O'REILLY®

Data Algorithms

RECIPES FOR SCALING UP WITH HADOOP AND SPARK

Mahmoud Parsian

Data Algorithms

Mahmoud Parsian

Data Algorithms

by Mahmoud Parsian

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

- Editors: Ann Spencer and Marie Beaugureau
- Production Editor: Matthew Hacker
- Copyeditor: Rachel Monaghan
- Proofreader: Rachel Head
- Indexer: Judith McConville
- Interior Designer: David Futato
- Cover Designer: Ellie Volckhausen
- Illustrator: Rebecca Demarest
- July 2015: First Edition

Revision History for the First Edition

- 2015-07-10: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491906187* for release details.

978-1-491-90618-7

[LSI]

Dedication

This book is dedicated to my dear family:

wife, Behnaz,

daughter, Maral,

son, Yaseen

Foreword

Unlocking the power of the genome is a powerful notion — one that intimates knowledge, understanding, and the ability of science and technology to be transformative. But transformation requires alignment and synergy, and synergy almost always requires deep collaboration. From scientists to software engineers, and from academia into the clinic, we will need to work together to pave the way for our genetically empowered future.

The creation of data algorithms that analyze the information generated from large-scale genetic sequencing studies is key. Genetic variations are diverse; they can be complex and novel, compounded by a need to connect them to an individual's physical presentation in a meaningful way for clinical insights to be gained and applied. Accelerating our ability to do this at scale, across populations of individuals, is critical. The methods in this book serve as a compass for the road ahead.

MapReduce, Hadoop, and Spark are key technologies that will help us scale the use of genetic sequencing, enabling us to store, process, and analyze the "big data" of genomics. Mahmoud's book covers these topics in a simple and practical manner. *Data Algorithms* illuminates the way for data scientists, software engineers, and ultimately clinicians to unlock the power of the genome, helping to move human health into an era of precision, personalization, and transformation.

— Jay Flatley

CEO, Illumina Inc.

Preface

With the development of massive search engines (such as Google and Yahoo!), genomic analysis (in DNA sequencing, RNA sequencing, and biomarker analysis), and social networks (such as Facebook and Twitter), the volumes of data being generated and processed have crossed the petabytes threshold. To satisfy these massive computational requirements, we need efficient, scalable, and parallel algorithms. One framework to tackle these problems is the MapReduce paradigm.

MapReduce is a software framework for processing large (giga-, tera-, or petabytes) data sets in a parallel and distributed fashion, and an execution framework for large-scale data processing on clusters of commodity servers. There are many ways to implement MapReduce, but in this book our primary focus will be Apache Spark and MapReduce/Hadoop. You will learn how to implement MapReduce in Spark and Hadoop through simple and concrete examples.

This book provides essential distributed algorithms (implemented in MapReduce, Hadoop, and Spark) in the following areas, and the chapters are organized accordingly:

- Basic design patterns
- Data mining and machine learning
- Bioinformatics, genomics, and statistics
- Optimization techniques

What Is MapReduce?

MapReduce is a programming paradigm that allows for massive scalability across hundreds or thousands of servers in a cluster environment. The term *MapReduce* originated from functional programming and was introduced by Google in a paper called “MapReduce: Simplified Data Processing on Large

Clusters.” Google’s MapReduce[8] implementation is a proprietary solution and has not yet been released to the public.

A simple view of the MapReduce process is illustrated in Figure P-1. Simply put, MapReduce is about scalability. Using the MapReduce paradigm, you focus on writing two functions:

