in the web browser:



http://localhost:50070/nn_browsedfscontent.jsp

Contents of directory /data/output

Goto : /data/output go

Go to parent directory

Name	Type	Size	Replication	Block Size	Modification Time	Permission	Owner	Group
_SUCCESS	file	0 B	3	64 MB	2013-10-23 13:52	rw-r--r--	hadoop	supergroup
_logs	dir				2013-10-23 13:52	rwxr-xr-x	hadoop	supergroup
part-00000	file	46 B	3	64 MB	2013-10-23 13:52	rw-r--r--	hadoop	supergroup

Note that in the preceding screenshot the block size is **64 MB** and data is replicated across three different nodes of the cluster. The owner, group, and permissions of folders and files are also visible. We can also use this web interface to access the files of a particular job.

Summary

This chapter introduced Scala, the reasons for its recent popularity, and how Scala and Scalding are associated. Build tools and IDEs were briefly presented, and then we executed our first Scala application.

A Hadoop environment for development was set up, and we moved to running a Scalding application in many modes while discussing in detail the commands involved. After executing a distributed MapReduce application in Scalding for the first time, we reviewed the web interfaces of the Hadoop applications.

In the next chapter, we will present an overview of Scalding and its capabilities, using trivial examples.

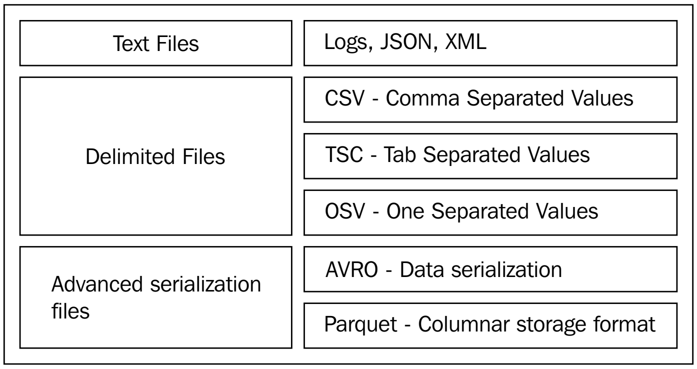
3
Scalding by Example

This chapter presents how to read and write local and **Hadoop Distributed File System** (**HDFS**) files with Scalding. It introduces the complete Scalding core capabilities through the Fields API and serves as a reference to look up how the Scalding commands can be used. In this chapter, we will cover:

- Map-like operations
- Join operations
- Pipe operations
- Grouping and reducing operations
- Composite operations

Reading and writing files

Data lives mostly in files stored in the filesystem in semi-structured text files, structured delimited files, or more sophisticated formats such as Avro and Parquet. Logfiles, SQL exports, JSON, XML, and any type of file can be processed with Scalding.



Text Files	Logs, JSON, XML
Delimited Files	CSV - Comma Separated Values
	TSC - Tab Separated Values
	OSV - One Separated Values
Advanced serialization files	AVRO - Data serialization
	Parquet - Columnar storage format

Scalding is capable of reading and writing many file formats, which are:

- The TextLine format is used to read and write raw text files, and it returns tuples with two fields named by default: *offset* and *line*. These values are inherited from Hadoop. After reading a text file, we usually parse with regular expressions to apply a schema to the data.
- Delimited files such as **Tab Separated Values** (**TSV**), **Comma Separated Values** (**CSV**), and **One Separated Values** (**OSV**), with the latter commonly used in Pig and Hive, are already structured text files, and thus, easier to work with.
- Advanced serialization files such as Avro, Parquet, Thrift, and protocol buffers offer their own capabilities. Avro, for example, is a data-serialization format that stores both schema and data in files. Parquet is a columnar storage format highly efficient for large-scale queries such as scanning particular columns within a table or performing aggregation operations on values of specific columns.

> Examples, including reading and writing advanced serialization files, can be found in the code accompanying the book and at `http://github.com/scalding-io/ProgrammingWithScalding`.

The Scalding Fields API follows the concept that data lives in named columns (fields) that can be accessed using symbols, like `'quantity`.

So, in Scalding, reading delimited files from TSV, CSV, or OSV requires a single line and associates each column of data to a field, according to the provided schema:

```
Tsv("data.tsv",('productID,'price,'quantity)).read
Csv("data.csv", "," , ('productID,'price,'quantity)).read
Osv("data.osv", ('productID,'price,'quantity)).read
```

Reading semi-structured text files requires a single line:

```
val pipe = TextLine("data.log").read
```

When we use `TextLine` to read a file, the pipe returned by the `.read` operation contains the fields `'offset` and `'line`. The `offset` contains the line number and the `line` contains the contents of the line.

Writing data in delimited files also requires line of code, as shown below:

```
pipe.write( Tsv("results.tsv") )
pipe.write( Csv("results.csv") )
pipe.write( Osv("results.osv") )
```

Writing to `TextLine` will concatenate all the tuples with a *Tab* delimiter before writing out each line:

```
pipe.write(TextLine("result.out"))
```

Best practices to read and write files

A flexible approach to reading files is:

```
val inputSchema = List('productID, 'price, 'quantity)
val data = Csv( args("input") , "," , inputSchema ).read
```

In the preceding code, a number of best practices are presented, such as:

- We define the schema as a list of fields to have more readable code, less code duplication, and overcome the Scala tuple limitation to a maximum of 22 elements
- We define that the location of the input data is provided at runtime using the argument `--input`
- We store the pipe returned by the operation `.read` in `data`, which is a fixed unmodifiable value, in order to be able reuse it in an example as follows:

  ```
  val inputSchema = List('productID, 'price, 'quantity)
  val data = Csv( args("input") , "," , inputSchema ).read
  data.write(Tsv(args("output.tsv"))
  data.write(Osv(args("output.osv"))
  ```

Using these techniques, the input and output can be specified at execution time as follows:

```
$ hadoop jar target/chapter3-0-jar-with-dependencies.jar
 com.twitter.scalding.Tool FirstScaldingJob --hdfs --input
 hdfs:///input --output.tsv hdfs:///tsvoutput --output.osv
 hdfs:///osvoutput
```

TextLine parsing

It is very common for data to exist in custom semi-structured formats such as:

```
[product15] 10.0 11
[product16] 2.5 29
```

When reading such text files, our first task is to transform semi-structured data into structured data and associate a schema, as shown in the following code:

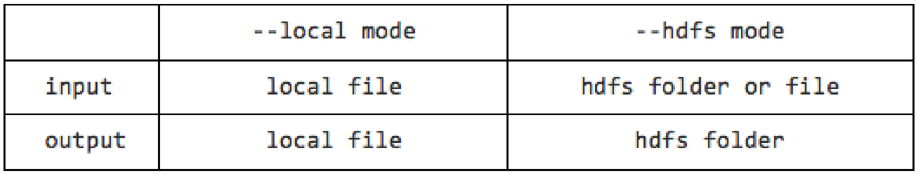
```
import com.twitter.scalding._
import scala.util.matching.Regex

class TextParsingExample (args: Args) extends Job(args) {

  val pipe = TextLine ( args("input") ).read
  .mapTo ('line -> ('productID, 'price, 'quantity))
    {  x: String => parseLine(x)  }
  .write ( Tsv ( args("output") ) )

  val pattern = new Regex("(?<=\\[)[^]]+(?=\\])")
  def parseLine(s : String) = {
    ( pattern findFirstIn s get ,    // 1st tuple element
      s.split(" ").toList.get(1),    // 2nd tuple element
      s.split(" ").toList.get(2)  ) // 3rd tuple element
  }
}
```

In the preceding code, we ignore the `'offset` and parse the `'line` with a regular expression to yield tuples with the three elements `'productID`, `'price`, and `'quantity`, and store results in tab-delimited files.

Executing in the local and Hadoop modes

When implementing a Scalding application, the local execution mode provides better debugging and quick feedback by reading and writing files in the local filesystem. To use HDFS to read or write data, we use the HDFS mode. To execute the application on a distributed Hadoop cluster, we need to use the `hadoop jar` command and the HDFS mode.

The execution mode is defined at execution time with parameters `--local` or `--hdfs`. For the local mode, paths to local files have to be used. For the HDFS mode, input has to be an HDFS file or folder, and output is always an HDFS folder.



	--local mode	--hdfs mode
input	local file	hdfs folder or file
output	local file	hdfs folder

When using the HDFS mode, remember the following:

- Output is stored in an HDFS folder, which is created if it does not exist or truncated if it exists. Note that any existing data in the output folder will be lost.
- Input can be either a set of files or an entire folder.
- Input can use pattern-matching to load data from multiple HDFS files or folders.

So, for example, when an HDFS directory structure `/dataset/YYYY/mm/dd/` is used, data can be read from `hdfs:///datasets/2014/01/0*/` or `hdfs:///datasets/*/01/01/`.

Understanding the core capabilities of Scalding

Scalding provides a rich set of core operations to perform data transformations. Map-like operations apply a function to each tuple in the pipe. Join operations can join data from multiple pipes. Pipe operations allow us to concatenate or debug pipes. Grouping/Reducing operations group related data together. Also, for data that has been grouped, there is a rich set of group operations.

Map-like operations

These operations are internally translated into map phases of MapReduce and apply a function to every row of data. The syntax of the map operation is:

```
pipe.map(existingFields -> additionalFields) { function }
```

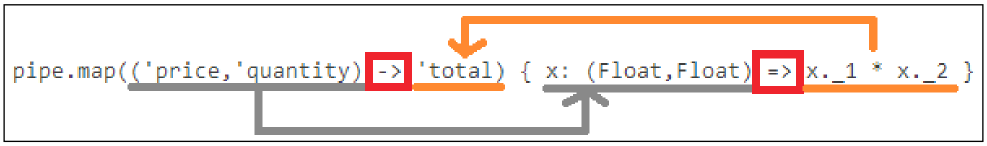
The `map` operation uses some of the existing fields of a pipe as input and creates a pipe with additional fields by applying a function to the elements of the input. In the following example, a new field `'priceWithVAT` is introduced:

```
pipe.map('price -> 'priceWithVAT) { price: Double => price*1.20 }
```

Operations can be executed to multiple fields at a time, as shown in the following code:

```
pipe.map(('price,'quantity)->'total)
  { x:(Float,Float)=>x._1 * x._2 }
```

The input tuple `x: (Float, Float)` receives the values of `'price` and `'quantity` for every row of the input, and it maps the result of the multiplication `x._1 * x._2` into an additional field `'total`.



```
pipe.map(('price,'quantity) -> 'total) { x: (Float,Float) => x._1 * x._2 }
```

Access to each element of the tuple is achieved using `._1` and `._2`. The `._` is a method that returns the value at a specific index of the tuple.

Another variation of the map operation is `mapTo`, as shown:

```
pipe.mapTo(existingFields -> resultingFields) { function }
```

This operation is similar to the `map` operation, but it generates a pipe holding only the resulting fields. This means that all existing fields are discarded from the pipe. The following example creates a pipe with a single column of data:

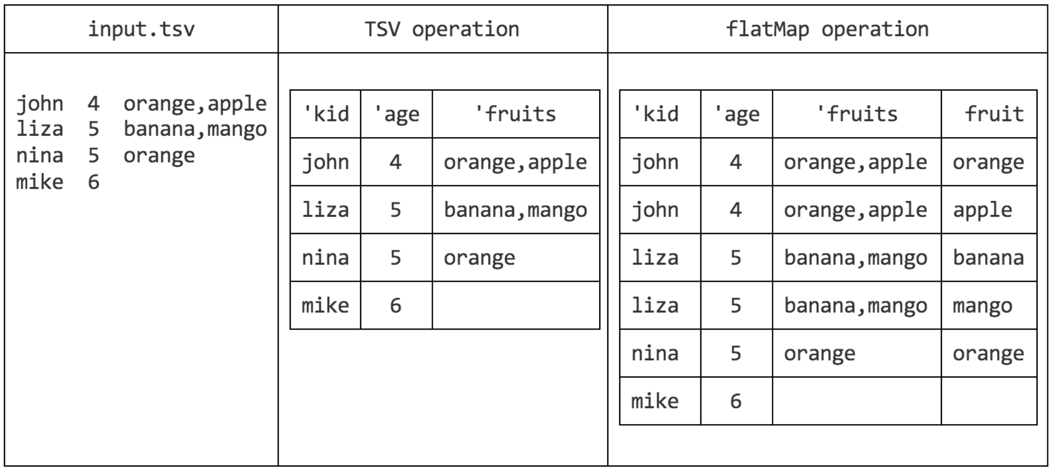
```
pipe.mapTo(('productId, 'price, 'quantity) -> 'productsValue)
  { x:(String,Double,Int) => x._2 * x._3 }
```

This operation is especially useful for transformations. For example, when transforming some fields into a JSON line, we can discard the original values of the input.

Similar to `map` and `mapTo`, an operation `flatMap` and `flatMapTo` exist. The syntax of `flatMap` is as follows:

```
pipe.flatMap(existingFields -> additionalFields) { function }
```

We use this operation to apply an operation to a field to generate a list of tuples, and then create new rows of data by flattening the list. As an example, imagine having information about kids liking specific types of fruits as a comma-separated string, and we want to compute some statistics on the fruits.



input.tsv

```
john  4  orange,apple
liza  5  banana,mango
nina  5  orange
mike  6
```

TSV operation

'kid	'age	'fruits
john	4	orange,apple
liza	5	banana,mango
nina	5	orange
mike	6	

flatMap operation

'kid	'age	'fruits	fruit
john	4	orange,apple	orange
john	4	orange,apple	apple
liza	5	banana,mango	banana
liza	5	banana,mango	mango
nina	5	orange	orange
mike	6		

In the preceding diagram, the original data is structured in three columns/fields. The `flatMap` operation splits the values in the column called **'fruits** into a list of values and generates multiple rows for each element of that list.

```
val kidsPipe = Tsv(args("input"), ('kid, 'age, 'fruits) ).read
.flatMap('fruits -> 'fruit) { text : String => text.split(",") }
```

The syntax of `flatMapTo` is as follows:

```
pipe.flatMapTo(existingFields -> resultingFields) { function }
```

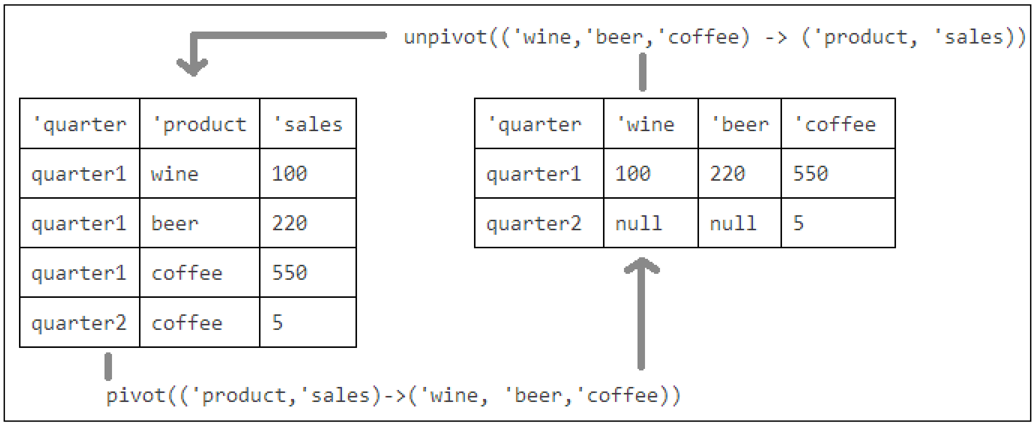
This operation is similar to `flatMap`, but the generated pipe contains only the resulting fields.

The syntax of the `unpivot` operation is as follows:

```
pipe.unpivot(fields -> columns)
```

It converts input from a column-based to a row-based representation. The pivot operation is a group operation and is presented later in this chapter. The following code example with the accompanying image illustrates the concept of the column- and row-based representations:

```
pipe.unpivot(('wine,'beer,'coffee) -> ('product, 'sales))
```



```
unpivot(('wine,'beer,'coffee) -> ('product, 'sales))
```

'quarter	'product	'sales
quarter1	wine	100
quarter1	beer	220
quarter1	coffee	550
quarter2	coffee	5

'quarter	'wine	'beer	'coffee
quarter1	100	220	550
quarter2	null	null	5

```
pivot(('product,'sales)->('wine, 'beer,'coffee))
```

The syntax of the project operation is as follows:

```
pipe.project(fields)
```

It creates a new pipe containing only the fields specified. In the following example, applying project on a pipe results in a pipe that contains a single column of data.

```
kidsPipe.project('fruit)
```

The syntax of the discard operation is as follows:

```
pipe.discard(fields)
```

It creates a new pipe that contains all the fields of the original pipe, excepting the fields specified. It is the opposite of project. In the following example, only the column 'fruit will remain in the pipe:

```
kidsPipe.discard('kid,'age,'fruits)
```

The syntax of the insert operation is as follows:

```
pipe.insert(field, value)
```

It creates a new pipe containing the data of the original pipe and an additional column holding the static value `0.20` in all the rows, as shown:

```
pipe.insert('vat, 0.20)
```

The syntax of the `limit` operation is as follows:

```
pipe.limit(number)
```

The resulting pipe contains only a fixed number of lines. In the following example, only one hundred rows of data will exist in the new pipe:

```
pipe.limit(100)
```

The syntax of the `filter` operation is as follows:

```
pipe.filter(fields) { function }
```

It filters out lines for which the function is false. In the following example, a new pipe will be created to contain only kids that like oranges:

```
val kidsLikeOranges = kidsPipe.filter('fruit)
  { f:String => f == "orange" }
```

The syntax of the `sample` operation is as follows:

```
pipe.sample(percentage)
```

It samples a percentage of lines from the original pipe. In the following example, 10 percent of the original data will exist in the new pipe:

```
pipe.sample(0.10)
```

The syntax of the `pack` operation is as follows:

```
pipe.pack [Type] (fields -> object)
```

Multiple fields can be packed into a single object using Java reflection. This works for objects which have a default constructor that takes no arguments. In the example, a class called `Product` is required with the three attributes `productID`, `price`, and `quantity` and the relevant setters `setProductID`, `setPrice`, and `setQuantity`.

```
val p = productPipe.pack [Product] ( ('productID,'price,'quantity)
->'product)
```

The syntax of the `unpack` operation is as follows:

```
pipe.unpack [Type] (object -> fields)
```

Contents of serialized objects can be unpacked into multiple fields. Packing and unpacking uses reflection in **Plain Old Java Objects** (**POJO**), standard Thrift and Protobuf generated classes, and Scala case classes. To work without reflection, `TuplePacker` and `TupleUnpacker` abstract classes can be implemented. In effect, `pack` and `unpack` group or ungroup fields in objects, as shown in the following code:

```
val p = products.unpack[Product] ('product ->('productId, 'price,
'quantity) )
```

All the preceding map-like operations are translated by Scalding that uses the Cascading framework into sets of map tasks.

Join operations

Join operations merge two pipes on a specified set of keys, similar to SQL joins. Scalding provides all the expected joining modes: inner, outer, left, and right, and it utilizes the `CoGroup` pipe assembly of Cascading, which is implemented as a single MapReduce job.