`map()`
: Filters and aggregates data

`reduce()`
: Reduces, groups, and summarizes by keys generated by `map()`

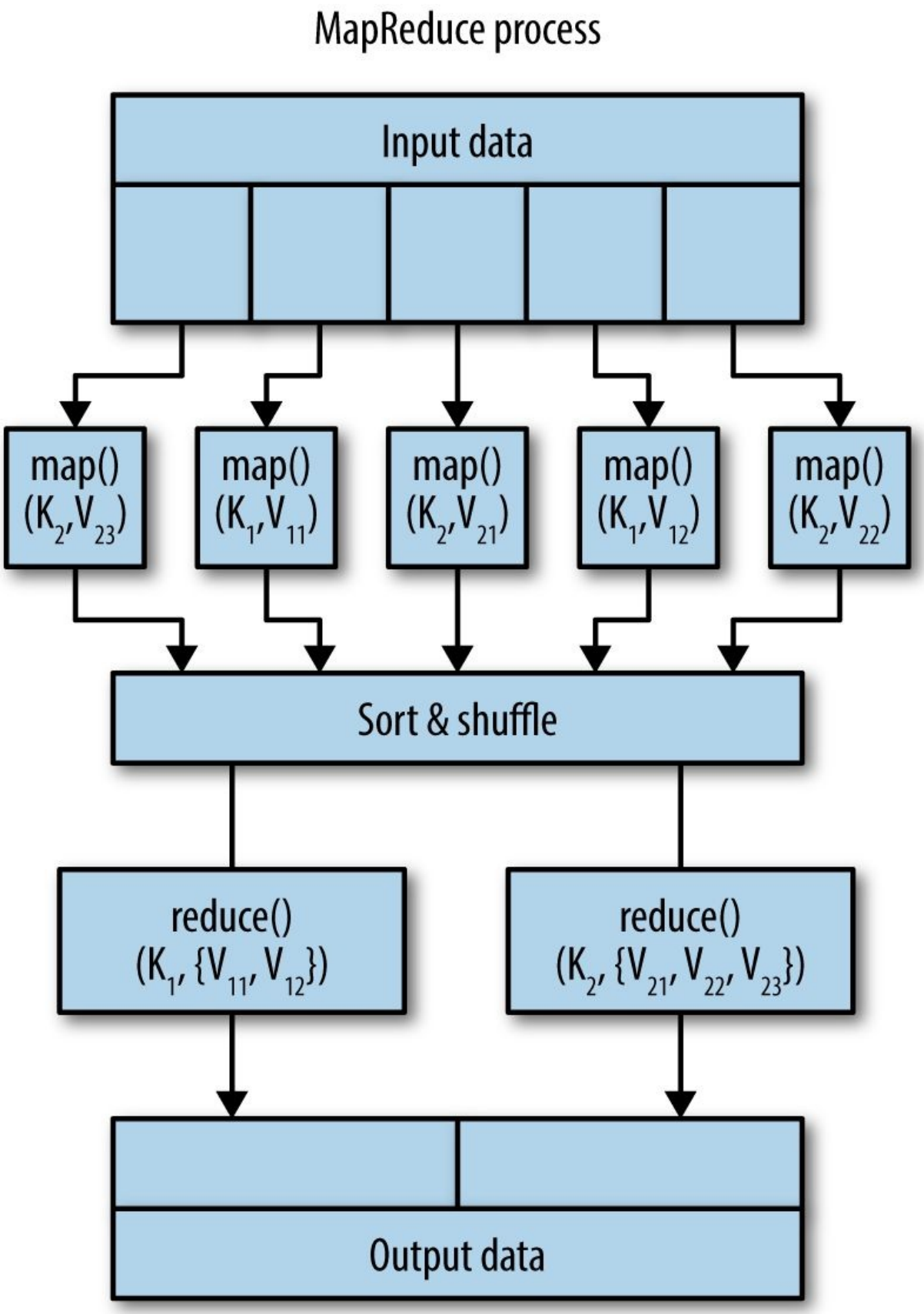
MapReduce process
Input data
map()
(K2,V23)
map()
(K1,V11)
map()
(K2,V21)
map()
(K1,V12)
map()
(K2,V22)
Sort & shuffle
reduce()
(K1, {V11, V12})
reduce()
(K2, {V21, V22, V23})
Output data

Figure P-1. The simple view of the MapReduce process

These two functions can be defined as follows:

`map()` function

The master node takes the input, partitions it into smaller data chunks, and distributes them to worker (slave) nodes. The worker nodes apply the same transformation function to each data chunk, then pass the results back to the master node. In MapReduce, the programmer defines a mapper with the following signature:

```
map(): (Key1, Value1) → [(Key2, Value2)]
```

`reduce()` function

The master node shuffles and clusters the received results based on unique key-value pairs; then, through another redistribution to the workers/slaves, these values are combined via another type of transformation function. In MapReduce, the programmer defines a reducer with the following signature:

```
reduce(): (Key2, [Value2]) → [(Key3, Value3)]
```

> **NOTE**
>
> In informal presentations of the `map()` and `reduce()` functions throughout this book, I've used square brackets, `[]`, to denote a list.

In Figure P-1, input data is partitioned into small chunks (here we have five input partitions), and each chunk is sent to a mapper. Each mapper may generate any number of key-value pairs. The mappers' output is illustrated by Table P-1.

Table P-1. Mappers' output

Key	Value
K_1	V_{11}
K_2	V_{21}
K_1	V_{12}
K_2	V_{22}
K_2	V_{23}

In this example, all mappers generate only two unique keys: $\{K_1, K_2\}$. When all mappers are completed, the keys are sorted, shuffled, grouped, and sent to reducers. Finally, the reducers generate the desired outputs. For this example, we have two reducers identified by $\{K_1, K_2\}$ keys (illustrated by Table P-2).

Table P-2. Reducers' input

Key	Value
K_1	$\{V_{11}, V_{12}\}$
K_2	$\{V_{21}, V_{22}, V_{23}\}$

Once all mappers are completed, the reducers start their execution process. Each reducer may create as an output any number — zero or more — of new key-value pairs.

When writing your `map()` and `reduce()` functions, you need to make sure that your solution is scalable. For example, if you are utilizing any data structure (such as `List`, `Array`, or `HashMap`) that will not easily fit into the memory of a commodity server, then your solution is not scalable. Note that your `map()` and `reduce()` functions will be executing in basic commodity servers, which might have 32 GB or 64 GB of RAM at most (note that this is just an example; today's servers have 256 GB or 512 GB of RAM, and in the next few years basic servers might have even 1 TB of RAM). Scalability is therefore the heart of MapReduce. If your MapReduce solution does not scale well, you should not call it a MapReduce solution. Here, when we talk about scalability, we mean scaling out (the term *scale out* means to add more commodity nodes to a

system). MapReduce is mainly about scaling out (as opposed to *scaling up*, which means adding resources such as memory and CPUs to a single node). For example, if DNA sequencing takes 60 hours with 3 servers, then scaling out to 50 similar servers might accomplish the same DNA sequencing in less than 2 hours.

The core concept behind MapReduce is mapping your input data set into a collection of key-value pairs, and then reducing over all pairs with the same key. Even though the overall concept is simple, it is actually quite expressive and powerful when you consider that:

- Almost all data can be mapped into key-value pairs.
- Your keys and values may be of any type: `Strings`, `Integers`, FASTQ (for DNA sequencing), user-defined custom types, and, of course, key-value pairs themselves.

How does MapReduce scale over a set of servers? The key to how MapReduce works is to take input as, conceptually, a list of records (each single record can be one or more lines of data). Then the input records are split and passed to the many servers in the cluster to be consumed by the `map()` function. The result of the `map()` computation is a list of key-value pairs. Then the `reduce()` function takes each set of values that have the same key and combines them into a single value (or set of values). In other words, the `map()` function takes a set of data chunks and produces key-value pairs, and `reduce()` merges the output of the data generated by `map()`, so that instead of a set of key-value pairs, you get your desired result.

One of the major benefits of MapReduce is its “shared-nothing” data-processing platform. This means that all mappers can work independently, and when mappers complete their tasks, reducers start to work independently (no data or critical region is shared among mappers or reducers; having a critical region will slow distributed computing). This shared-nothing paradigm enables us to write `map()` and `reduce()` functions easily and improves parallelism effectively and effortlessly.

Simple Explanation of MapReduce

What is a very simple explanation of MapReduce? Let's say that we want to count the number of books in a library that has 1,000 shelves and report the final result to the librarian. Here are two possible MapReduce solutions:

- Solution #1 (using `map()` and `reduce()`):

  `map()`: Hire 1,000 workers; each worker counts one shelf.