The relative size of the data to be joined allows Cascading to execute the join using the most optimized and efficient algorithm. Thus, three types of joins are provided as follows:

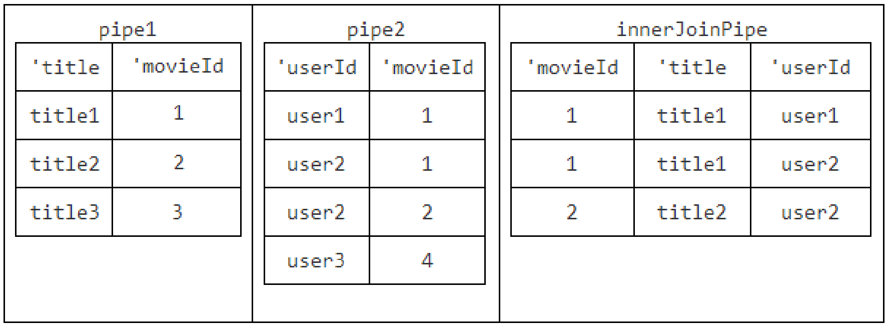
- `joinWithSmaller`: This is used when the pipe on the right is smaller than the pipe on the left.
- `joinWithLarger`: This is used when the pipe on the right is larger than the pipe on the left.
- `joinWithTiny`: This is the most optimized type of join, implemented as a single map job. Use when the pipe on the right holds a tiny amount of data (that is, a few megabytes), and thus data can be distributed to all the nodes that contain the larger pipe. Storing the tiny data in memory allows a single map to perform the join.

The syntax of the `joinWithSmaller` operation is as follows:

```
pipe1.joinWithSmaller(pipe1key -> pipe2key, pipe2,optionalJoiner)
```

The following example performs the default inner join using `'movieId` as a join key from `pipe1` and `pipe2`. Inner join means that the resulting pipe contains all possible combinations of rows, where the keys of each pipe are the same. In the following diagram, the movie `title3` does not exist in the results because of the inner nature of the join.

```
val innerJoinPipe = pipe1.joinWithSmaller('movieId -> 'movieId, pipe2)
```



pipe1

'title	'movieId
title1	1
title2	2
title3	3

pipe2

'userId	'movieId
user1	1
user2	1
user2	2
user3	4

innerJoinPipe

'movieId	'title	'userId
1	title1	user1
1	title1	user2
2	title2	user2

To perform a left, right, or outer join, we need to specify the type of `optionalJoiner`. In this case, join keys must be *disjoint*. This means that we can no longer join two pipes that contain the fields `'movieId` and `'movieId`. To overcome this, we need to rename the fields that compose the join key in one of the pipes, as shown:

```
val pipe2Renamed = pipe2.rename('movieId -> 'movieId_)
```

Once the pipes to be joined do not have any join fields in common, we can specify the type of joiner to be used, as shown in the following code:

```
pipe1.joinWithSmaller('movieId -> 'movieId_, pipe2Renamed, joiner=new
LeftJoin)
pipe1.joinWithSmaller('movieId -> 'movieId_, pipe2Renamed, joiner=new
RightJoin)
pipe1.joinWithSmaller('movieId -> 'movieId_, pipe2Renamed, joiner=new
OuterJoin)
```

The *left join* creates a pipe that contains all the data an inner join will produce and also all the data of the pipe on the left (`pipe1`) that was not joined.

The *right join* creates a pipe that contains all the data an inner join will produce and also all the data of the pipe on the right (`pipe2Renamed`) that was not joined.

Finally, the *outer join* creates a pipe that contains all the data an inner join will produce and retains all the data from both pipes that did not match in the join.

Pipe operations

Scalding pipes provide a set of operations useful for integration and debugging.

The syntax of the `++` operation is as follows:

```
pipe1 ++ pipe2
```

It creates a new pipe that is a union of two or more pipes that have the same fields.

The syntax of the `name` operation is as follows:

```
pipe.name("new name")
```

It gives a new name to the pipe. Associating a name to a pipe is useful for visualizing pipelines in a tool, and it is presented in a later chapter.

The syntax of the `debug` operation is as follows:

```
pipe.debug
```

It provides pipe debugging information by printing out some pipe content on the screen.

The syntax of the `addTrap` operation is as follows:

```
pipe.addTrap(sink)
```

It adds a trap that captures tuples that cause exceptions in any of the pipe operations and stores them in the filesystem. The actual exceptions are not captured, and there can be only one trap in a pipe.

```
pipe.addTrap(Tsv("/project/error_folder/")
```

The syntax of the `rename` operation is as follows:

```
pipe.rename(fields -> fields)
```

It changes the name of the fields. In the following example, the field `'kid` is renamed to `'kidName`, and `'fruit` is renamed to `'favoriteFruit`.

```
kidsPipe.rename( ('kid,'fruit) -> ('kidName,'favoriteFruit) )
```

Grouping/reducing functions

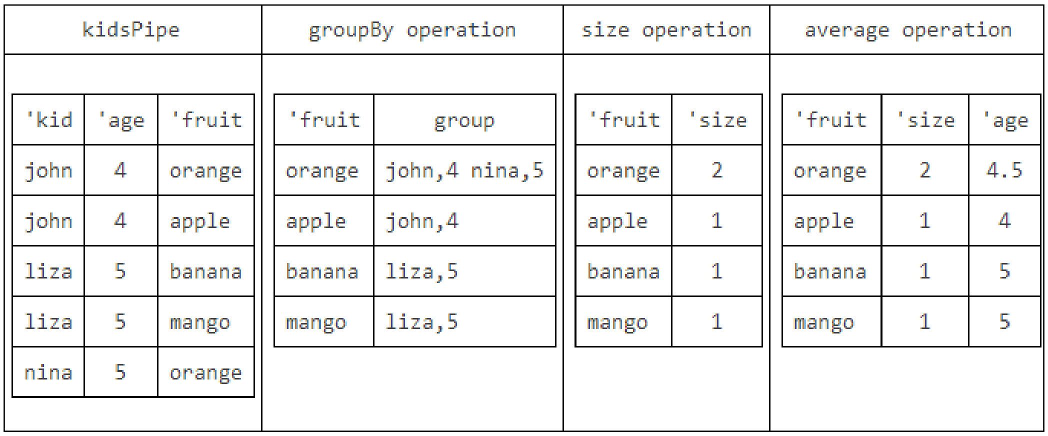
Grouping/reducing operations are internally translated into MapReduce pairs and operate over groups of rows in a pipe.

The syntax of the `groupBy` operation is:

```
pipe.groupBy( fields ) { group => group.operation1.operation2 }
```

It groups the values in the pipe by the specified set of fields, known as the key. The purpose of grouping by key is to apply one or more group operations on the group. In the following example, all rows with the same value of fruit are grouped, and then the operations `size` and `average` of age are calculated:

```
val data=kidsPipe.groupBy('fruit){group=>group.size.average('age)}
```



kidsPipe

'kid	'age	'fruit
john	4	orange
john	4	apple
liza	5	banana
liza	5	mango
nina	5	orange

groupBy operation

'fruit	group
orange	john,4 nina,5
apple	john,4
banana	liza,5
mango	liza,5

size operation

'fruit	'size
orange	2
apple	1
banana	1
mango	1

average operation

'fruit	'size	'age
orange	2	4.5
apple	1	4
banana	1	5
mango	1	5

This operation is highly parallelized and distributes data to a number of nodes in a Hadoop cluster. The number of reducers is calculated by the framework, but we can also control it by manually specifying the exact number of reducers using the `.reducers` operation.

The syntax of the `groupAll` operation is:

```
pipe.groupAll { group => group.groupOperation1.groupOperation2 }
```

It groups the entire pipe into a single group. This is *not* a parallelized operation as a single reducer is used resulting in us utilizing a single physical node of a Hadoop cluster for that operation.

One or more operations can be applied to each group. In the following example, a pipe is sorted based on `'age` and `'size`:

```
val sortedKids=kidsPipe.groupAll{group =>group.sortBy('age,'size)}
```

To highlight the importance of the parallelism of `groupAll` and `groupBy`, consider the ensuing examples. In the first example, we want to sort all of the data alphabetically based on surnames. Use the `groupAll` operator so that all the data is grouped into a single node of a Hadoop cluster and the comparison function is correctly applied. In the second example, we first want to sort all males and then all females based on surname. In this case, `groupBy` is the operator to use in order to distribute work in two nodes of a Hadoop cluster, as shown in the following code:

```
val sortAll = pipe.groupAll { group => group.sortBy('surname) }
val paralleSort = pipe.groupBy('gender) { group => group.
 sortBy('surname) }
```

Using parallelism is beneficial in real use cases. It makes sense to use `groupBy` to distribute *comedies* in a node, *thrillers* on another node, and manage to distribute computations over multiple Hadoop nodes, if we are working on a movie recommendation algorithm.

Operations on groups

Operations `groupAll` and `groupBy` are essential building blocks of Scalding applications, and they generate groups. `groupAll` generates a single group containing all the available tuples. `groupBy` generates *m* number of groups, where *m* is the number of unique keys in the data.

For example, if `groupBy('color)` is executed and three unique colors exist in the data, then three groups will be generated. Once grouping is achieved, a number of group operations can be applied to them.

The first seven group operations average, count, min, max, sum, size, and sizeAveStdev are useful to extract statistics from data, and their syntax is as follows:

```
group.average(field -> newField)
group.count(field -> newField) { function }
group.min(field -> newField)
group.max(field -> newField)
group.sum(field -> newField)
group.size(newField)
group.sizeAveStdev(field -> sizeField,averageField, stdField)
```

We can also apply multiple group operations on the same group. To calculate, for example, the minimum and maximum age, the average and total age, the total population of kids liking a particular fruit, and a count of how many kids of age four like the particular fruit, we can use the following code:

```
val fruitStats = kidsPipe.groupBy('fruit) { group => group
    .min('age -> 'minAge)
    .max('age -> 'maxAge)
    .average('age -> 'averageAge)
    .sum[Int]('age->'totalAge)
    .size('totalKids)
    .count('age -> 'age4){ x:Int => x == 4 }
    .sizeAveStdev('age -> ('totalAges, 'meanAge, 'stdevAge))
}
```

The sizeAveStdev group operation calculates the size, average, and standard deviation in a single command.

The syntax of the mkString operation is as follows:

```
group.mkString(field -> newField, separator)
```

It concatenates the contents of a field in the group into a string using the separator we specify. In the following example, all names of kids are concatenated into a hyphen-separated string:

```
val allNames=kidsPipe.groupAll
  { group=>group.mkString('kid->'names, "-") }
```

The syntax of the `toList` group operation is as follows:

```
group.toList(field)
```

It transforms a column of the group into a list, while skipping any null items, shown as follows.

```
val nameList = kidsPipe.groupAll
  { group => group.toList('kid ->'nameList) }
```

> Among group operations, `mkString` and `toList` require all the data in memory. Thus, using `mkString` on a one billion row data set would result into a heap overflow exception. Other group operations iterate through the tuples, and thus, can be executed on a billion row data set.

The syntax of the `sortBy` group operation is as follows:

```
group.sortBy(field)   or   group.sortBy(field).reverse
```

It sorts every group, ascending or descending (when using the `reverse` operator).

```
val sortedGroup = kidsPipe.groupBy('fruit)
  { group => group.sortBy('age) }
```

Seven operators are provided to allow us to get a specific number of tuples from a group: `head`, `last`, `take`, `takeWhile`, `drop`, `sortWithTake`, and `sortedReverseTake`.

So, the syntax of the `head` group operation is as follows:

```
group.head(fields)
```

It returns the first element of a group. We usually use this when we sort a group and want to get the very first element of that group.

The syntax of the `last` group operation is as follows:

```
group.last(fields)
```

It returns the last element of a group. We usually use this when we sort a group and want to get the last element of that group.

The syntax of the `take` group operation is as follows:

```
group.take(number)
```

It takes a specific number of elements from the beginning of each group. The following code example will sort each gender by age and take the first 10 elements:

```
pipe.groupBy('gender) { group => group.sortBy('age).take(10) }
```

The syntax of the `takeWhile` group operation is as follows:

```
group.takeWhile ( fields ) { function }
```

It takes elements from a group until the function returns false. It stops taking when the first false is returned. The following example will start taking all lines from the pipe until the condition `age <= 4` is broken:

```
val takeGroup = kidsPipe.groupAll
  { group => group.takeWhile('age) { age:Int => age <= 4  }  }
```

The syntax of the `drop` group operation is as follows:

```
group.drop(number)
```

It drops a specific number of elements from the beginning of each group.

The syntax of the `sortWithTake` group operation is as follows:

```
group.sortWithTake(fields -> result_field, number)
```

It sorts using the comparison function provided and then takes the first `number` of items. This operation is more efficient than first sorting and then taking because it uses a small buffer of size `number`, allowing the process to be completed on a map phase.

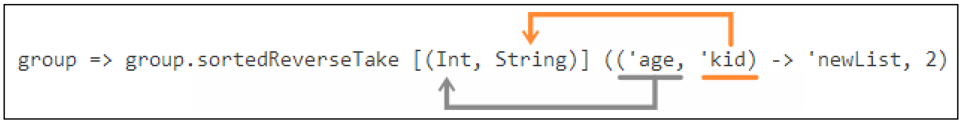
```
group => group.sortWithTake ( (('age,'kid) -> 'newList), 2 ) {
(prev: (Int,String), next:(Int,String)) => (prev._1 > next._1) }
```

The syntax of the `sortedReverseTake` group operation is as follows:

```
group.sortedReverseTake [Types] (fields -> resultFields, number)
```

It sorts data in reverse (descending) order and returns a list containing the top `number` of tuples. In the following example, kids are sorted first by oldest to youngest in age and then by name in descending order. The types of fields used in ordering need to be provided so that the correct comparator is used.

```
group => group.sortedReverseTake [(Int,String)] (('age, 'kid)-
>'newList, 2)
```



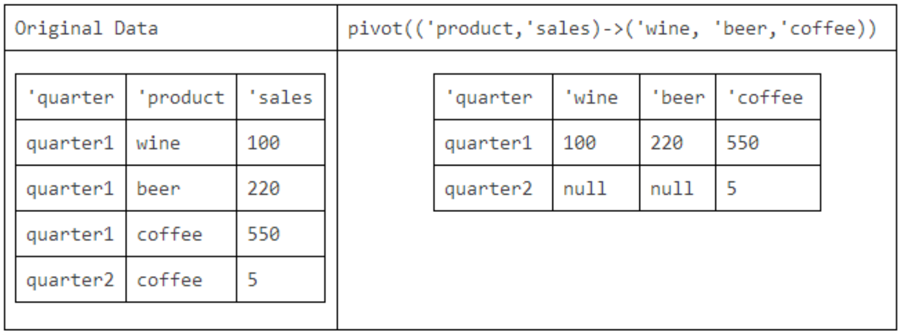
```
group => group.sortedReverseTake [(Int, String)] (('age, 'kid) -> 'newList, 2)
```

The syntax of the `pivot` group operation is as follows:

```
group.pivot (fields -> valuesIntoColumns, defaultValue)
```

`pivot` and `unpivot` are similar to SQL and Excel functions: `pivot` changes data from a row-based representation to a column-based one, and `unpivot` performs the reverse operation.

```
pipe.groupBy('quarter) {
  group => group.pivot(('product,'sales)->('wine, 'beer,'coffee)) }
```



Original Data

'quarter	'product	'sales
quarter1	wine	100
quarter1	beer	220
quarter1	coffee	550
quarter2	coffee	5

pivot(('product,'sales)->('wine, 'beer,'coffee))

'quarter	'wine	'beer	'coffee
quarter1	100	220	550
quarter2	null	null	5

The preceding pivot operation fills in blanks with the value `null`. To define a specific value to be used instead of `null`, the `defaultValue` can be specified.

The syntax of the `reducers` group operation is as follows:

```
group.reducers(number)
```

> Scalding applications are translated into a number of map and reduce tasks of the MapReduce framework by Cascading. This operation is useful to control the number of reducers in the reduce phases to optimize a job.

The flow planner calculates the number of reducers to be used for optimal performance for `groupBy` operations (`groupAll` always results in a single reducer). Use the `reducers` group operation to override the number of reducers to be used.

The syntax of the `reduce` group operation is:

```
group.reduce(field) {function}
```

Reduce groups on a map phase. In the following example, the volume of boxes is calculated by multiplying the lengths of each dimension: width, height, and depth:

```
group => group.reduce('length -> 'volume)
  { (volumeSoFar : Int, length : Int) => volumeSoFar * length }
```

The syntax of the `foldLeft` group operation is as follows:

```
group.foldLeft(fields -> result_field) (initial_value) {function}
```

The `foldLeft` group completes work in the reduce phase of the MapReduce framework. An initial value has to be specified for the operation, and a final value is created for that group.

For example, let us imagine that we have customers who pay either for `service-A` or `service-B`. The data structure is as follows:

customer1	**service-A**	**false**
customer1	**service-B**	**false**
customer2	**service-A**	**true**
customer2	**service-B**	**false**

If we want to calculate whether each customer is paying, we can instantiate `foldLeft` with the default value `false`, as shown in the following code:

```
pipe.groupBy('customer) {
  group => group.foldLeft('is_paying -> 'is_paying)(false) {
    (previous: Boolean, current: Boolean) => previous || current
  }
}
```

The `previous || current` condition will compare every previous value with the new one and yield `true` only if the customer is paying for at least one service.

The syntax of the `dot` group operation is as follows:

```
group.dot [Type] (left, right, result)
```

It calculates the dot product of a group by multiplying the left column with the right and storing the results to `result` for each group. If we imagine a scenario of colored postal cards of `x` width and `y` height, the following example calculates the total area each color of cards can occupy:

```
pipe.groupBy('square) { group => group.dot[Int]('x,'y,'x_dot_y) }
```

The syntax of the `histogram` group operation is as follows:

```
group.histogram (field -> histogram, binWidth)
```

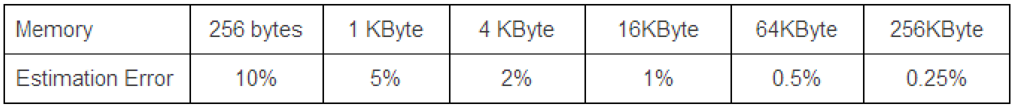
It returns the histogram of a group as a Scalding `Histogram` object. This object allows the extraction of the three quartiles, coefficient of dispersion, inner quartile range, the Lorenz Curve, and other values used mostly by data scientists. In the following example, the median and first and third quartile of the histogram for the image sizes uploaded to a famous photo sharing application are calculated:

```
pipe.groupBy('date) { group => group.histogram(('imageSize,
'histogram)) }
.map('histogram -> ('q1, 'median, 'q3)) {
     x:Histogram => (x.q1 , x.median , x.q3)
}
```

The syntax of the `hyperLogLog` group operation is as follows:

```
group.hyperLogLog [Type] (field -> newField, errPercentage)
```

`HyperLogLog` is an algorithm able to make accurate cardinality estimates using small fixed memory. An accurate count of unique element scales with *O(log(n)), on the other hand*. `HyperLogLog` uses a small fixed amount of memory and provides approximate results with an estimation.



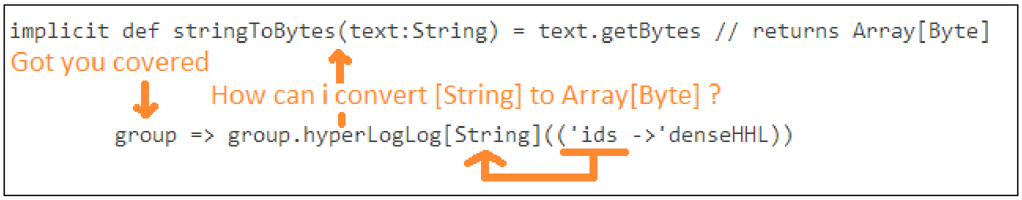
Memory	256 bytes	1 KByte	4 KByte	16KByte	64KByte	256KByte
Estimation Error	10%	5%	2%	1%	0.5%	0.25%

To find further information, on how `HyperLogLog` algorithm works, check and read the responses to the question.

The following code is capable of counting the size of unique elements in a billion row dataset while using just 16 kilobytes of memory:

```
implicit def stringToBytes(text: String) = text.getBytes
val errPercent = 1D // 1% -> 16kB buffer
pipe.groupAll {
   group => group.hyperLogLog[String]
     (('ids ->'denseHHL),errPercent)
}.mapTo('denseHHL -> 'approximateSize) {
   x: DenseHLL => x.approximateSize.estimate }
```

HyperLogLog expects input in the Array[Byte] format. However, the value at 'ids is cast into a String. The implicit function stringToBytes provides the required conversion functionality String -> Array[Byte].



```
implicit def stringToBytes(text:String) = text.getBytes // returns Array[Byte]

        group => group.hyperLogLog[String](('ids ->'denseHHL))
```

Composite operations

Finally, Scalding provides some operations that are useful in many common use cases. The following list of operations internally uses the core operations presented already, but they are actually handy to be in place. The syntax of the unique operation is as follows:

```
pipe.unique(fields)
```

It keeps unique elements of the specified fields. In the following example, the new pipe will contain only the unique fruits, and all other fields are discarded. Internally, the preceding command uses a group operation, and thus, jobs can be optimized if this is applied during a groupBy operation.

```
val uniqueFruits = kidsPipe.unique('fruit)
```

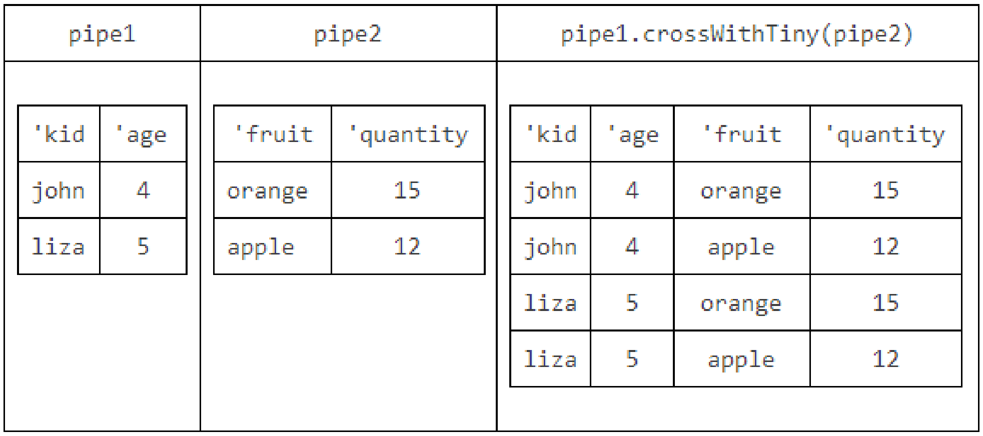
> The unique operator is actually implemented using the groupBy, size, and project operators, as shown in the following code.

```
groupBy(fields) { group => group.size } .project(fields)
```

The syntax of the crossWithTiny operation is as follows:

```
pipe1.crossWithTiny(pipe2)
```

It calculates the cross product of two pipes. The pipe on the right (`pipe2`) is tiny, and it is thus replicated to all the nodes where the `pipe1` data exist locally. A pipe with *X* number of rows crossed with a pipe with *Y* number of rows will always result in a pipe with *X* times *Y* number of rows.



<table>
<tr><th>pipe1</th><th>pipe2</th><th>pipe1.crossWithTiny(pipe2)</th></tr>
<tr>
<td><table><tr><td>'kid</td><td>'age</td></tr><tr><td>john</td><td>4</td></tr><tr><td>liza</td><td>5</td></tr></table></td>
<td><table><tr><td>'fruit</td><td>'quantity</td></tr><tr><td>orange</td><td>15</td></tr><tr><td>apple</td><td>12</td></tr></table></td>
<td><table><tr><td>'kid</td><td>'age</td><td>'fruit</td><td>'quantity</td></tr><tr><td>john</td><td>4</td><td>orange</td><td>15</td></tr><tr><td>john</td><td>4</td><td>apple</td><td>12</td></tr><tr><td>liza</td><td>5</td><td>orange</td><td>15</td></tr><tr><td>liza</td><td>5</td><td>apple</td><td>12</td></tr></table></td>
</tr>
</table>

The syntax of the `normalize` operation is as follows:

```
pipe.normalize(fields)
```

It normalizes the values of a column by dividing each value by the sum of the column.

The syntax of the `partition` operation is as follows:

```
pipe.partition(fields -> partition) { function } {groupOperations}
```

It partitions data in the pipe into several groups and applies one or more group operations to each group. The following example calculates the average weight of minors and adults:

```
Tsv("football_team.tsv", 'player,'age,'weight,).read
  .partition('age -> 'isAdult) { (:Int)> 18 }
     { group => group.average }
```

A simple example

To make clear that Scalding operations can be chained together to implement a complete pipeline look at the following example:

```
Tsv(args("input"), ('kid,'age,'fruits))
.read
.flatMap('fruits -> 'fruit) { text : String => text.split(",") }
.project('kid, 'fruit)
.write(Tsv("results.tsv"))
```

The same example can be expressed in a number of pipes, where we assemble and control how each pipe connects to another:

```
val logs = Tsv(args("logfiles"), LogsOperations.schema )
  .read
  .extractSomeUserInfo

val customers = Tsv(args("cust_log"),COperations.schema).read
  .extractSomeCustomerInfo

val joined = logs.joinWithSmaller(customers, 'user)

val result = joined.filter(
  .write(Tsv(args("output")))
```

The preceding code allows us to whiteboard our designs for processing data before implementing a data processing flow. Refer to the pipeline definition of the first chapter to see how the theory matches the implementation.

Typed API

Scalding offers another type-safe API. The type-safe API is not named. Data streams through the typed pipe as tuples, and reference to data is achieved through the notation `tuple._1` or by using the `case` classes, as shown in the following code:

```
case class Animal(name : String, kind: String)
val animals : TypedPipe[Animal] = getAnimals
val birds = animals.filter { _.kind == "bird" }
```

Operations that exist in both APIs and pipes interoperate. We can convert from `Fields` to `Typed` and the other way round. Using any one of the APIs is a matter of preference and code readability.

Summary

This chapter presented the complete Scalding Fields API. Effective usage of the preceding operations can easily solve complex data processing jobs. Ad targeting, extracting, transferring and loading, logfile analysis, bioinformatics, machine learning, predictive analytics, finance, web content mining, and other applications can be built.

In the following chapter, we will do exactly that. We will analyze and implement real use cases.

4
Intermediate Examples

This chapter goes through a real implementation in Scalding of non-trivial applications using the operations presented in the previous chapter. We will go through the data analysis, design, implementation, and optimization of data-transformation jobs for the following:

- Logfile analysis
- Ad targeting

Analyzing logfiles that have been stored for some time is a usual starting application of a new Hadoop team in an organization. The type of value to extract from the logfiles depends on the use case. As an example, we will use a case where we will need to think a lot about how to manage the data.

Another example of Ad targeting will make us look at how to structure and store the data to allow us to run daily jobs. It will involve input from data scientists and deep analysis of customer behavior to recommend personalized advertisements.

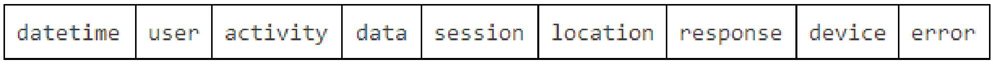
Logfile analysis

The results of this data-processing job will be displayed on a web application that presents on an interactive map, the geographic locations where users log in from. This web application will allow filtering data based on the device used.

Our job is to analyze 10 million rows of logs and generate such a report in a JSON file that can drive the web application. Because of the nature of the web application, the maximum size of that file should not exceed a few hundred kilobytes. The challenge here is how to manage data in such a way as to efficiently construct this report.

It is all about the data, and we will be using Scalding to start exploring. Around 10 million rows of data exist in tab-separated files in a Hadoop cluster in the location `hdfs:///log-files/YYYY/MM/DD`.

The TSV files contain nine columns of data. We discover that the 'activity column contains values such as login, readArticle, and streamVideo, and we are interested only in the login events. Also, if we go through the available columns of data, we will understand that we are interested in just the columns 'device and 'location.

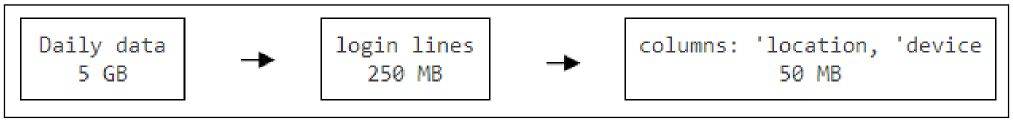


datetime	user	activity	data	session	location	response	device	error

We can implement a job in Scalding to read data, filter login lines, and project the columns with the following code:

```
import com.twitter.scalding._

class ExploreLogs (args: Args) extends Job(args) {

  val logSchema = List ('datetime, 'user, 'activity, 'data,
   'session, 'location, 'response, 'device, 'error)

  val logs = Tsv("/log-files/2014/07/01/", logSchema )
   .read
   .filter('activity) { x:String => x=="login" }
   .write(Tsv("/results/2014/07/01/log-files-just-logins/"))

  val sliced_logs = logs
   .project('location, 'device)
   .write(Tsv("/results/2014/07/01/logins-important-columns/"))
}
```

Executing this job will highlight data distribution. By filtering lines and projecting important columns, we have minimized the amount of data to be processed by two orders of magnitude compared to the original input data:



Having this insight into the data will allow us to optimize our MapReduce job. Overall, the performance of the job depends on the size of the Hadoop cluster it runs against. A decently sized cluster should only take a few seconds to complete the job and generate a number of files in the output folders.

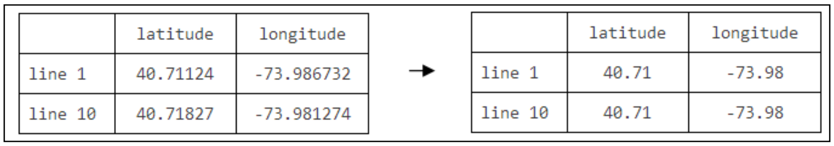
Let's take a moment to understand how Scalding code is executed as a MapReduce application and how it scales and parallelizes in the cluster. First of all, the *flow planner* knows that reading from HDFS, applying a filter, projecting columns, and writing data can be packed inside a single map phase. So the preceding diagram is a map-only job at the moment.

Scalability now depends on the size of the data. Once the job gets submitted to the Hadoop JobTracker, HDFS is interrogated about the number of files to be read and the number of HDFS blocks the input data consists of.

In this specific example, if input data of 5 GB is stored in a single file in a Hadoop cluster that uses a block size of 128 MB, then in total, 40 blocks of input data exist. For every block of data, one map task will be spawned containing our entire Scalding application logic.

So our job has two pipes, one that stores only login lines, and another that further projects some columns and stores data. For each pipe, there is a map phase that consists of 40 map tasks (to match the number of blocks). No reduce phase is required.

Now, we have to tackle the problem of reducing the data by another two magnitudes. Results reveal that latitudes and longitudes are precise, and that login events originate mostly from urban areas with a high density of population. Multiple login locations are only a few hundred meters away, and for the purpose of an interactive map, a geographic accuracy of a few miles would be sufficient. We can thus apply some compression to the data by restricting accuracy.



	latitude	longitude
line 1	40.71124	-73.986732
line 10	40.71827	-73.981274

→

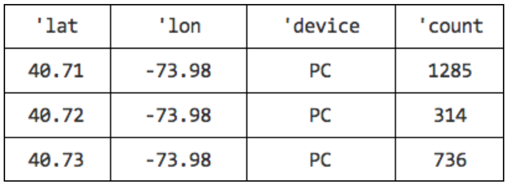
	latitude	longitude
line 1	40.71	-73.98
line 10	40.71	-73.98

This technique is known as *bucketing* and *binning*. Instead of having accurate locations (as this is not part of the specifications), we will aggregate the login events to an accuracy of two decimal points. To complete the data-processing job, we will group events by the latitude, longitude, and device type, and then count the number of login events on that particular location.

This can be achieved by introducing the following code:

```
.map('location -> 'lat, 'lon)
 { x:String => val (lat,lon) = x.split(",")
  ("%4.2f" format lat.toFloat, "%4.2f" format lon.toFloat)
 }
.groupBy('lat, 'lon, 'device)
  { group => group.size('count) }
```

In the map operation, we split the comma-separated value of `'location` into `'lat` and `'lon`, and format the location into a float with an accuracy of two decimals. We then group all the login events that occurred in that day at a specific latitude and longitude, and for the same device type, and apply an operation to count the number of elements of each group, resulting in the following:



'lat	'lon	'device	'count
40.71	-73.98	PC	1285
40.72	-73.98	PC	314
40.73	-73.98	PC	736

For the preceding specific example, thousands of log lines with locations have been compressed into just a few bytes. Executing the data-processing application to the whole data reveals that we have reduced the amount of data to more than two orders of magnitude (to less than 100 KB).

Let's take a moment to analyze how our code is executed as a MapReduce job on the cluster. The tasks of mapping `'location` into `'lat` and `'lon` and applying the accuracy restriction to the floats are packaged together and parallelized in the same 40 map tasks.

We know that after the map phase, a Reduce phase is to be executed because of the `groupBy` operation we used. We usually do not define the exact number of reduce tasks to be executed. We let the framework calculate how many reduce tasks to parallelize the task into.

In our case, we can see in the JobTracker web interface (presented in *Chapter 2*, *Get Ready for Scalding*) that the `groupBy` operation is packaged into a reduce phase that consists of 30 reduce tasks. So this is now a full MapReduce job with a map phase and a reduce phase.

The question is why do we get 30 reducers. As we said, we let the framework try to optimize the execution. Before executing the job, the flow planner knows the size of the input data (that is, 40 blocks). It knows the flow as well, which we filter and project, but it cannot infer how much of the data will be filtered out, before the execution time. Without any insight, it assigns 30 reducers to be executed for this task, as it assumes that it is possibly in the worst-case scenario—there is no data to be filtered out.

As we have already explored the data, we know that only around 50 MB are to be reduced. So three reducers should be more than enough to group that amount of data and perform the count. To improve the performance, we can optimize the execution by specifying the number of reducers, for example, as three:

```
    { group => group.size('count).reducers(3) }
```

By executing the job including the `reducers` operation, we will discover that the results are stored in three files, `part-00000`, `part-00001`, and `part-00002` (one file per reducer), as that reduce was the last phase of our job before writing the results to the file system.

Our job has not been completed before generating the single valid JSON object in a file. To achieve that, we first need to transform each line of the results into valid JSON lines with the following code:

```
  .mapTo( ('lat, 'lon, 'device, 'count) -> 'json)
    { x:(String,String,String,String) =>
         val (lat,lon,device,count) = x
         s"""{"lat":$lat,"lon":$lon,"device":"$device",count:$count}"""
  }
```

Adding the above operation to our pipeline, we now generate valid JSON lines:

```
  { "lat":40.71, "lon":-73.98, "device":"PC", count: 1285 }
```

The final step required is to aggregate all the above lines into a single valid JSON array, and this is exactly what `groupAll` achieves:

```
  .groupAll { group => group.mkString('json, ",") }
  .map('json -> 'json) { x:String => "[" + x + "]" }
```

All JSON lines are reduced in a single reducer and then the final result is encapsulated within brackets `"["` and `"]"` to construct a valid JSON array. A single file is now generated, thereby fulfilling the requirements of the project.

Completing the implementation

The final code, in a single file that contains the full data transformation flow, is as follows:

```
import com.twitter.scalding._
import cascading.pipe.Pipe

class LoginGeo (args:Args) extends Job(args) {

 val schema = List ('datetime, 'user, 'activity, 'data,
    'session, 'location, 'response, 'device, 'error)

 def extractLoginGeolocationIntoJSONArray (input:Pipe) =
  input.filter('activity) { x:String => x=="login" }
   .project('location, 'device)
   .map ('location -> ('lat, 'lon)) { x:String => {
      val Array(lat, lon) = x.split(",")
      ("%4.2f" format lat.toFloat, "%4.2f" format lon.toFloat)
     }
   }
   .groupBy('lat, 'lon, 'device)
     { group => group.size('count).reducers(3) }
   .mapTo( ('lat, 'lon, 'device, 'count) -> 'json) {
       x:(String,String,String,String) =>
       val (lat,lon,dev,count) = x
       s"""{"lat":$lat,"lon":$lon,"device":"$dev",count:$count}"""
   }
   .groupAll { group => group.mkString('json, ",") }
   .map('json -> 'json) { x:String => "[" + x + "]" }

   val input = Tsv( args("input"), schema ).read
   val result = extractLoginGeolocationIntoJSONArray(input)
    .write(Tsv( args("output") ))
}
```

To analyze the finalized job scalability, there is a map phase that reads and filters the input in 40 map tasks. This is followed by a reduce phase of three reduce tasks, then another map phase of three map tasks that generate the JSON lines, followed by a reduce phase with a single reducer where we insert a comma between the lines, and finally, the last map phase that consists of one map task that adds the brackets to the string and stores to a single file.

So in effect, the application is executed as:

```
Map phase | Reduce phase | Map phase | Reduce phase | Map phase
 40 tasks |    3 tasks   |  3 tasks  |    1 task    |  1 task
```

That's it! With Scalding, we expressed in just a few lines of code a complex algorithm with multiple map and reduce phases. The same functionality would require hundreds of lines of code in Java MapReduce.

Testing such Scalding jobs will be covered thoroughly in *Chapter 6, Testing and TDD*. A simple example of a test that uses some mock data as the input and asserts that the expected output is the same as the mock output is as follows:

```
import com.twitter.scalding._
import org.scalatest._

class LoginGeoTest extends WordSpec with Matchers {
  import Dsl._
  val schema = List ('datetime, 'user, 'activity, 'data,
    'session, 'location, 'response, 'device, 'error)
  val testData = List(
    ("2014/07/01","-","login","-","-","40.001,30.001","-","PC","-"),
    ("2014/07/01","-","login","-","-","40.002,30.002","-","PC","-"))

  "The LoginGeo job" should {
   JobTest("LoginGeo")
   .arg("input", "inputFile")
   .arg("output", "outputFile")
   .source(Tsv("inputFile", schema), testData )
   .sink[(String)](Tsv("outputFile")) {
    outputBuffer => val result = outputBuffer.mkString
    "identify and bucket nearby login events" in { res shouldEqual
       s"""[{"lat":40.00,"lon":30.00,"device":"PC",count:2}]"""
    }
  }.run
   .finish
}
```

Exploring ad targeting

As part of our second example, we will explore the same logfiles with another job for the purpose of generating personalized ads. Let's assume that the company we are working for provides news articles with associated videos to users. For the purpose of the example, we will assume that four categories of news are presented: sports, technology, culture, and travel.

Category	**Subcategories**				
Sports	Football	Rugby	Tennis	F1	Cycling
Tech	Games	Mobile	Gadget	Apps	Internet
Culture	Books	Film	Music	Art	Theatre
Travel	Hotels	Skiing	Family	Budget	Breaks

Analyzing and understanding the data deeply, requires lots of exploration. Fortunately, a Data Scientist validates and calculates some assumptions that result in the following conclusions:

- Our users spend time reading articles and spend more than 20 seconds if they are slightly interested and more than 60 seconds if they are really interested.
- Users who also view the video accompanying each article are considered as engaged users.
- Occasionally, users get interested in a category they are normally not interested in. The recent behavior has more relevance than past behavior.

> Recent interest in Travel-Skiing is a high indication for us to the recommended relevant travel ads.

Quantifying the preceding observations, and for the sake of simplicity, we will assume that the recommendation system will be based on an algorithm that assigns to each user points on each category and subcategory. So, the type of ads to associate with that user depends on the category and subcategory the user is most interested in.