  `reduce()`: All workers get together and add up their individual counts (by reporting the results to the librarian).

- Solution #2 (using `map()`, `combine()`, and `reduce()`):

  `map()`: Hire 1,110 workers (1,000 workers, 100 managers, 10 supervisors — each supervisor manages 10 managers, and each manager manages 10 workers); each worker counts one shelf, and reports its count to its manager.

  `combine()`: Every 10 managers add up their individual counts and report the total to a supervisor.

  `reduce()`: All supervisors get together and add up their individual counts (by reporting the results to the librarian).

When to Use MapReduce

Is MapReduce good for everything? The simple answer is no. When we have big data, if we can partition it and each partition can be processed independently, then we can start to think about MapReduce algorithms. For example, graph algorithms do not work very well with MapReduce due to their iterative approach. But if you are grouping or aggregating a lot of data, the MapReduce paradigm works pretty well. To process graphs using MapReduce, you should take a look at the Apache Giraph and Apache Spark GraphX projects.

Here are other scenarios where MapReduce should not be used:

- If the computation of a value depends on previously computed values. One good example is the Fibonacci series, where each value is a summation of the previous two values:

  ```
  F(k + 2) = F(k + 1) + F(k)
  ```

- If the data set is small enough to be computed on a single machine. It is better

to do this as a single `reduce(map(data))` operation rather than going through the entire MapReduce process.

- If synchronization is required to access shared data.
- If all of your input data fits in memory.
- If one operation depends on other operations.
- If basic computations are processor-intensive.

However, there are many cases where MapReduce is appropriate, such as:

- When you have to handle lots of input data (e.g., aggregate or compute statistics over large amounts of data).
- When you need to take advantage of parallel and distributed computing, data storage, and data locality.
- When you can do many tasks independently without synchronization.
- When you can take advantage of sorting and shuffling.
- When you need fault tolerance and you cannot afford job failures.

What MapReduce Isn’t

MapReduce is a groundbreaking technology for distributed computing, but there are a lot of myths about it, some of which are debunked here:

- MapReduce is not a programming language, but rather a framework to develop distributed applications using Java, Scala, and other programming languages.
- MapReduce’s distributed filesystem is not a replacement for a relational database management system (such as MySQL or Oracle). Typically, the input to MapReduce is plain-text files (a mapper input record can be one or many lines).
- The MapReduce framework is designed mainly for batch processing, so we should not expect to get the results in under two seconds; however, with

proper use of clusters you may achieve near-real-time response.

- MapReduce is not a solution for all software problems.

Why Use MapReduce?

As we've discussed, MapReduce works on the premise of "scaling out" by adding more commodity servers. This is in contrast to "scaling up," by adding more resources, such as memory and CPUs, to a single node in a system); this can be very costly, and at some point you won't be able to add more resources due to cost and software or hardware limits. Many times, there are promising main memory–based algorithms available for solving data problems, but they lack scalability because the main memory is a bottleneck. For example, in DNA sequencing analysis, you might need over 512 GB of RAM, which is very costly and not scalable.

If you need to increase your computational power, you'll need to distribute it across more than one machine. For example, to do DNA sequencing of 500 GB of sample data, it would take one server over four days to complete just the alignment phase; using 60 servers with MapReduce can cut this time to less than two hours. To process large volumes of data, you must be able to split up the data into chunks for processing, which are then recombined later. MapReduce/Hadoop and Spark/Hadoop enable you to increase your computational power by writing just two functions: `map()` and `reduce()`. So it's clear that data analytics has a powerful new tool with the MapReduce paradigm, which has recently surged in popularity thanks to open source solutions such as Hadoop.