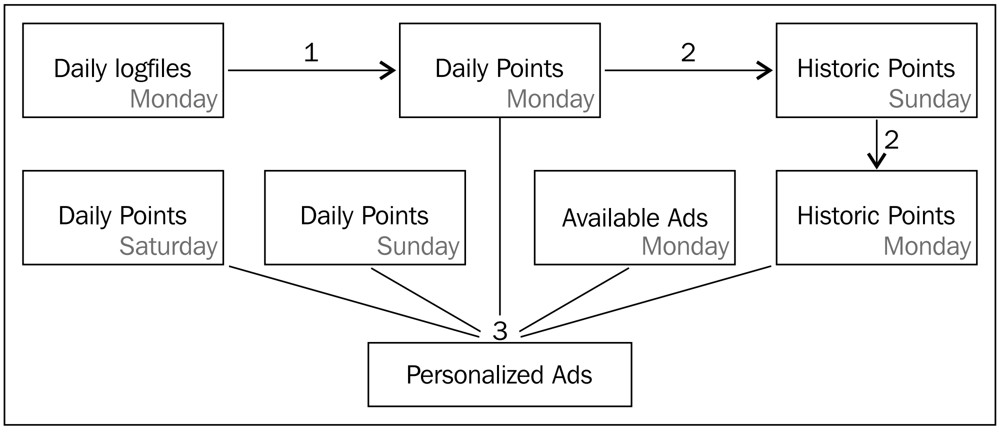
- One point for each read event that lasts more than 20 seconds
- Three points for each read event that lasts more than 60 seconds
- Three points per video view

- The resulting ranking calculation for each user is as follows:

```
User points = 40% * points yesterday + 20% * points 2 days ago +
  10% * points 3 days ago + 30% of the average historic points
```

To implement the algorithm, we can conceptualize three tasks:

1. Initially process daily logfiles and calculate the user points for that day. Store the results in a structure `/YYYY/MM/DD/` so that the data is nicely partitioned across multiple executions.
2. In a similar way, calculate the historic points in another pipeline.
3. Once all the data is there, read the daily points of the last three days and the historic points, and join the results with the available advertisements to generate the personalized ads:



The important aspect of the preceding diagram is how we manage and transform data in the filesystem. For a particular day, for example, Monday, we calculate the daily points and store the results in an HDFS folder `/dailypoints/YYYY/MM/DD/`.

Now, we can generate the historic points by joining the daily points generated today, (Monday as shown in the previous diagram), with the historic points calculated yesterday (Sunday). We perform the same partitioning structure to the historic points, that is, `/historicpoints/YYYY/MM/DD/`.

Storing the resulting data that makes sense in an organized way is a good practice if you want to reuse that data at a later date to extract different types of values out of it.

We will proceed with the implementation of the three pipelines.

Calculating daily points

Our first task is to calculate the daily points. We always explore data before even thinking about the implementation. To put the data into perspective, a small job will group data together by the user and sort it by time:

```
import com.twitter.scalding._

class CalculateDailyAdPoints (args: Args) extends Job(args) {

  val logSchema = List ('datetime, 'user, 'activity, 'data,
   'session, 'location, 'response, 'device, 'error)

  val logs = Tsv("/log-files/2014/07/01", logSchema )
   .read
   .project('user,'datetime,'activity,'data)
   .groupBy('user) { group => group.sortBy('datetime) }
   .write(Tsv("/analysis/log-files-2014-07-01"))
}
```

Remember that this is the exact same data we used in the previous example, but now, we are not interested in login events, or latitude and longitude locations. Now, we are interested in the `readArticle` and `streamVideo` activities. The following is the data a particular user generated yesterday:

```
user1   2014-07-01 09:00:00   login
user1   2014-07-01 09:00:05   readArticle   sports/rugby/12
user1   2014-07-01 09:00:20   readArticle   sports/rugby/7
user1   2014-07-01 09:01:00   readArticle   sports/football/4
user1   2014-07-01 09:02:30   readArticle   sports/football/11
user1   2014-07-01 09:03:50   streamVideo   sports/football/11
user1   2014-07-01 09:05:00   readArticle   sports/football/2
user1   2014-07-01 11:05:00   readArticle   sports/football/3
```

Looking at the data, we clearly see that we should focus on how to calculate the duration in seconds; a user is reading a specific article like `sports/football/4`. We can achieve this using a buffered operation such as `scanLeft`, which scans through the buffer and has access to the event time of the previous line and the event time of the current line. Before thinking more about it, let's continue observing the data.

With a more careful look, we can observe that there is a huge two-hour gap between 09:05:00 and 11:05:00. The user did not generate any log lines during this period, and the user was of course not spending two hours reading the article. He was somehow disengaged, that is, he was having his breakfast or chatting on the phone.

Also, we cannot calculate the duration of the very last event. For all we know, the user might have switched off their laptop after that event.

```
user1   2014-07-01 11:05:00   readArticle   sports/football/3
```

When we have such lines where we do not have a full picture of what happened in reality, and when the duration is more than 20 minutes, the requirements mention that we should treat them as a partial read and associate one point.

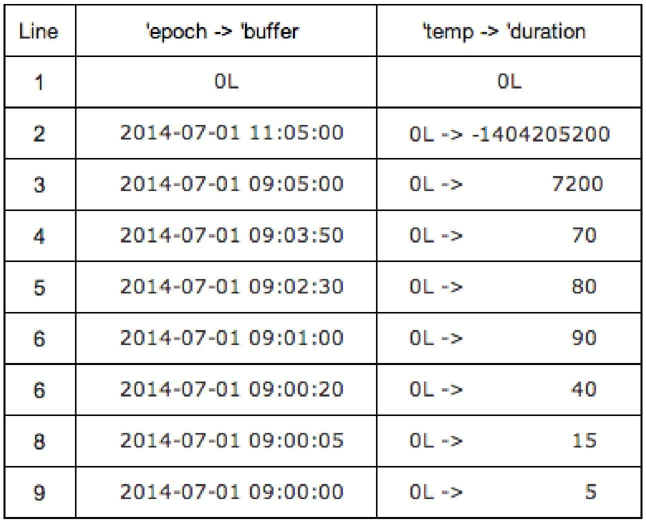
A naïve implementation of the duration calculation algorithm would be to group by user, sort by datetime, and then apply a toList operation in order to iterate over that list. In that iteration, we can calculate the duration as nextTime - previousTime and then flatten the results. Remember that toList is one of the operations that put everything in memory. This could even result in out-of-heap space errors in our job execution and is not the most optimized way.

For efficient windowed calculations, Scalding provides the group operation scanLeft, which utilizes a tiny buffer to achieve the same result. So for the event that is happening at 09:00:05, we can calculate the duration as 09:00:20 - 09:00:05 = 15 seconds. While performing this calculation, we store the current event time in the buffer for the following line to use in its calculations.

For this calculation, we will be emitting a tuple of two elements: duration and previous epoch. As we are emitting a tuple of size two, the input to the scanLeft operation should also be two. For that, we will use as input the current epoch and a helper field called temp.

```
import com.twitter.scalding._

class CalculateDailyAdPoints (args: Args) extends Job(args) {

  val logSchema = List ('datetime, 'user, 'activity, 'data,
   'session, 'location, 'response, 'device, 'error)

  val logs = Tsv("/log-files/2014/07/01", logSchema )
   .read
   .project('user,'datetime,'activity,'data)
val logsWithDurations = logs
  .map('datetime -> 'epoch) { x: String => toEpoch(x) }
  .insert('temp, 0L) // helper field for scanLeft
  .groupBy('user) { group =>
    group.sortBy('epoch)
    .reverse
    .scanLeft(('epoch,'temp)->('buffer,'duration))((0L,0L)) {
      (buffer: (Long, Long), current: (Long, Long)) =>
        val bufferedEpoch = buffer._1
        val epoch = current._1
        val duration = bufferedEpoch - epoch
        (epoch, duration)
    }
   }
   .filter('duration) { x: Long => x = 0 }
   .discard('bufferedEpoch, 'epoch, 'temp)
   .write(Tsv("/log-files-with-duration/2014/07/01"))
}
```

During the left scan, we read the value from the `epoch` symbol and store it in the `buffer` variable so that the next scan can access the current date time. We also read `temp` but do not use it. Instead, we calculate the duration as the difference between the value in the buffer and the current epoch. Running the `scanLeft` on the data generates the event duration in seconds:



Line	'epoch -> 'buffer	'temp -> 'duration
1	0L	0L
2	2014-07-01 11:05:00	0L -> -1404205200
3	2014-07-01 09:05:00	0L -> 7200
4	2014-07-01 09:03:50	0L -> 70
5	2014-07-01 09:02:30	0L -> 80
6	2014-07-01 09:01:00	0L -> 90
6	2014-07-01 09:00:20	0L -> 40
8	2014-07-01 09:00:05	0L -> 15
9	2014-07-01 09:00:00	0L -> 5

The first two lines look bizarre, and we get nine lines as the output using eight lines of input. The first line is the side effect of initializing `scanLeft` with the default values `(0L, 0L)`. The second line is the result of the calculation of the duration as zero minus the current date time. This happens only in line `11:05:00`. Of course, this is the last event line in our logs for that user. Remember that for the last event, it is impossible to calculate the duration as the user might have just switched off his laptop.

The specifications mention that for such occasions where we do not have the full picture, we should treat them as a partial read and associate one point. Also, if the duration is more than 20 minutes, we have to treat it as a partial read. We can solve both issues with a map that uses the duration to fix to a partial read.

```
.map('duration->'duration)
  { x:Long => if ((x<0) | (x>1200)) 20 else x }
```

We also clean up one extra line that is generated by `scanLeft` using the following code:

```
   .filter('duration) { x: Long => x != 0 }
```

The most complex part of the algorithm is now complete. We have correctly calculated the duration of events. Generating points is just another `map` operation:

```
.map(('activity , 'duration) -> 'points) { x:(String,Int) =>
  val (action, duration) = x
  action match {
    case "streamVideo" => 3
    case "readArticle" =>
     if (duration>=1200) 1 else if (duration>=60) 3 else
     if (duration>=20) 1 else 0
    case _ => 0
  }
}
```

The process requires us to filter out lines that do not contribute any points in this calculation, and extract the category and subcategory from `'data`:

```
.filter('points) { x: Int => x > 0 }
.map('data -> ('category, 'subcategory)) { x: String =>
   val categories = x.split("/")
   (categories(0), categories(1))
}
```

Then, group by the user, category, and subcategory, and aggregate the daily points:

```
  .groupBy('user,'category,'subcategory)
    { group => group.sum[Int]('points) }
```

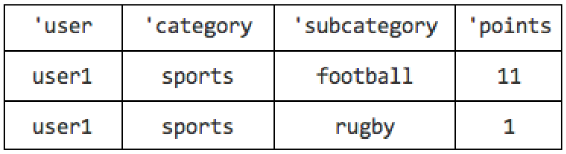
The resulting implementation of the pipeline that calculates points is as follows:

```
val logs = Csv(args("input"), ",", logSchema ).read
  .project('user,'datetime,'activity,'data)
  .map('datetime -> 'epoch) { x: String => toEpoch(x) }
  .insert('temp, 0L) // helper field for scanLeft
  .groupBy('user) { group =>
   group.sortBy('epoch)
    .reverse
    .scanLeft(('epoch, 'temp)->('buffer,'duration))((0L, 0L)) {
     (buffer: (Long, Long), current: (Long, Long)) =>
       val bufferedEpoch = buffer._1
       val epoch = current._1
       val duration = bufferedEpoch - epoch
       (epoch, duration)
    }
  }
  .map('duration->'duration)
    { x:Long => if ((x<0) || (x>1200)) 20 else x }
  .filter('duration) { x: Long => x != 0 }
  .map(('activity , 'duration) -> 'points) { x:(String,Int) =>
    val (action, duration) = x
    action match {
      case "streamVideo" => 3
      case "readArticle" =>
        if (duration>=60) 3 else if (duration>=20) 1 else 0
      case _ => 0
    }
  }
  .filter('points) { x: Int => x > 0 }
  .map('data -> ('category, 'subcategory)) { x: String =>
    val categories = x.split("/")
    (categories(0), categories(1))
  }
  .groupBy('user,'category,'subcategory)
    { group => group.sum[Int]('points) }
```

This Scalding code is executed in the cluster as follows:

Map phase | Reduce phase | Map phase | Reduce phase

It generates the expected results:



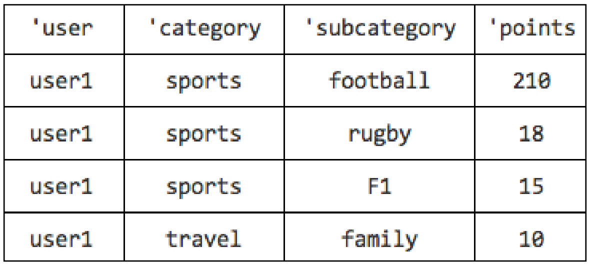
'user	'category	'subcategory	'points
user1	sports	football	11
user1	sports	rugby	1

Calculating historic points

The implementation of the historic point calculation requires a separate data pipeline to be implemented in Scalding. It is a simple one, and we read the existing historic points (the ones generated yesterday) and add the just calculated new points.

```
val historyPipe = Tsv(args("input_history"),schema).read

val updatedHistoric =
  (dailyPipe ++ historyPipe)
  .groupBy('user,'category,'subcategory)
    { group => group.sum[Int]('points) }
  .write(Tsv("/historic-points/2014/07/01"))
```

Thus, if the historic points of a user are as follows:



'user	'category	'subcategory	'points
user1	sports	football	210
user1	sports	rugby	18
user1	sports	F1	15
user1	travel	family	10

Generating targeted ads

The final task is to implement the ranking algorithm:

```
user points = 40% * points yesterday + 20% * points 2 days ago +
     10% * points 3 days ago + 30% of the average historic points
```

We can achieve this using `map`, and we can also calculate the average of the historic points over the number of days for which the analysis has been running. The ranking algorithm is as follows:

```
val pipe1 = yesterdayPipe.map('points -> 'points)
 { x:Long => x*0.4 }
val pipe2 = twoDaysAgoPipe.map('points -> 'points)
 { x:Long => x*0.2 }
val pipe3 = threeDaysAgoPipe.map('points -> 'points)
 { x:Long => x*0.1 }
val normalize = 40 // Days we calculate historic points
val pipe4 = historyPipe.map('points -> 'points)
 { x:Long => (x /40) * 0.3 }

val user_category_point = (pipe1 ++ pipe2 ++ pipe3 ++ pipe4)
  .groupBy('user, 'category, 'subcategory)
    { group => group.sum[Long]('points) }
```

We read all input from the respective folders and apply the ranking algorithm. The important bit is that we use the `++` operator to add together the four input pipes and aggregate the total points of each user in the `.sum` operation.

Nothing is left except for getting the recommendations. To find the best ad for each user, we group by `user`, sort by `points`, and take the first element of each group. So we are keeping the top category-subcategory for every user based on the ranking algorithm.

```
.groupBy('user)
  { group => group.sortedReverseTake('points, 1) }
```

Doing this, we are keeping the top category-subcategory for every user based on the ranking algorithm. The final step is to join that information to the available ads for tomorrow using the category-subcategory as a join key.

```
user_category_point.joinWithSmaller( ('category,'subcategory)->
  ('category, 'subcategory), adsPipe )
```

That's it. We just implemented a recommendation system for targeted ads in less than two pages.

Summary

In this chapter, we used the same dataset to present two completely different use cases. For each use case, we explored the data and then designed and implemented data-processing applications in Scalding. We also looked at how an abstract pipelining language (Scalding) is translated in MapReduce phases.

We also introduced techniques such as bucketing and windowed calculations through a solution to a problem. The expressiveness of the language allows us to implement even complex use cases with ease.

In the following chapter, we will present some design patterns that will enable us to develop more modular and testable code.

5
Scalding Design Patterns

MapReduce applications, like all software artifacts, need to be written to be reusable, modular, and testable. They must have specific requirements related to the fact that they run in a Hadoop environment. The goal of this chapter is to present some design patterns to be applied in the implementation of Scalding applications. While the principles they follow are common to software development, we will present how to implement them in the specific domain.

The principles they follow are simplicity, single responsibility, and dependency inversion. In the context of Scalding, we will call them:

- The external operations pattern
- The dependency injection pattern
- The late bound dependency pattern

The external operations pattern

To achieve modularity and fulfill the single responsibility principle, we can structure our data processing job in an organized way. An object, a trait, and a job can share parts of the responsibilities, explained as follows:

- In `package object`, we can store information about the *schema* of the data
- In `trait`, we can store all the *external operations*
- In a Scalding job, we can manage arguments, define taps, and use external operations to construct data processing pipelines

A particular dataset will usually be processed by multiple jobs to extract different value from the data. Thus, we can create an object called LogsSchemas to store input and output schemas, and also to document the locations in HDFS, where the data resides. This object can act as a registry of all the variations of datasets, and we can reuse it in any of our Scalding jobs, as shown in the following code:

```
package object LogsSchemas {
  // that is, hdfs:///logs/raw/YYYY/MM/DD/
  val LOG_SCHEMA = List('datetime, 'user, 'url)

  // that is, hdfs:///logs/daily-visits/YYYY/MM/DD/
  val LOGS_DAILY_VISITS = ('day, 'user, 'visits)
}
```

Our *operations* trait can contain all the pipe transformations. We will normally package a decent amount of complexity in an external operation. In the following example, we define two simple external operations: logsAddDayColumn and logsCountVisits, as shown in the following code:

```
import cascading.pipe.Pipe
import com.twitter.scalding._

trait LogsOperations extends FieldConversions {

  def self: RichPipe

  val fmt = org.joda.time.format.DateTimeFormat.
    forPattern("dd/MM/yyyy HH:mm:ss")

  def logsAddDayColumn : Pipe = self
    .map('datetime -> 'day) {
      date: String => fmt.parseDateTime(date).toString("yyyyMMdd")
    }

  def logsCountVisits : Pipe = self
    .groupBy(('day, 'user)) { _.size('visits) }

}
```

The trait containing all the transformation operations is external to the job and structured in a way that is easy to be tested. The self is an abstract member that refers to the pipe the operations will be applied in.

The next step is to implement a Scalding job that uses the preceding modular code using the following code:

```
import com.twitter.scalding._

class SimpleJob(args: Args) extends Job(args) {
  import LogsSchemas._
  import LogsWrapper._

  Tsv(args("input"), LOG_SCHEMA).read
    .logsAddDayColumn
    .logsCountVisits
    .write(Tsv(args("output"), LOGS_DAILY_VISITS))
}
```

The responsibilities of the Scalding job are now restricted to the following:

- Validating and using input arguments and parameters
- Creating input and output taps
- Composing the data processing pipelines using external operations

The glue that connects the schemas and external operations to a job are the `import` lines at the beginning of the job, as shown in the following code:

```
  import LogsSchemas._
  import LogsWrapper._
```

The wrapper is an object which provides a constructor that enables the external operations on `Pipe` and offers an implicit conversion of `RichPipe` to `Pipe`.

```
import com.twitter.scalding.RichPipe

object LogsWrapper {
  implicit def wrapPipe(self: cascading.pipe.Pipe): LogsWrapper =
     new LogsWrapper(new RichPipe(self))
  implicit class LogsWrapper(val self: RichPipe) extends
     LogsOperations with Serializable
}
```

This wrapper enables our external operations to be executed in a Scalding pipe.

> This technique is known as "Extension Methods" in Scala. For further details on how this works, please read `http://docs.scala-lang.org/overviews/core/value-classes.html#extension_methods`.

Now, we can easily imagine the example in the previous chapter to be structured in a Scalding job, as shown:

```
class AdsJob(args: Args) extends Job(args) {
  ...

  val dailyPoints = Tsv(args("logs"), LOG_SCHEMA).read
    .calculateDailyAdPoints
    .write(Tsv(args("daily-points")))

  val histPoints = Tsv(args("historic-points"), HPOINTS).read

  (dailyPoints ++ historicPoints)
    .calculateHistoricAdPoints
    .write(Tsv(args("historic-points"))

  val points2daysAgo = Tsv(args("points-2d"), POINTS_SCHEMA).read
  val points3daysAgo = Tsv(args("points-3d"), POINTS_SCHEMA).read

  (dailyPoints ++ points2daysAgo ++ points3daysAgo + histPoints)
    .calculateTopCategorySubcategory
    .joinWithAds(Tsv(args("daily-ads"), ADS_SCHEMA).read)
    .write(Tsv(args("suggested-ads")))
}
```

Extracting the operations into a trait allows us to reuse them in a different project by simply extending the trait. If this is not needed, it is possible to simplify the pattern implementation by simply implementing the methods inside the wrapper classes.

The external operations pattern is the basis of the dependency injection that is presented next, and also of the test driven development approach presented in the following chapter.

We can execute the preceding code with the following:

```
$ java -cp target/chapter5-0-jar-with-dependencies.jar
  com.twitter.scalding.Tool externaloperations.ExampleJob --local
  --input src/main/resources/logs.tsv --users
  src/main/resources/users.tsv --output data
```

The dependency injection pattern

The dependency injection design pattern allows us to remove hard-coded dependencies to make it possible to change them without recompile or at runtime. It enables us to effectively unit test the code, increase its reuse and flexibility, and support application configuration.

The only cost introduced by the pattern is a slightly more complex structure, since we have to expose the dependency and provide it to our underlying code. This is in general completely justified, whenever we can (or we have to) extract part of our logic into independent components.

Dependency injection is based on the external operations pattern. Again, we have a package object as shown:

```
package object ExampleSchema {
  val LOG_SCHEMA = List('datetime, 'user, 'url)
  val OUTPUT_SCHEMA = List('datetime,'user,'url, 'email, 'address)
}
```

This time though we will join the users using an external REST API to fetch the email and address information. The interface and a mock implementation can be provided as the following:

```
trait ExternalService {
  def getUserInfo(userId: String): (String,String)
}
class ExternalServiceImpl extends ExternalService {
  def getUserInfo(userId: String) = ("email", "address")
}
```

Then, in our external *operations* trait, we can use the external service to query data as shown:

```
trait ExampleOperations extends FieldConversions {
  import Dsl._
  def self: Pipe
  def externalService: ExternalService
  def addUserInfo: Pipe = self.map('user ->('email, 'address))
    { userId: String => externalService.getUserInfo(userId)}
}
```

The binding to the external service is done in the definition of the wrapper object. This is where we will actually inject the dependency:

```
object ExternalServiceWrapper {

    implicit class ExampleServiceWrapper(val self: Pipe)(implicit val
externalService : ExternalService) extends ExampleOperations with
Serializable
    implicit def fromRichPipe(rp: RichPipe)(implicit externalService :
ExternalService) = new ExampleServiceWrapper(rp.pipe)

}
```

Our Scalding job preserves the same single responsibilities and injects an implementation defined as the `externalService`.

```
import com.twitter.scalding._

class ExampleJob(args: Args) extends Job(args) {

  import ExampleSchema._
  import ExternalServiceWrapper._

  implicit val externalService = new ExternalServiceImpl()

  Tsv(args("input"), LOG_SCHEMA).read
    .addUserInfo
    .write(Tsv(args("output"), OUTPUT_SCHEMA))
}
```

Rather than joining with an existing file source, we will now join data over an external service where we inject the dependency. We can execute the preceding code with the following:

```
$ java -cp target/chapter5-0-jar-with-dependencies.jar
 com.twitter.scalding.Tool dependencyinjection.ExampleJob --local
 --input src/main/resources/logs.tsv --output data/output.tsv
```

Find complete project files in the code accompanying this book at `http://github.com/scalding-io/ProgrammingWithScalding`.

The late bound dependency pattern

Serialization of objects is a problem in all distributed systems since we have to send objects between different machines. Sometimes, a class does not extend `Serializable`, and we cannot simply extend `java.io.Serializable` to solve the problem (that is, because of a dependency to a library like the `HttpClient`). An example of such a case could be the following code:

```
case class UserInfo(email: String, address: String)
// Note this is a non- serializable class
class ExternalServiceImpl extends ExternalService { ... }
```

In such cases, we need to postpone the object instantiation. We want this object to be created in every node of the Hadoop cluster, instead of being instantiated and then transferred among cluster nodes.

To achieve this, we can define all the *non-serialiazable* objects as abstract members in the operations trait. The binding can then be done by either using a `lazy val` member or by using a constructor function.

In Scala, defining a variable as `lazy` will cause the class to be instantiated just at the very first usage, thus avoiding the need to transfer it through the cluster. For this, we need to take care and not call the variable beforehand.

Working on the same example used to present dependency injection, we can add `lazy` in the wrapper, as shown in the following code:

```
object LateBoundWrapper {
  implicit class LateBoundWrapper(val self: Pipe) extends
    ExampleOperations with Serializable {
      lazy val externalService = new ExternalServiceImpl
    }
  implicit def fromRichPipe(richPipe: RichPipe) =
    new LateBoundWrapper(richPipe.pipe)
}
```

The `Job` class will be similar to the one used in the dependency injection example as shown:

```
class ExampleJob(args: Args) extends Job(args) {
  import ExampleSchema._
  import LateBoundWrapper._

  implicit val externalServiceFactory = new ExternalServiceImpl()
```

```
     Tsv(args("input"), LOG_SCHEMA).read
       .addUserInfo
       .write(Tsv(args("output"), OUTPUT_SCHEMA))
   }
```

This pattern is simple and can be used independently. Also, it doesn't have overhead or other cons, thanks to the simplicity of declaring `lazy val` members in Scala.

We can execute the above code with the following:

```
$ java -cp target/chapter5-0-jar-with-dependencies.jar
 com.twitter.scalding.Tool latebound.ExampleJob --local
 --input src/main/resources/logs.tsv --output data/latebound.tsv
```

Summary

In this chapter, we presented some design patterns for solving common problems. Initially, we presented the idea that a job should not contain complex logic. The job has defined responsibilities and delegates all the complexity of data transformations to external operations implemented in a trait. By keeping all schemas in an object and using a wrapper, we can structure our code in a modular and reusable way.

The dependency injection pattern presented how to inject dependencies at compile time, and the late bound pattern displayed how to overcome situations where an object cannot be serialized over the network.

In the next chapter, we will present the various testing strategies around our Scalding data-processing applications.

6
Testing and TDD

Testing has always been a critical aspect of application development, and in recent years, its importance has been rising steadily. New design and development techniques such as **Test-Driven Design** (**TDD**) and **Domain-Driven Design** (**DDD**) put testing at the center of the development process.

MapReduce applications are not exempt from testing requirements. Given their specificities, they require even more exhaustive testing than other applications. In this chapter, we will discuss testing and aspects related to the development of MapReduce applications that affect the way applications can be tested.

The goal of this chapter is as follows:

- Introducing testing in the context of MapReduce and its challenges
- Presenting unit and functional testing
- Presenting the test-driven development methodology

Introduction to testing

The definition of testing in software development is:

> "*Software testing can be stated as the process of validating and verifying that a computer program/application/product meets the requirements that guided its design and development, works as expected, can be implemented with the same characteristics, and satisfies the needs of stakeholders*" *(*`http://en.wikipedia.org/wiki/Software_testing`*).*

Properly testing an application means that we will have to focus on every layer of the application being developed:

- **Unit/component testing**: This verifies that a specific part or component of the application works as designed.
- **Integration testing**: This verifies that the different components interact as expected. Also, this validates that interfaces between components are consistent to the expectations of other components.
- **Acceptance testing**: This validates the final application behavior against stakeholder requirements.
- **System testing**: This validates the application completely in integration and production environments. It makes sure that the application is production ready and stable, and it respects the performance requirements in terms of execution speed and resources.

Testing the application at all the mentioned levels is important in several ways. It helps us developers maintain confidence while code evolves and have a proper understanding of the responsibilities of the different elements. Another aspect is the capability to understand where a problem is by simply following the track of the failing tests in different layers.

An important aspect about tests is that they represent a live and executable documentation of the written code. For this reason, writing concise and expressive tests is of paramount importance. Fortunately, there are frameworks and techniques to help us unclutter test code and express the definition and validation of the business logic clearly.

MapReduce testing challenges

MapReduce applications process large amounts of data in order to infer and extract information. This causes the following:

- A long feedback cycle from the execution to the validation of results
- Difficulty in finding what data to use as mock data to validate results

Another set of problems is related to the logical complexity of operations. When used for business intelligence, for example, a MapReduce application is responsible for applying a possibly complex mathematical model to vast amounts of data.

Often, the computation complexity lies in the logical and mathematical concepts behind every step, similar to what happens in the development of cryptographic applications. Thus, we need to approach design and testing at a higher level.

Due to the complexity, it is difficult to specify the expected outcome of the computation in a test. This is why we need to focus on the testability of the components participating in the computation.

Development lifecycle with testing strategy

The testing strategy described here is deeply intertwined with the software development lifecycle we follow. For data processing applications, everything starts with a data science phase, where we perform two tasks:

- **Data exploration**: Analysis of the format, frequency of arrival, and contents of the data
- **Whiteboard design**: Definition of the processing algorithm and the mathematical models to be used to generate features

These tasks are followed by two development tasks, which are:

- **TDD implementation**: Conversion of the algorithm into a scalable MapReduce application using Scalding
- **Production deployment and monitoring**: Execution, performance enhancement, and monitoring of the MapReduce job

TDD for Scalding developers

This section describes an approach to deal with the development process with a testing context. We will work through code examples in Scalding, but the majority of the concepts are also valid in a broader context, in other Scala-based MapReduce frameworks such as *Scoobi* and *Scrunch* in particular. We will first describe the testing strategy, and then implement a framework to support this strategy.

Implementing the TDD methodology

The rest of the chapter presents a step-by-step implementation of the TDD methodology. We will use the example presented in the previous chapter, which requires the following actions:

- Read a logfile and count the number of visits per user per day, where the logfiles contain user URL requests
- Enrich user information with the e-mail and address retrieved from an additional input file containing user info, which is indexed via the user ID

Once the exploration of the data and algorithm design is complete, the test-driven methodology requires the following steps:

1. Decompose the algorithm in smaller testable steps.
2. Define a set of acceptance tests to validate the consistency of the implementation with the algorithm as designed.
3. Define a set of integration tests to validate that the different steps are correctly chaining together.
4. Define a set of unit tests for each step and automate execution routinely. For simpler jobs, the acceptance and integration tests usually coincide. When our logic spans several jobs, the integration and acceptance tests will operate at different levels of granularity.
5. Implement the MapReduce logic. Steps 4 and 5 are usually interleaved. Every time a test is written and executed with failure, the developer will implement the part of logic that takes care of making the tests succeed and so on.
6. Define and perform a set of system tests to verify correctness and scalability using real-size or near to real-size amounts of data.

Decomposing the algorithm

To decompose the algorithm in smaller and testable steps, we will use the external operations pattern presented in the previous chapter, which results in the following code for the job:

```
class ExampleJob(args: Args) extends Job(args) {
  val visitsPerDay = Tsv(args("input"), LOG_SCHEMA).read
    .logsAddDayColumn
    .logsCountVisits
    .logsJoinWithUsers(Tsv(args("users"), USER_SCHEMA).read)
    .write(Tsv(args("output"),LOG_DAILY_WITH_ADDRESS))
}
```

Defining acceptance tests

The acceptance tests are defined in terms of mock data. Input mock data is provided, and the expected output is also mocked. The correctness of the application will be verified when it is executed against the mock input data and produces the expected resulting data.

The following is the input event logs:

```
"01/07/2014 10:22:11"  1000002L    "http://youtube.com"
"01/07/2014 10:22:11"  1000003L    "http://twitter.com"
"01/07/2014 10:22:11"  1000002L    "http://google.com"
"01/07/2014 10:22:11"  1000002L    "http://facebook.com"
```

The following is the input user information:

```
1000002  "stefano@email.com"     "10 Downing St. London "
1000003  "antonios@email.com"    "1 Kingdom St. London"
```

The execution of the job that counts the number of visits per user and then joins with the addresses should result in the following output:

```
"2014/07/01",1000002L,3L,"stefano@email.com","10 Downing St. London"
"2014/07/01",1000003L,1L,"antonios@email.com","1 Kingdom St. London"
```

Implementing integration tests

Once we have the acceptance testing mock data, we can implement an integration test using the `JobTest` class. This class replaces the input and output taps with their in-memory versions, and it allows us to validate that the algorithm works as expected using our mock data.

`ExampleJobTest` is implementation agnostic, which means that the following integration test can validate the full job execution, whether the external operations design pattern is used or not:

```
class ExampleJobTest extends FlatSpec with Matchers with
  FieldConversions with TupleConversions {
  import ExampleSchema._

  "A sample job" should "do the full transformation" in {

    val logs = List(
      ("01/07/2014 10:20:10", 1000002L, "http://youtube.com"),
      ("01/07/2014 10:20:35", 1000003L, "http://twitter.com"),
      ("01/07/2014 10:21:38", 1000002L, "http://google.com"),
      ("01/07/2014 10:21:55", 1000002L, "http://facebook.com")
    )
    val users = List(
      (1000002L, "stefano@email.com", "10 Downing St. London"),
      (1000003L, "antonios@email.com", "1 Kingdom St. London")
    )
    val expectedOutput = List(
      ("2014/07/01", 1000002L, 3L, "stefano@email.com","10 Downing St. 
London"),
      ("2014/07/01", 1000003L, 1L, "antonios@email.com", "1 Kingdom 
St. London")
)

    JobTest(classOf[ExampleJob].getName)
      .arg("input", "input-logs")
      .arg("users", "users-logs")
      .arg("output", "output-data")
      .source(Tsv("input-logs", LOG_SCHEMA), logs)
      .source(Tsv("users-logs", USER_SCHEMA), users)
      .sink(Tsv("output-data", LOG_DAILY_WITH_ADDRESS)) {
         buffer: mutable.Buffer[(String, Long, Long, String, 
          String)] =>
         buffer should equal(expectedOutput)
      }
      .run
  }
}
```

The preceding code serves great as a documentation of the algorithm and validates the complete implementation of the job, including the following:

- The definition of the input path and format
- The end-to-end data transformation
- The output locations and format

Internally, the `JobTest` class performs the execution of the Scalding job in a controlled environment, in which tests replace the input taps with their in-memory versions and feed in the job, mock test data. The output taps are also replaced with in-memory sinks in which it is possible to execute assertions on the generated data. As a result, integration testing forces the tester to deal with the whole job.

Implementing unit tests

To perform proper unit testing, we will use the external operations design pattern. Following the test-driven approach, we will not provide the full implementation in our operations trait, as shown:

```
trait ExampleOperations {
  import Dsl._
  def self: Pipe
  def addDayColumn : Pipe = pipe
  def countUserEventsPerDay : Pipe = pipe
  def addUserInfo(userData: Pipe) : Pipe = pipe
}
```

Next, we will start using the **ScaldingUnit** framework that has been accepted in Scalding since Version 0.9.1 as `com.twitter.scalding.bdd`.

> ScaldingUnit has been developed and committed by the authors of this chapter, and should be used in Version 0.8.11 of Scalding, or earlier. Project files are provided at `https://github.com/scalding-io/ScaldingUnit`.

Having extracted the operations into an external class not extending `Job`, we are able to test every single step independently, ignoring all the logic related to the definition of the source, sink types, and paths. We can precede implementing one unit test for each external operation, starting with `logsAddDayColumn`, as shown:

```
class ExampleOperationsUnitTests extends FlatSpec with Matchers with
BddDsl {
  import ExampleSchema._
  import ExampleWrapper._

  "Unit-Test: The example Job" should "add column with day" in {
    Given {
      List(("12/07/2014 10:22:11", 1000002L,
        "http://www.youtube.com")) withSchema LOG_SCHEMA
    } When {
      pipe:Pipe => pipe.logsAddDayColumn
    } Then {
      buffer: mutable.Buffer[(String, Long, String, String)] =>
        buffer.toList(0) should equal (("12/07/2014 10:22:11",
1000002L, "http://www.youtube.com", "2014/07/12"))
    }
  }
```

The preceding unit test defines the expected behavior of the operation and asserts the correctness of the output. We can continue implementing a unit test for the external operation `logsJoinWithUsers`, as shown in the following code:

```
it should "add user info" in {
  Given {
    List(("2014/07/01", 1000002L, 1L)) withSchema
      LOGS_DAILY_VISITS
  } And {
    List((1000002L, "stefano@email.com", "10 Downing St.
      London")) withSchema USER_SCHEMA
  } When {
    (logs: Pipe, users: Pipe) => logs.logsJoinWithUsers(users)
  } Then {
    buffer: mutable.Buffer[(String,Long,Long,String,String)] =>
    buffer.toList should equal (List(("2014/07/01", 1000002L,
      1L, "stefano@email.com", "10 Downing St. London")))
  }
}
```

Another unit test for the operation `logsCountVisits` is shown in the following code:

```
it should "count visits per day" in {
  Given {
    List(
      ("2014/07/01", 1000002L, "http://youtube.com"),
      ("2014/07/01", 1000003L, "http://twitter.com"),
      ("2014/07/01", 1000002L, "http://google.com"),
      ("2014/07/01", 1000002L, "http://facebook.com")
    ) withSchema ('day, 'user, 'url)

  } When {
    pipe: Pipe => pipe.logsCountVisits
  } Then {
    buffer: mutable.Buffer[(String, Long, Long)] =>
      buffer.toSet should equal (Set(
        ("2014/07/01", 1000002L, 3L),
        ("2014/07/01", 1000003L, 1L)
      ))
  }
}
}
```

The preceding unit tests independently test all the external operations. The benefits of unit testing modular code are more evident when applied to more complex Scalding applications.

Implementing the MapReduce logic

Following the test-driven methodology, at this point we compile and execute the integration and unit tests only to see them failing:

```
chapter6 $ mvn test
```

> Using the maven plugins *scalatest-maven-plugin* and *maven-surefire-plugin*, we can execute both integration and unit tests easily with maven.

We now need to provide the complete implementation of the external operations, as shown in the following code:

```
val fmt = org.joda.time.format.DateTimeFormat
  .forPattern("dd/MM/yyyy HH:mm:ss")

def logsAddDayColumn : Pipe = pipe
  .map('datetime -> 'day) { date: String =>
    fmt.parseDateTime(date).toString("yyyy/MM/dd")
  }

def logsCountVisits : Pipe = pipe
    .groupBy(('day, 'user)) { _.size('visits) }

def logsJoinWithUsers(userData: Pipe) : Pipe = pipe
    .joinWithLarger('user -> 'user, userData)
    .project(LOG_DAILY_WITH_ADDRESS)
```

Completing the preceding implementation and re-executing the unit and integration tests asserts the correctness of our code.

Defining and performing system tests

We implemented unit and functional tests using small amounts of predefined data samples. Now, it's time to execute the same Scalding application on the Hadoop cluster to work with a few GB of data.

Huge amounts of data can smoke test and reveal blockage points in our MapReduce applications. Operations such as `toList` and `mkString` may result in heap space error if misused, and only massive data can ensure that we have tested the application completely.

If the application runs successfully, we can start evaluating the performance. Sometimes, making a simple change, such as adding a `unique` operation before a join, can result in major performance gains. So, here is the opportunity to enhance performance.

Black box testing

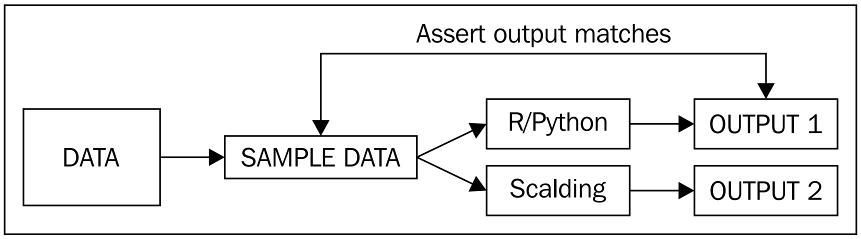
During test-driven development, we retain an internal perspective of the system. We identify all possible paths and exercise them through test case inputs to validate the expected output. However, using only valid input is not sufficient, especially when implementing MapReduce applications that execute against possibly billions of lines of data. As we cannot generate all possible cases of invalid input, we look at techniques that increase the data coverage of tests.

Taking a step back, the development lifecycle begins with data exploration followed by the algorithm design. Having a data scientist performing these tasks in a non-scalable development language such as R or Python is the basis of black box testing. Data scientists use multiple tools to extract meaning, insights, and ultimately, value from data. These tools provide powerful capabilities and rich visualizations that enable them to quickly conclude into mathematical models. The drawback is that the resulting implementation is not scalable.

We can easily generate a sample dataset in Scalding with the following code:

```
class ExampleJob(args: Args) extends Job(args) {
  val visitsPerDay = Tsv(args("input"), LOG_SCHEMA).read
   .sample(0.001)
   .write(Tsv(args("sample-input"), LOG_SCHEMA))
}
```

Running the preceding job generates a new logfile that consists of 0.1 percent of the original dataset. We can use this sample data against both our Scalding application and the Python implementation, and assert that both produce the same result:



The `sample(0.001)` operation generates a different sample every time, and we can re-execute the black box testing job to increase our data coverage.

Another benefit of black box testing is that a data scientist validates the outcome of a developer while the developer validates the assumptions and the correctness of the implementation of the data scientist at the same time.

Summary

In this chapter, we presented the development life cycle that follows the test-driven strategy. We went through the process of defining acceptance tests, and we implemented and executed integration and unit tests. Then we presented, ideas on using black box testing to increase our data coverage.

By testing every layer of the application, the number of bugs is reduced, maintainability enhances, and productivity increases. Scalding offers remarkable testing capabilities, satisfying all the requirements to build robust and complex MapReduce applications.

In the following chapter, we will discuss how to run our jobs on a production cluster and how to configure, monitor, and optimize them.

7 Running Scalding in Production

We now know how to implement complex pipelines, use appropriate design patterns, and test our Scalding data processing applications in multiple layers. In this chapter, we will look at how to productionize a job, and more specifically, we will see how to:

- Deploy, execute, and schedule
- Coordinate task execution
- Configure using property files and Hadoop parameters
- Monitor and optimize

Executing Scalding in a Hadoop cluster

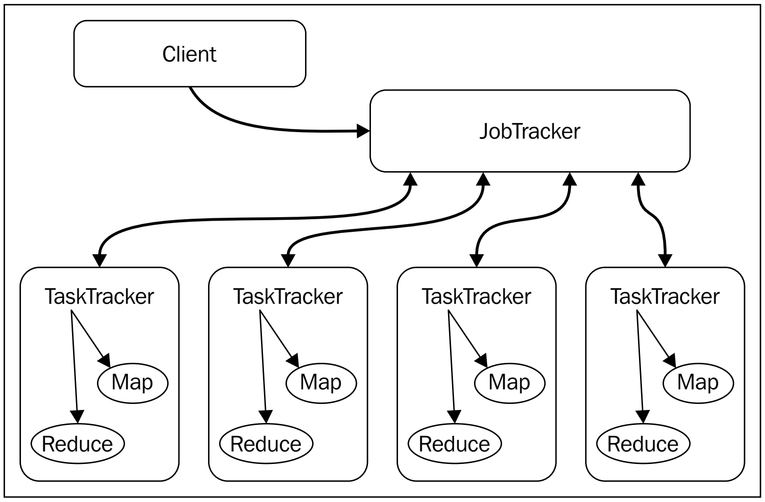
Deploying an application requires using a build tool to package our application into a jar file and copying it to a client node of the Hadoop cluster. The process of execution is straightforward and is very similar to submitting any JAR file for execution on a Hadoop cluster, as shown in the following command:

```
$ hadoop jar myjar.jar com.twitter.scalding.Tool mypackage.MyJob
 --hdfs --input /data/set1/ --output /output/res1/
```

The submitted job has the same permissions in HDFS as the user that submitted the job. If the read and write permissions are satisfied, it will process the input and store the resulting data.

> Scalding applications, when storing in HDFS, write data to the output folder defined in a sink in our job. Any existing content on that folder is purged every time a job begins its execution.

Internally, the JAR file is submitted to the JobTracker service that orchestrates the execution of the map and reduce phases. The actual tasks are executed in the TaskTrackers, as shown in the following diagram:



Scheduling execution

Data processing applications usually run frequently. Some run once a day and others run every few hours or minutes. The following is a list of some tools that can be used for job scheduling:

- **Cron**: The time-based job scheduler in Unix-like operating systems. It is not a very sophisticated tool but suffices for scheduling few jobs.
- **Jenkins**: The continuous integration tool that offers scheduling via a cron-like mechanism. It also preserves the history and messages, can send e-mail notifications, and use version control. It is capable of scheduling thousands of job executions per day.
- **Oozie**: The official workflow scheduling system to manage Apache Hadoop jobs. It is a scalable, reliable, and extensible system, and it is built specifically to allow workflow scheduling. The downside is that it is based on XML configuration files that can easily grow in size and complexity.

- **Azkaban**: The batch workflow job scheduler created at LinkedIn to run Hadoop jobs. It solves the problem of ordering job dependencies and provides an easy-to-use web user interface to maintain and track the workflows.

Each scheduling solution offers different capabilities. The easiest tool to use for a few executions is cron. Jenkins is another tool that most developers are already familiar with. Oozie is the tool best integrated into Hadoop. Azkaban overall provides very good visibility over workflows.

When using a script-based approach for scheduling, a good tip is to trap errors. Exit code statuses are not propagated in scripts by default. This means that if a script fails to execute commands, but the very last command succeeds, the script will return success as an exit status. Add the following trap function at the beginning of a script to capture all errors:

```
# Configure bash behavior
set -o pipefail         # Trace errors through pipes
set -o errtrace         # Trace ERR through 'time command'

error() {
  JOB="$0"              # job name
  LASTLINE="$1"         # line of error occurrence
  LASTERR="$2"          # error code
  echo "ERROR in ${JOB}:line ${LASTLINE} - exit code ${LASTERR}"
  exit 1
}
trap 'error ${LINENO} ${$?}' ERR
# hadoop jar ...
```

Coordinating job execution

A lot of times, we need to coordinate the execution of Scalding jobs and even mix them with other applications. For example, let us assume that we have implemented two Scalding jobs and one Scala application, as shown:

```
class JobA (args: Args) extends Job (args) { /*pipeline*/ }
class JobB (args: Args) extends Job (args) { /*pipeline*/ }
object ScalaApp extends App { /*Application logic*/ }
```

To ensure that we execute the preceding tasks in a predefined order, we can implement a *runner* class. A runner class implemented in Scala should extend `App` in order to work in the imperative programming style. This means that commands are executed synchronously and sequentially.

We can use this to our advantage and coordinate the execution of MapReduce tasks, other applications, and even external system commands, such as shown in the following example code:

```
object ExampleRunner extends App {

  val runnerArgs = Args(args)
  val configuration = new org.apache.hadoop.conf.Configuration

  // Executing a [Scalding] Job - A
  ToolRunner.run(configuration, new Tool,
    (classOf[JobA].getName :: runnerArgs.toList).toArray )

  // Executing a [Scala] application
  ScalaApp.main(null)

  // Executing external system command
  import sys.process._
  "ls -la" !

  // Executing [Scalding] Job - B
  ToolRunner.run(configuration, new Tool,
    (classOf[JobB].getName :: runnerArgs.toList).toArray )

}
```

To execute the JAR file that contains the preceding code and all the dependencies in Hadoop mode and in order to parallelize the execution of the MapReduce tasks, we execute the following command:

```
$ hadoop jar jar-with-dependencies.jar ExampleRunner --hdfs
```

Configuring using a property file

To read configuration data from a property file, we can use the configuration library for JVM languages from *typesafe*. This is mostly a Scala feature, but an interesting example is running Scalding applications in two different clusters.

While during developing and testing, we execute our Scalding application in a *development* cluster, when the jobs are production ready, they are executed in a powerful *production* Hadoop cluster.

In this case, each cluster will have different configuration requirements. To solve this issue, first define two property files as shown:

```
dev-cluster.properties
production-cluster.properties
```

Each file contains environment and job configuration information. For example, the file `dev-cluster.properties` can contain the following:

```
mysql=dev.mysql.company.com
zookeeper=dev.zookeeper.company.com:2181
reducers=30
```

The best way to implement this capability (keeping in mind the single responsibility principle) is in a different class called `JobBase`. In this class, we can inject the capability and extend the `com.twitter.scalding.Job` class, as is shown in the following code:

```
class JobBase(args: Args) extends Job(args) {

val appConfig = com.typesafe.config.ConfigFactory.parseFile(new java.
io.File(getString("cluster-config")))

  def getString(key: String): String = {
    args.m.get(key) match {
      case Some(v) => v.head
      case None => sys.error(f"Argument [$key%s] - NOT FOUND")
    }
  }
}
```

Scalding applications that need the capability to read configuration values from property files can now extend `JobBase` instead of `Job`:

```
class ExampleJob(args: Args) extends JobBase (args) {

   println("MySQL    : " + appConfig.getString("mysql"))
   println("ZooKeeper: " + appConfig.getString("zookeeper"))
   // val pipe = ...
}
```

We can now pass the path through a `property` file using a parameter at execution time:

```
$ hadoop jar chapter7.jar com.twitter.scalding.Tool
externalconfiguration.ExampleJob --hdfs --cluster-config dev-cluster.
properties
```

Configuring using Hadoop parameters

There are many Hadoop configuration parameters that can be tuned at job execution. A set of default values is assigned at execution time, based on Hadoop configuration files. We can, however, overwrite the default values.

We can, for example, set the amount of memory allocated to each map and reduce the task of that job as well as the default number of reduce tasks per job. Note that all Hadoop parameters have to be added right after `com.twitter.scalding.Tool`, as in the following example:

```
$ hadoop jar myjar.jar com.twitter.scalding.Tool \
 -D mapred.child.java.opts=-Xmx2048m \
 -D mapred.reduce.tasks=20 \
 com.company.myclass \
 --hdfs --input $input --output $output
```

Perform a search on the web for *map reduce client default values* to find out more information about the available Hadoop parameters that can be used.

Monitoring Scalding jobs

A web application that helps us visualize the operational details around all phases of our Scalding applications such as application development, debugging, performance tuning, and operator monitoring is **Driven**. This application is developed by **Concurrent**, the same company that developed and open sourced Cascading.

Driven (`http://driven.cascading.io/`) is a free cloud service that receives and visualizes telemetry data from running Scalding applications.

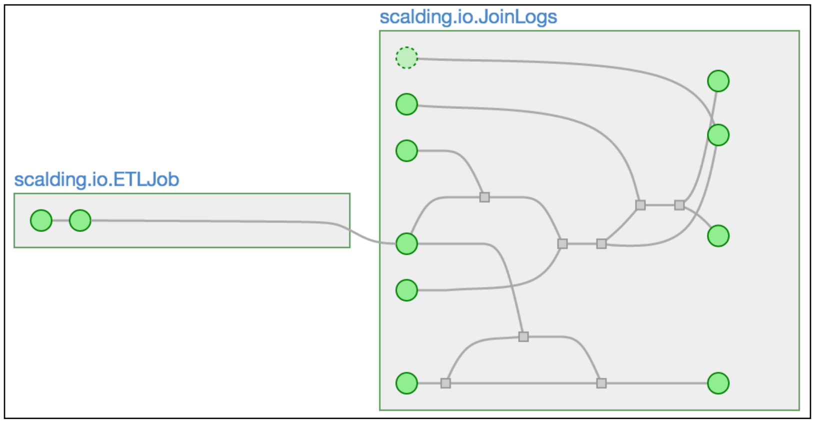
To enable this, we need to include the following plugin:

```
<dependency>
  <groupId>driven</groupId>
  <artifactId>driven-plugin</artifactId>
  <version>1.0-eap-59</version>
  <classifier>io</classifier>
</dependency>
```

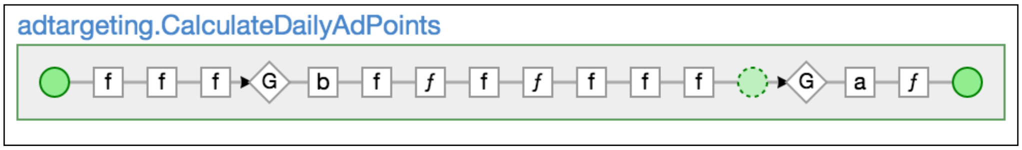
Then, create a new account and receive an appropriate API key that can be defined as a system variable, as shown:

```
$ export DRIVEN_API_KEY=D991A15E7A174E098900CDEE4F3A3CA6
$ hadoop jar ...
```

Driven provides both high-level and low-level representations of our Scalding data processing applications.



Executing the example presented in *Chapter 4*, *Intermediate Examples*, we can see in the web interface of Driven how long the job took to complete and also the physical view of our job:



We can also drill down into our job to discover the MapReduce phases that the job was converted into, the number of tasks, and the time spent. We discover that the job for processing 5 GB of input data is executed as follows:

```
Map phase | Reduce phase | Map phase  | Reduce phase
 39 tasks |  222 tasks   | 222 tasks  |  222 tasks
 50 sec   |   24 sec     |  20 sec    |   26 sec
```

If we review the job execution, the first map phase is optimized. 39 map tasks are exactly the number of tasks required to read a file of 5000 MB (recall that each block is 128 MB, and 39 times 128 equals 5000).

The second and third phases of the job can be improved. 222 tasks for the reduce and map phases means that each task will process approximately 22.5 MB (5000 MB divided by 222 equals 22.5). Setting that number to 100 tasks would not degrade performance and would use fewer resources from the Hadoop cluster.

The fourth phase of our job is the least optimized. If we recall the code, the final group operation was `group.sum[Int]('points)`. This means that the resulting output will be minimal. Indeed, if we check in HDFS after executing the job, we will discover that the 222 reduce tasks generate 222 part files `part-00000` and `part-00001 ... part-00221`, and the size of each output file will be just 201 KB.

This final phase can optimally be executed in a single reduce task that stores all the results in a single 43 MB file.

> HDFS prefers few and large files. The service **NameNode** keeps the directory and file tree in-memory. This makes it difficult to scale HDFS beyond a few tens of millions of blocks/files.

This optimization can be done by using the `reducers` group operation, as shown:

```
group => group.sortBy('epoch).reverse.reducers(100)
group => group.sum[Int]('points).reducers(1)
```

Using slim JAR files

The majority of articles and tutorials recommend developers to package all the dependencies and the application code into a single JAR file. This is known as the *fat jar* approach and can be achieved using maven or sbt.

A build tool can generate the JAR file with a single command, such as `mvn package`, once we have all the appropriate plugins, such as `maven-assembly-plugin`, in place.

This process is awesome, until we have to deal with the compiler or the deployment process more than once a day. Assembling a single distributable archive takes time. The plugin needs to iterate through all the project dependencies, uncompressing every single dependency and aggregating the project output along with its dependencies, modules, and other files.

Scalding applications depend on Hadoop libraries, the Scala library, Cascading libraries, and other utility libraries. The dependency hierarchy means that the resulting JAR files occupy between 60 MB and 100 MB, depending on the amount of libraries used. So, there is an extra overhead of storing or transferring it over the network.

A solution to this problem is to generate *slim jar* files with a size of just a few KBs. To achieve this, we need to set the dependencies in the following two places:

- In `HADOOP_CLASSPATH` at execution time
- In the classpath of each map and reduce task

Our first task will be to use the plugin `maven-dependency-plugin` to copy all dependency libraries in a specific folder. Provided we store all the dependencies in the `libs` folder, we can easily set the `HADOOP_CLASSPATH` environment variable at execution time, as shown:

```
$ SLIMJAR=target/chapter7-0.jar
$ for f in libs/*.jar; do CP=$CP:$f; done
$ export HADOOP_CLASSPATH=$SLIMJAR;$CP
```

The next task is to put all the external dependencies into an HDFS folder, as follows:

```
$ hadoop fs -mkdir -p /project1/libs/
$ hadoop fs -put libs/* /project1/libs/
$ hadoop fs -put $SLIMJAR /project1/libs/
```

Then, we can execute the slim (23 KB) JAR file with the following:

```
$ hadoop jar $SLIMJAR slimjar.JobRunner slimjar.ExampleJob --hdfs
--heapInc --libjars project1/libs/
```

In the preceding command, we use the `JobRunner` class to initiate the job. This class looks for the parameter `--libjars`. If it exists, it loads all the `*.jar` files located at the HDFS folder specified in that parameter into the distributed cache and the job configuration.

This is achieved with the following code block, in the `JobLibLoader` class:

```
JobLibLoader.loadJars(hadoopPath, conf)
conf.addResource(hadoopPath)
```

`JobLibLoader` reads the dependencies at execution time. What is interesting about this technique is that although the dependency files need to be both in the local filesystem and in HDFS, they change much less frequently than our own code.

> Find the complete project code, including `JobLibLoader`, at `http://github.com/scalding-io/ProgrammingWithScalding`.

Scalding execution throttling

Scalding execution throttling is a Hadoop-specific trick. It makes sense to highlight it here as we may read billions of rows of data when running Scalding applications in production.

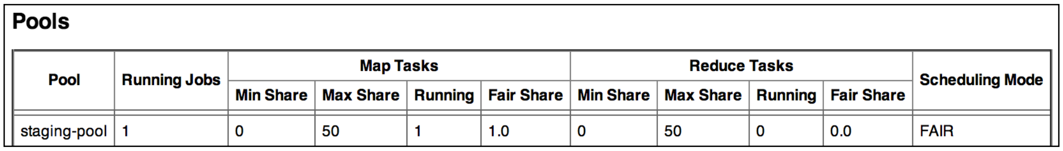
For resource management, Hadoop offers a number of schedulers. Each cluster has a specific capacity, for example 600 simultaneous map tasks and 300 reduce tasks. The most common scheduler used in Hadoop is the Fair Scheduler. It attempts to assign resources to jobs so that in average they get an equal amount of resources.

There are occasions, however, when we will want to protect some resources for business critical jobs, or we will want to throttle some job. Sometimes, we may need to limit resources to newer members of the team, or limit resources on a new beta release of an application.

For this, we can access the JobTracker using `ssh` and add a new pool in the file `fair-scheduler.xml`, as shown in the following code:

```
<pool name="staging_pool">
  <maxMaps>50</maxMaps>
  <maxReduces>50</maxReduces>
</pool>
```

The allocation file is reloaded periodically at runtime, allowing us to change pool settings without restarting any Hadoop service. We can verify the existence of a new pool called *staging-pool* with maximum allocation of resources at JobTracker. For more information, check `http://localhost:50030/scheduler`.



Pools

Pool	Running Jobs	Map Tasks				Reduce Tasks				Scheduling Mode
		Min Share	Max Share	Running	Fair Share	Min Share	Max Share	Running	Fair Share	
staging-pool	1	0	50	1	1.0	0	50	0	0.0	FAIR

To allocate a job to the new pool, we use the following Hadoop parameter at execution time:

```
-Dmapred.fairscheduler.pool=staging-pool
```

Summary

In this chapter, executing and scheduling jobs were presented. As our data processing workflows can depend on a number of applications, we showed how to chain together and coordinate the execution of a number of tasks. Then, we proceeded with configuring our jobs using both property files and Hadoop parameters.

Monitoring and optimizing our job execution was also presented. Finally, two more techniques we presented about using slim jars were to optimize the deployment process and how to throttle job execution.

In the next chapter, we will see how to use external data sources to read data from and store data to.

8

Using External Data Stores

By now, we have presented how to run MapReduce applications on data present in files and how to use external services to enhance data.

On top of this, Scalding offers a rich set of capabilities regarding interaction with external systems. In this chapter, we will present:

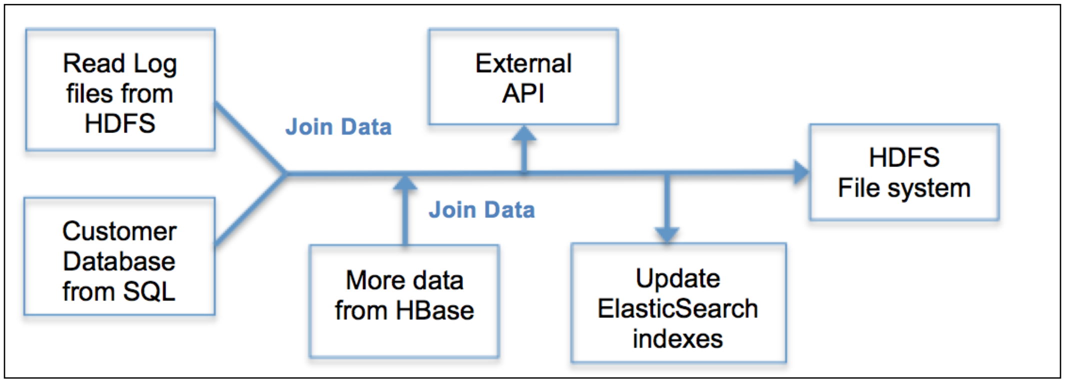
- How to read and write in SQL databases
- How to read and write in NoSQL databases
- Using search and analytics engines

Interacting with external systems

Scalding allows us to build rich pipelines that read data from one or more sources, perform data transformations, and store results into one or more sinks. The sources and the sinks are called *taps*.

With Scalding, we can tap into the HDFS filesystem. A characteristic of HDFS is that it does not allow appending to files. Once a file is closed, it is immutable and can only be changed by writing a new copy with a different filename. This style of file access fits nicely with MapReduce and batch processing jobs.

There are, however, use cases where data changes very frequently, or fast response times are required for real-time applications. The use cases fit nicely with in-memory systems. Fortunately, Scalding can tap to multiple external data stores, and thus, elaborate pipelines can be achieved:



Scalding supports interaction with SQL, NoSQL, and in-memory systems either through external libraries or by using wrappers over Cascading libraries. Moreover, with Scalding, external data stores can be used both to read and write data. This chapter presents thorough examples of how this can be achieved.

SQL databases

It is a common scenario for a Scalding job to process files from HDFS and join them with data fetched from a SQL database. Similarly, we will often have to implement a MapReduce job that writes some results into a SQL database.

For SQL, and in the context of MapReduce, we are interested to have support for all access patterns, many SQL dialects, and also batch capabilities. Batching is the technique of aggregating multiple, possibly hundreds of SQL statements and executing them as a single batch command into the database system.

The latter is very important as a MapReduce application can easily scale to hundreds of Java virtual machines, running the map and reduce tasks. Having hundreds of nodes trying to communicate with a database system at the same time can stress the system to its limits.

In SQL, the available *access patterns* are as follows:

- **SELECT**: This is used to select data from a database and add them into a pipe
- **INSERT**: This is used to insert new records from a pipe into a database
- **UPDATE**: This is used to update existing records in a database with new values
- **UPSERT**: This is used to `INSERT` new records or `UPDATE` existing records
- **DELETE**: This is used to delete existing records from a database

Several SQL *dialects* are used by proprietary and open source database systems. They mostly adhere to the SQL standard, but there is some misalignment on commands that are useful while splitting data, such as `LIMIT` and `OFFSET`.

Fortunately, multiple **JDBC** (**Java Database Connectivity**) tap implementations exist. Cascading, since Version 2.2, announced that moving forward they will curate all the integrations into a single project called `cascading-jdbc` at `https://github.com/Cascading/cascading-jdbc/`.

This library currently supports six database systems: H2, Derby, Oracle, MySQL, PostgreSQL, and Amazon Redshift. The project is modular, and adding support for more databases is possible by following instructions found in the provided documentation.

To use `cascading-jdbc` in Scalding, we need to include the appropriate library and a Scalding wrapper into our dependency tool. For example, use the following in Maven:

```
<dependency>
  <groupId>cascading</groupId>
  <artifactId>cascading-jdbc-mysql</artifactId>
  <version>2.5.1</version>
</dependency>
<dependency>
  <groupId>com.twitter</groupId>
  <artifactId>scalding-jdbc_2.10</artifactId>
  <version>0.10.0</version>
</dependency>
```

To connect a Scalding pipe with a database, we can use the `JDBCSource` wrapper and specify an object with connectivity details:

```
case object MySQLTableTap extends JDBCSource {
  override val tableName = "tableName"
  override val columns = List(
    varchar("user", 16),
    date("time"),
    varchar("activity",256),
    smallint("code")
  )

  val connectUrl = "jdbc:mysql://localhost:3306/testdb"
  val dbuser = "user"
  val dbpass = "password"
  val adapter= "mysql"
  override def currentConfig = ConnectionSpec (connectUrl, dbuser,
      dbpass, adapter)
}
```

To use this new `MySQLTableTap`, we need a pipe with a schema similar to the SQL table that we want to read from or write to. For every column in the table, the relevant field should exist in the pipe. We can use the tap to write to the database table with the following code:

```
val schema = List('user, 'time, 'activity, 'code)
pipe
  .project(schema)
  .write(MySQLTableTap)
```

Similarly, we can read from the database table with the following code:

```
val read_mysql = MySQLTableTap
  .read
  .write(Tsv("jdbc-output"))
```

By default, the `JDBCSource` batches up to 1000 requests before contacting the database to minimize the overhead to the external system.

NoSQL databases

Another common scenario is to read, insert, or update existing data in a NoSQL database. Such systems are usually used to drive real-time applications such as an Analytics platform, or store and provide traversal capabilities to a graph database.

Fortunately, a lot of taps are available, and Cascading provides a number of extensions for popular NoSQL databases such as MongoDB, Cassandra, ElaphantDB, and HBase. A brief introduction to these NoSQL systems follows:

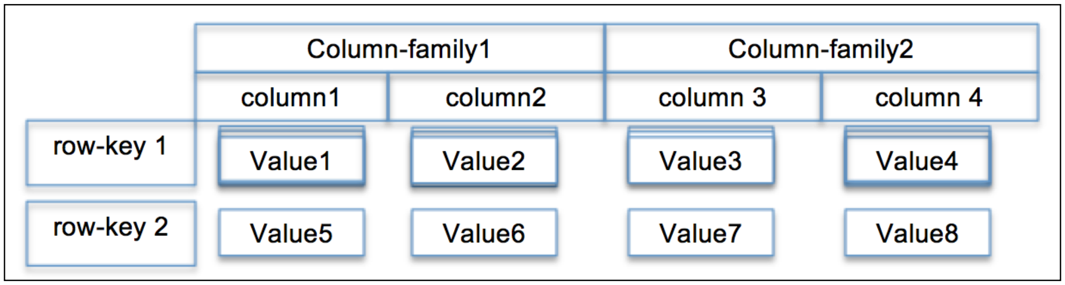
- **MongoDB**: This is a document-oriented database that uses JSON-like documents with dynamic schemas and is the most popular NoSQL database.
- **Cassandra**: This is a highly distributed database capable of spanning over multiple data centers that aims to provide low latency for real-time applications. It uses a flat hierarchy across nodes architecture and is not dependent on Hadoop applications or HDFS.
- **ElephantDB**: This is a distributed database specializing in exporting key-value data from Hadoop. The library `elephantdb-cascading` allows interacting with this data store from Cascading workflows.
- **HBase**: This is a distributed, versioned, and non-relational database modeled after Google's Bigtable. It builds directly on top of the capabilities of Hadoop and HDFS.

Each NoSQL database provides unique features related to sharding, indexing, sorting, and internal data representation. Scalding interoperation with NoSQL databases is continuously improving, as integration with MapReduce pipelines is becoming a highly desirable feature. In the following section of the book, we will first present the specifics of HBase, and then show how to use Scalding to interact with this distributed database.

Understanding HBase

HBase is the Hadoop database. It is a platform to store and retrieve data with random access, which means that we can write data as we like and read the data back again as we need it. It can store structured and semi-structured data similar to those held in a SQL database, such as the products and the customer reviews of an e-commerce site. It can store unstructured data too. It does not care about types and allows for a dynamic and flexible data model that does not constrain the kind of data we store. Thus, it does not mind storing one integer in one row and a string in another for the same column.

Similar to Cassandra, HBase can be described as a key-value store, but for the value, it can have multiple columns of data. Multiple *columns* of data can also be grouped into *families*. Values can be stored in multiple dimensions. The default number is three, but we can control this as well, which means that we can retain a full history of values changing over time.



Reading and writing to HBase works amazingly fast when we have many nodes, each contributing CPU, memory to the distributed cache, and disk storage. As a distributed system, each node in the cluster holds different *regions* of data in disk and in memory. HBase utilizes caching to preserve most frequently requested items in cache.

Internally, data is sorted based on the key, and the design of the row-key is of upmost importance. So, data in HBase is sorted *lexicographically* based on the row-key. The point is that monotonically increasing values are bad. When saving entities to HBase at a high write rate, we must avoid using monotonically increasing keys as they will hit the same nodes in our cluster continuously due to how data is split into shards.

Reading from HBase

In this section, we will present the Scalding library for HBase with advanced features. Visit `https://github.com/ParallelAI/SpyGlass for` more information.

SpyGlass can easily read data from HBase when we know exactly which keys we are interested in. To do this, we need to define a new tap and specify the following:

- Which table to read from
- How to communicate with *zookeeper*, the service that knows how the database is distributed over the cluster
- In which Scalding field to store the row-key
- Which column families to read
- Which columns of the preceding families to read

- The operation to be performed is a `GET` request
- How many versions of each value to read
- Which keys are we interested in

Once we know what we need to read, structuring an `HBaseSource` is straightforward, as shown in the following code:

```
val hbaseSource = new HBaseSource(
   "table_name",
   "zookeeper.quorum.ip:2181",
   'key,
    Array("column_families"),
    Array('column_names),
    sourceMode = SourceMode.GET_LIST,
    versions = 5,
    keyList = List("5003914", "5000687", "5004897"))
```

In the preceding example, the five most recent versions of the values stored in the specific column families and columns are retrieved for the three keys specified. In a real-world scenario, we can use a single tap to read up to a few hundred thousand values.

HBase stores data internally using an `ImmutableBytesWritable` format. However, we want data to flow as strings or other objects inside our Scalding pipes. To convert to such a format when reading data, we can use the `fromBytesWritable(schema)` command to automatically translate data into plain strings. Similarly, before writing into HBase, we can use the `toBytesWritable(schema)` command to translate all the current data in the pipe into the HBase format.

So, in order to read data from `HBaseSource` into a Scalding pipe, we can use the following code:

```
val schema = List('key, 'column1, 'column2)
val pipe = hbaseSource.read.fromBytesWritable(schema)
```

HBase lacks *query* commands of any kind. However, it effectively provides the capability to *scan*. We can scan the data from a particular key up to another key, and apply a filter to retrieve only the relevant records.

```
val hbaseSink = new HBaseSource(
   "table_name",
   "zookeeper.quorum.ip:2181",
   'key,
    Array("column_families"),
    Array('column_names),
    sourceMode = SourceMode.SCAN_RANGE,
    startKey = "5003914",
    stopKey = "5010000")
  .read
  .fromBytesWritable(schema)
  .filter('action) { x:String => x="login" }
  .write(Tsv("results")))
```

Writing in HBase

SpyGlass can be used to write new data or update existing data in HBase. Versions in the database tables are nothing more than values associated with a timestamp. The most recent timestamp is also the most recent version. This is a handy feature when building a system for ad-targeting. We can store data on a daily basis but also retain older versions. To specify the version, use the timestamp parameter. If this parameter is not present, then the current time is used by default.

```
Val schema = List('rowkey,'artistname,'country)
val hbaseSink = new HBaseSource(
   "table_name",
   "zookeeper.quorum.ip:2181",
   'key,
    schema.tail.map((x: Symbol) => "data").toArray,
    schema.tail.map((x: Symbol) => new Fields(x.name)).toArray,
    )
pipe
  .toBytesWritable(schema)
  .write(hbaseSink)
```

The byte transformation capabilities are provided in a trait called `HBasePipeConversions` that we need to inherit into the job:

```
class HBaseJob (args: Args)  extends Job(args)
  with HBasePipeConversions { /* scalding code */ }
}
```

Using advanced HBase features

SpyGlass provides more HBase-specific capabilities, which are as follows:

- Deleting from HBase by specifying a list of row-keys to be deleted. This is achieved using the `SinkMode.REPLACE` mode that deletes all rows, including all versions.

- Preventing the common problem of *region hot spotting* with HBase by providing a simple and configurable hashing capability.
- Scanning capabilities on hashed row-keys. When using hashing, row-keys are distributed equally in a cluster, achieving maximum performance. However, the distribution makes it difficult to perform sequential scans. This problem is solved with the `SplitType.REGIONAL` mode.

Detailed examples can be found at `https://github.com/ParallelAI/SpyGlass`.

Search platforms

The Apache **Lucene** library provides Java-based indexing and search technology as well as spellchecking, hit highlighting, advanced analysis, and tokenization capabilities. There are two popular open source projects that use this library and provide a distributed platform with replication and caching capabilities.

Solr has been an Apache open source project since 2006, and thus, it has been used by many enterprises and has grown and improved as a project. **ElasticSearch** was released a few years later, and it was designed since the beginning to be distributed and easy-to-scale out to handle massive amounts of data.

As distributed systems, they both fit nicely in the Hadoop environment. Nodes that participate in the cluster can run both the Hadoop applications—an HBase database and a search platform.

As high memory nodes are usually in place, we can allocate enough memory to each system. Then, depending on the job running, we can utilize the processing and caching capabilities of the hardware as much as possible.

Elastic search

Elasticsearch is a powerful search and analytics engine that makes data easy to explore. Integration with Hadoop with support for MapReduce, Cascading, Hive, and Pig is provided through the library `elasticsearch-hadoop` available at `https://github.com/elasticsearch/elasticsearch-hadoop`.

We will implement a Scalding wrapper for elastic search as an exercise. To implement `ElasticSearchTap` that can read and write to elastic search, we need to create a class that extends the `Source` class and overrides the method `createTap`, as shown in the following code:

```
case class ElasticSearchTap (
   esHost : String,
   esPort : Int,
   esResource : String,
   esQuery : String,
   esFields : Fields)
  extends Source {

  def createTap: Tap[_, _, _] =
    new EsTap(esHost, esPort, esResource,esQuery,esFields)

  override def createTap(readOrWrite: AccessMode)
    (implicit mode: Mode): Tap[_, _, _] = {
    mode match {
      case Local(_) | Hdfs(_, _) => { createTap }
    }
  }
}
```

The preceding code is a working tap to read and write to elastic search. To use it, we need to define the host and the port that the elastic search server is listening to, and also define the resource, that is, the index and the type that we need to access. Finally, we have to define the fields we are interested in.

Writing to an elastic search server from Scalding is as simple as shown in the following code:

```
val schema = List('number, 'product, 'description)
val pipe = Tsv(arg("input"),schema).read
.write(ElasticSearchTap("localhost",9200,"index/data","",schema))
```

From the elastic search, we can read either the whole index or the part of it specified by a particular query as shown:

```
val query = "number:(>=10 AND < 20)"
val readDataFromElasticSearch =
  ElasticSearchTap("localhost", 9200, "index/data", query)
  .read
  .write(Tsv("results"))
```

A more advanced elastic search tap is available at `https://github.com/scalding-io/scalding-taps`.

Summary

In Big Data, the HDFS filesystem solves the storage and distribution of data on multiple nodes. MapReduce solves the problem of distributing execution and takes advantage of data locality. NoSQL databases solve the problem of driving real-time applications and storing frequently updated data efficiently. Some NoSQL databases by design, however, do not provide the commonly requested feature of *querying* the data. Distributed search platforms provide this capability.

As presented, Scalding is capable of taping into multiple systems. The ubiquity and expressiveness of the language make it a valid technology for completing tasks such as transferring data between SQL, NoSQL, or search systems. Given that the taps are also testable components, there are practically unlimited use cases where Scalding can be used to integrate various distributed systems.

In the next chapter, we will look at some advanced statistical calculations using matrix calculations, and we will see how Scalding can be used in machine learning applications.

9
Matrix Calculations and Machine Learning

In this chapter, we will look at matrix calculations and machine learning. The main differences between data processing applications, is that this chapter focuses on matrix and set algebra.

Machine learning requires understanding of the basic vector and matrix representations and operations. A vector is a list (or a tuple) of elements, and a matrix is a rectangular array of elements. The transpose of matrix A is a matrix that is formed by turning all the rows of a given matrix into columns.

We will use the above principles and present how Scalding can be utilized to implement concrete examples, including the following:

- Text similarity using term frequency/inverse document frequency
- Set-based similarity using the Jaccard coefficient
- Clustering algorithm using K-Means

Text similarity using TF-IDF

Term Frequency/Inverse Document Frequency (**TF-IDF**) is a simple ranking algorithm useful when working with text. Search engines, text classification, text summarization, and other applications rely on sophisticated models of TF-IDF. The algorithm is based on term frequency (the number of times the term `t` occurs in document `d`) and document frequency (the number of documents in which the term `t` occurs). Inverse document frequency is the log of the total number of documents, `N`, divided by the document frequency.

The basic idea is that common words, such as the word *the*, should receive a smaller significance compared to words that appear less frequently in documents.

We will use a collection of 62 books as an example dataset. A document consists of the title of the book and the actual text. For example:

In Book A (ASHPUTTEL), the word *the* is repeated 184 times and the word *child* two times. In Book B (Cat And Mouse In Partnership), the word *the* is repeated 73 times and the word *child* five times. Overall, the word *the* exists in all 62 documents whereas the word *child* exists in 27 of them.

Word	**TF Book A**	**TF Book B**	**Document Frequency**	**Inverse DF**
the	184	73	62	0
child	2	5	27	0.361

The inverse document frequency is obtained by dividing the total number of documents by the number of documents containing the term and then taking the logarithm of that quotient, as shown:

$$\boxed{\mathrm{idf}(t,D) = \log \frac{N}{\left|\{d \in D : t \in d\}\right|}}$$

In the occasion of the term *the*, the idf value is 0 (because *log(62/62) = 0*). This means that the term *the* adds no value in characterizing a document. Thus, we can safely ignore it in the book's similarity algorithm.

The tf-idf variable reflects how important a word is to a document. So, the word *child* is more important to Book B and less important to Book A. It is calculated as shown:

$$\boxed{\mathrm{tfidf}(t,d,D) = \mathrm{tf}(t,d) \times \mathrm{idf}(t,D)}$$

To calculate the preceding values in Scalding, we need the following:

- A pipe to transform the original dataset into tuples of `'book` and `'word`
- A pipe to calculate the term frequency
- A pipe to calculate the inverse document frequency
- A pipe to calculate the tf-idf score

The complete code that performs all the required calculations including counting the population in the data is as follows:

```
val inputSchema = List('book, 'text)
val books = Tsv("data/books.txt", inputSchema).read
  .flatMap('text -> 'word) { text:String =>
     text.toLowerCase.replaceAll("[^a-zA-Z0-9\\s]","")
         .split("\\s+").filter(_.length > 0)
  }
  .project('book, 'word)

val numberOfBooks = books.unique('book)
  .groupAll { _.size('numberOfBooks) }

val tf = books.groupBy('book, 'word) { _.size('tf) }
  .project('book, 'word, 'tf).crossWithTiny(numberOfBooks)

val df = tf.groupBy('word) { _.size('df) }

val tfidf = tf.joinWithSmaller('word -> 'word, df)
  .map(('tf, 'df) -> 'idf) { x:(Int,Int) =>
      x._1 * math.log(numberOfBooks / x._2)
  }
  .filter('tfidf) { x:Double => x > 0}
  .project('book, 'word, 'tfidf)
```

The preceding statistics can now be used to compute the similarity between the books. Cosine similarity gives a useful measure of how similar two documents are likely to be in terms of the words they contain. To achieve this, we need to normalize the vectors and then use sparse matrix multiplication.

> In statistics, *normalization* is the process of adjusting values measured on different scales to a notionally common scale. For example, transform all the values into a range of 0 to 1.

This can be implemented using the Matrix API as shown:

```
import com.twitter.scalding.mathematics.Matrix._
val booksMatrix = tfidf.toMatrix[String,String,Double
    ('book, 'word,'tfidf)
val normedMatrix = booksMatrix.rowL2Normalize
val similarities = (normedMatrix * normedMatrix.transpose)
  .filter('row, 'col) { x:(String,String) => x._1 < x._2 }
  .groupAll { group => group.sortBy('similarity).reverse }
  .write(Tsv(args("output"))
```

In the preceding Scalding code, we import the matrix library in order to transform the existing `tfidf` pipe into a matrix. The matrix is then normalized, and the similarity is calculated as the product of the matrix with the transpose of the matrix.

The result is the similarities between all the possible pairs of books. This includes the calculation of the similarity between the items 1-1, 1-2, and 2-1. To deduplicate the results, we use the `filter` operation.

Finally, we group all the results and sort them in descending order, based on similarity.

Setting a similarity using the Jaccard index

Quite often, we have to work with sets of data in machine learning. Users *like* posts, *buy* products, *listen* to music, or *watch* movies. In this case, data is structured in the two columns: `'user` and `'item`.

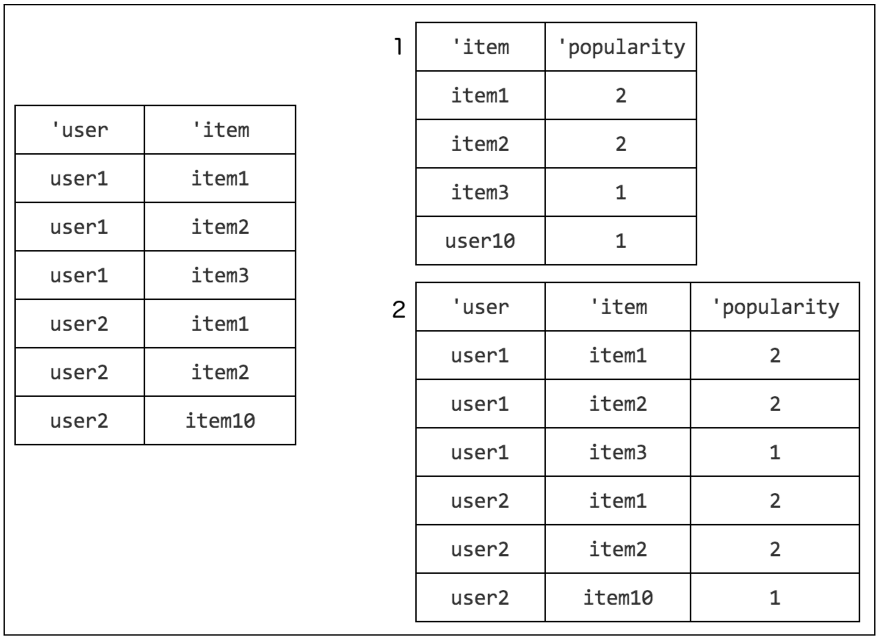
In order to calculate correlations, we need to work with sets. The Jaccard similarity coefficient is a statistic that measures the similarity between *sets*. The level of similarity is the calculation of the size of the intersection divided by the size of the union of the sample sets, as shown.

$$\boxed{\mathrm{J}(A,B)=\frac{|A\cap B|}{|A\cap B|}}$$

For example, if two users in the dataset are related to the same two items, and each user is also related to a distinct item, the Jaccard similarity indicates the following:

- The similarity between item1 and item2 is 100 percent
- The similarity between the common and distinct items is 50 percent
- The similarity between two distinct items is 0 percent

To begin the implementation, we first need to calculate the item popularity, and then add the popularity back to the data, as shown:



'user	'item
user1	item1
user1	item2
user1	item3
user2	item1
user2	item2
user2	item10

1

'item	'popularity
item1	2
item2	2
item3	1
user10	1

2

'user	'item	'popularity
user1	item1	2
user1	item2	2
user1	item3	1
user2	item1	2
user2	item2	2
user2	item10	1

This can be implemented with the following:

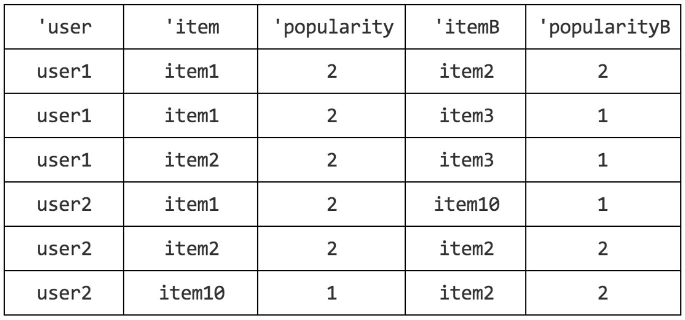
```
val pipe1 = inputPipe.groupBy('item) { _.size('popularity) }
val pipe2 = inputPipe.joinWithSmaller('item->'item, pipe1)
```

Once the item popularity is added to every line, we can generate a new pipe that contains all possible pairs of items. The `allItemPairs` pipe can be generated by joining `pipe2` with a clone of `pipe2` based on `'user`.

In the preceding diagram, `pipe2` contains two users. The inner join based on `'user` produces the complete product of all the possible pairs: (item1-item1), (item1-item2), and (item2-item1). As the similarity between the same items is 100 percent by default, and the similarity between (item1-item2) is the same as the similarity between (item2-item1), we can deduplicate using the `filter` operation, as shown in the following code:

```
val clone=pipe2.rename(('item,'popularity)->('itemB,'popularityB))
val allItemPairs = pipe2
    .joinWithSmaller('user -> 'user, clone)
    .filter('item, 'itemB) { x: (String,String) => x._1 < x._2 }
```

The resulting pipe `allItemPairs` contains all the possible pairs of items for every user and the popularity of each item, as shown:

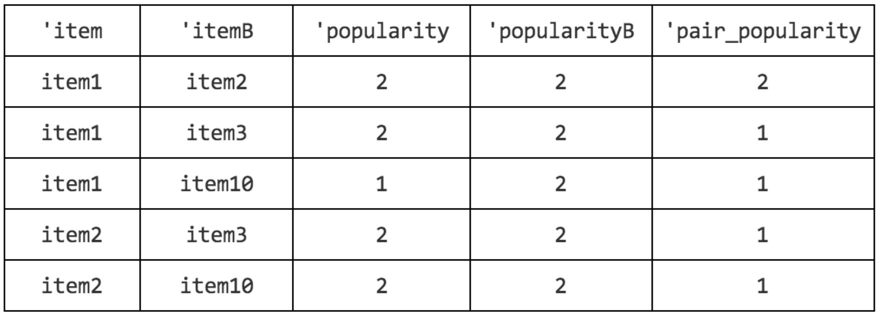


'user	'item	'popularity	'itemB	'popularityB
user1	item1	2	item2	2
user1	item1	2	item3	1
user1	item2	2	item3	1
user2	item1	2	item10	1
user2	item2	2	item2	2
user2	item10	1	item2	2

The following step requires calculating the pair popularity. This can be achieved with the `size` operation in `groupBy('item, 'itemB)`. As the column `'user` is no longer required, we will propagate only the popularity of each item, as shown in the following code:

```
allItemPairs.groupBy('item, 'itemB) { group => group
  .size('pairPopularity)
  .head('popularity)
  .head('popularityB)
}
```

The resulting pipe contains all the information required to calculate the Jaccard similarity, as shown:



'item	'itemB	'popularity	'popularityB	'pair_popularity
item1	item2	2	2	2
item1	item3	2	2	1
item1	item10	1	2	1
item2	item3	2	2	1
item2	item10	2	2	1

To calculate the final item-to-item similarity, we can use a map operation, as shown in the following code:

```
val jaccardPopularity = allItemPairs
  .map(('popularity, 'popularityB, 'pairPopularity)-> 'jaccard) {
     x:(Double, Double, Double) => {
        val item1Popularity = x._1
        val item2Popularity = x._2
        val pairPopularity = x._3

       pairPopularity / ( item1Popularity + item2Popularity
          - pairPopularity )
     }
  }
```

K-Means using Mahout

K-Means is a clustering algorithm that aims to partition `n` observations in `k` clusters.

Clustering is a form of unsupervised learning that can be successfully applied to a wide variety of problems. The algorithm is computationally difficult, and the open source project Mahout provides distributed implementations of many machine algorithms.

> Find more detailed information on K-Means at `http://mahout.apache.org/users/clustering/k-means-clustering.html`.

The K-Means algorithm assigns observations to the nearest cluster. Initially, the algorithm is instructed how many clusters to identify. For each cluster, a random centroid is generated. Samples are partitioned into clusters by minimizing a measure between the samples and the centroids of the cluster. In a number of iterations, the centroids and the assignments of samples in clusters are refined.

The distance *between* each sample and a centroid can be measured in a number of ways. *Euclidean* is usually used for samples in numerical space, and the *Cosine* and *Jaccard* distances are often employed for document clustering.

To provide a meaningful example, we will consider a popular application that users interact in unpredictable ways. We want to identify all abnormal behavior—users who manipulate services instead of legitimately using them. On such occasions, we don't really know what to look for.

The technique we will use is known as *outlier detection* and is based on the idea of generating a number of *features* for every user. Thinking as a data scientist, we can generate tens or hundreds of such features for each user, such as the number of sign-ins, clicks, periodicity between actions, and others.

K-Means can then be instructed to generate a single cluster. This will force the algorithm to calculate the center (centroid) of that single cluster. The centroid highlights the average and normal user behavior. Outliers (values that are "far away" from the cluster) are more interesting than common cases.

The implementation of the solution requires the chaining of Scalding and Mahout jobs.

In the following `KMeansRunner` application, we can perform these required steps:

1. Execute a Scalding ETL job that generates the user features and stores them into HDFS in a format that Mahout is compatible with.
2. Use Mahout to generate the initial centroid of the cluster.
3. Use Mahout to run the K-Means algorithm and calculate the centroid.
4. Run a Scalding job to calculate the Euclidean distance between the centroid and each user, and store results in descending order.

This can be implemented as follows:

```
object KMeansRunner extends App {

  val mArgs = Args(args)
  val configuration: Configuration = new Configuration

  ToolRunner.run(configuration, new Tool,
      (classOf[ETLJob].getName :: mArgs.toList).toArray )

  RandomCentroid.main(null)

  KMeans.main(null)

  ToolRunner.run(configuration, new Tool,
    (classOf[FinalJob].getName :: mArgs.toList).toArray )
}
```

The ETLJob processes logs and stores data in the Mahout vector [Text, VectorWritable], storing the user as text and the features as vectors. Mahout requires input data to be stored in sequence files so that the WritableSequenceFile object is used for storage. Thus, if a pipe contains a column with the users, and another column with the features, as a comma-separated string, we can generate the Mahout input with the following code:

```
val userFeatures=pipe.mapTo(('user,'features)->('user, 'vector)) {
  x: (String, String) =>
    val user = x._1
    val allFeatures = x._2.split(",").map(_.toDouble)

    val namedVector = new NamedVector(new DenseVector
        (allFeatures), user)
    val vectorWritable = new VectorWritable(namedVector)
        (new Text(user), vectorWritable)
  }

  val out = WritableSequenceFile [Text, VectorWritable]
     ("data/kmeans/mahout_vectors", 'user -> 'vector)
  userFeatures.write(out)
```

Then, RandomCentroid runs a Mahout job that takes the output of the previous job as input and generates one *random* centroid to initialize the next job. This is achieved with:

```
org.apache.mahout.clustering.kmeans.RandomSeedGenerator
  .buildRandom(conf,("data/kmeans/mahout_vectors",
  "data/kmeans/random_centroids", 1, new EuclideanDistanceMeasure)
```

Then, KMeans runs a Mahout job that uses the output of the two previous jobs and stores the resulting centroid to the filesystem after completing a maximum of 20 iterations as shown in the following code:

```
org.apache.mahout.clustering.kmeans.KMeansDriver.run(
  conf,
  new Path("data/kmeans/mahout_vectors"),
  new Path("data/kmeans/random_centroids"),
  new Path("data/kmeans/result_cluster"),   // OUTPUT_PATH
  0.01,  // convergence delta
  20,    // maximum number of iterations
  true,  // run clustering
  0,     // cluster classification threshold
  false) // run sequential
```

Mahout is peculiar in the way it stores output. Over a number of iterations, it generates results into folders `/clusters-0`, `/clusters-1` … `/clusters-N-final`, where *N* depends on the number of iterations.

However, `FinalJob` can easily access the final cluster results using the wildcard pattern `"/*-final"`. Then, we can extract the vector that holds the centroid of the final cluster from the `ClusterWritable` object as shown in the following code:

```
val finalClusterPath = "data/kmeans/result_cluster/*-final"
val finalCluster = WritableSequenceFile [IntWritable,
      ClusterWritable](finalClusterPath, 'clusterId -> 'cluster)

  val clusterCenter = finalCluster.read
    .map('cluster -> 'center) {
      x: ClusterWritable => x.getValue.getCenter
    }
```

By crossing the above pipe with the user's pipe, we can construct a new pipe that holds all the dates, including users, the vector containing the features of each user, and the vector of the center of the cluster.

We can then calculate the distance of each user from the center of the cluster and sort users in descending order, with the following pipeline:

```
val userVectors  = WritableSequenceFile [Text,
 VectorWritable]("data/kmeans/mahout_vectors", 'user -> 'vector)
    .crossWithTiny(clusterCenter)
    .map(('center, 'vector) -> 'distance) {
      x:(DenseVector, VectorWritable) =>
        (new EuclideanDistanceMeasure()).distance(x._1, x._2.get)
    }
    .project('user, 'distance)
    .groupAll { group => group.sortBy('distance).reverse }
    .write(Tsv("data/kmeans/result-distances.tsv"))
```

To execute this coordinated data processing application, run the following command:

```
$ hadoop jar chapter9-0-all-dependencies.jar kmeans.KMeansRunner --hdfs
```

> Find complete project files in the code accompanying this book at `http://github.com/scalding-io/ProgrammingWithScalding`.

Other libraries

For mining massive datasets, we can utilize the **Algebird** abstract algebra library for Scala, also open sourced by Twitter. The code was originally developed as part of the Scalding Matrix API. As it had broader applications in aggregation systems, such as Scalding and Storm, it became a separate library.

Locality Sensitivity Hashing is a technique that minimizes the data space and can provide an approximate similarity. It is based on the idea that items that have high-dimensional properties can be hashed into a smaller space but still produce results with high accuracy.

An implementation of the approximate Jaccard item-similarity using **Locality Sensitive Hashing** (**LSH**) is provided in the source code accompanying this book.

Another interesting open source project that integrates Mahout vectors into Scalding and provides implementations of Naive Bayes classifiers and K-Means is **Ganitha**, which can be found at `https://github.com/tresata/ganitha`. This library, among others, simplifies the interaction with Mahout vectors. Random access sparse vectors or dense vectors can be created with the following:

```
val randAccessSparseVector = RichVector(6,List((1,1.0),(3,2.0)))
val denseVector = = RichVector(Array(1.0,2.0,3.0))
```

Summary

Scalding provides a number of ways to implement and execute machine learning algorithms. As presented, we can manipulate pipes, use the Matrix API or algebird, and interoperate with existing libraries such as Mahout.

The majority of ML jobs originate as Big Data ETL jobs that reduce to a smaller data space. The final result usually needs some form of post-processing, and it is then stored in an external source. Scalding provides great interoperability with external systems, and it is thus one of the most suitable technologies to solve such problems.

Index

[PACKT] open source
PUBLISHING community experience distilled

Thank you for buying
Programming MapReduce with Scalding

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike.
For more information, please visit our website: `www.packtpub.com`.

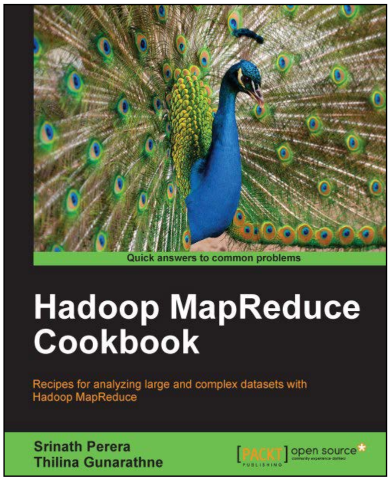
About Packt Open Source

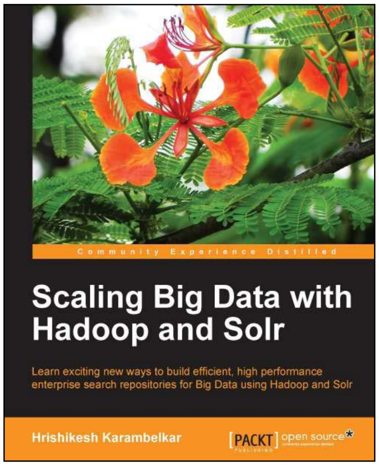
In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

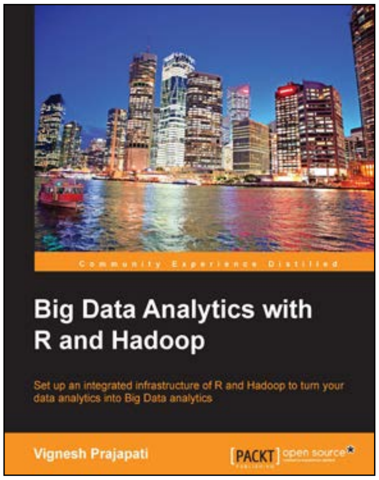
Writing for Packt

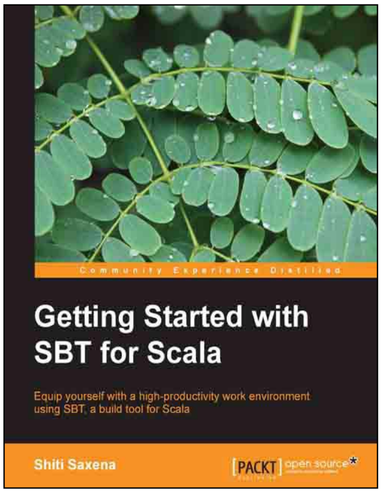
We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to `author@packtpub.com`. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.


Quick answers to common problems
Hadoop MapReduce Cookbook
Recipes for analyzing large and complex datasets with Hadoop MapReduce
Srinath Perera
Thilina Gunarathne
[PACKT] open source


Community Experience Distilled
Scaling Big Data with Hadoop and Solr
Learn exciting new ways to build efficient, high performance enterprise search repositories for Big Data using Hadoop and Solr
Hrishikesh Karambelkar
[PACKT] open source


Community Experience Distilled
Big Data Analytics with R and Hadoop
Set up an integrated infrastructure of R and Hadoop to turn your data analytics into Big Data analytics
Vignesh Prajapati
[PACKT] open source


Community Experience Distilled
Getting Started with SBT for Scala
Equip yourself with a high-productivity work environment using SBT, a build tool for Scala
Shiti Saxena
[PACKT] open source