In a nutshell, MapReduce provides the following benefits:

- Programming model + infrastructure
- The ability to write programs that run on hundreds/thousands of machines
- Automatic parallelization and distribution
- Fault tolerance (if a server dies, the job will be completed by other servers)
- Program/job scheduling, status checking, and monitoring

Hadoop and Spark

Hadoop is the de facto standard for implementation of MapReduce applications. It is composed of one or more master nodes and any number of slave nodes. Hadoop simplifies distributed applications by saying that "the data center is the computer," and by providing `map()` and `reduce()` functions (defined by the programmer) that allow application developers or programmers to utilize those data centers. Hadoop implements the MapReduce paradigm efficiently and is quite simple to learn; it is a powerful tool for processing large amounts of data in the range of terabytes and petabytes.

In this book, most of the MapReduce algorithms are presented in a cookbook format (compiled, complete, and working solutions) and implemented in Java/MapReduce/Hadoop and/or Java/Spark/Hadoop. Both the Hadoop and Spark frameworks are open source and enable us to perform a huge volume of computations and data processing in distributed environments.

These frameworks enable scaling by providing "scale-out" methodology. They can be set up to run intensive computations in the MapReduce paradigm on thousands of servers. Spark's API has a higher-level abstraction than Hadoop's API; for this reason, we are able to express Spark solutions in a single Java driver class.

Hadoop and Spark are two different distributed software frameworks. Hadoop is a MapReduce framework on which you may run jobs supporting the `map()`, `combine()`, and `reduce()` functions. The MapReduce paradigm works well at one-pass computation (first `map()`, then `reduce()`), but is inefficient for multipass algorithms. Spark is not a MapReduce framework, but can be easily used to support a MapReduce framework's functionality; it has the proper API to handle `map()` and `reduce()` functionality. Spark is not tied to a map phase and then a reduce phase. A Spark job can be an arbitrary *DAG* (directed acyclic graph) of map and/or reduce/shuffle phases. Spark programs may run with or without Hadoop, and Spark may use *HDFS* (Hadoop Distributed File System) or other persistent storage for input/output. In a nutshell, for a given Spark program or job, the Spark engine creates a DAG of task stages to be performed on the cluster, while Hadoop/MapReduce, on the other hand, creates a DAG with two predefined stages, map and reduce. Note that DAGs created by Spark can contain any number of stages. This allows most Spark jobs to complete faster

than they would in Hadoop/MapReduce, with simple jobs completing after just one stage and more complex tasks completing in a single run of many stages, rather than having to be split into multiple jobs. As mentioned, Spark's API is a higher-level abstraction than MapReduce/Hadoop. For example, a few lines of code in Spark might be equivalent to 30–40 lines of code in MapReduce/Hadoop.

Even though frameworks such as Hadoop and Spark are built on a "shared-nothing" paradigm, they do support sharing immutable data structures among all cluster nodes. In Hadoop, you may pass these values to mappers and reducers via Hadoop's `Configuration` object; in Spark, you may share data structures among mappers and reducers by using `Broadcast` objects. In addition to `Broadcast` read-only objects, Spark supports write-only accumulators. Hadoop and Spark provide the following benefits for big data processing:

Reliability
: Hadoop and Spark are fault-tolerant (any node can go down without losing the result of the desired computation).

Scalability
: Hadoop and Spark support large clusters of servers.

Distributed processing
: In Spark and Hadoop, input data and processing are distributed (they support big data from the ground up).

Parallelism
: Computations are executed on a cluster of nodes in parallel.

Hadoop is designed mainly for batch processing, while with enough memory/RAM, Spark may be used for near real-time processing. To understand basic usage of Spark RDDs (resilient distributed data sets), see Appendix B.

So what are the core components of MapReduce/Hadoop?

- Input/output data consists of key-value pairs. Typically, keys are integers, longs, and strings, while values can be almost any data type (string, integer, long, sentence, special-format data, etc.).

- Data is partitioned over commodity nodes, filling racks in a data center.

- The software handles failures, restarts, and other interruptions. Known as *fault tolerance*, this is an important feature of Hadoop.

Hadoop and Spark provide more than `map()` and `reduce()` functionality: they provide plug-in model for custom record reading, secondary data sorting, and much more.

A high-level view of the relationship between Spark, YARN, and Hadoop's HDFS is illustrated in Figure P-2.

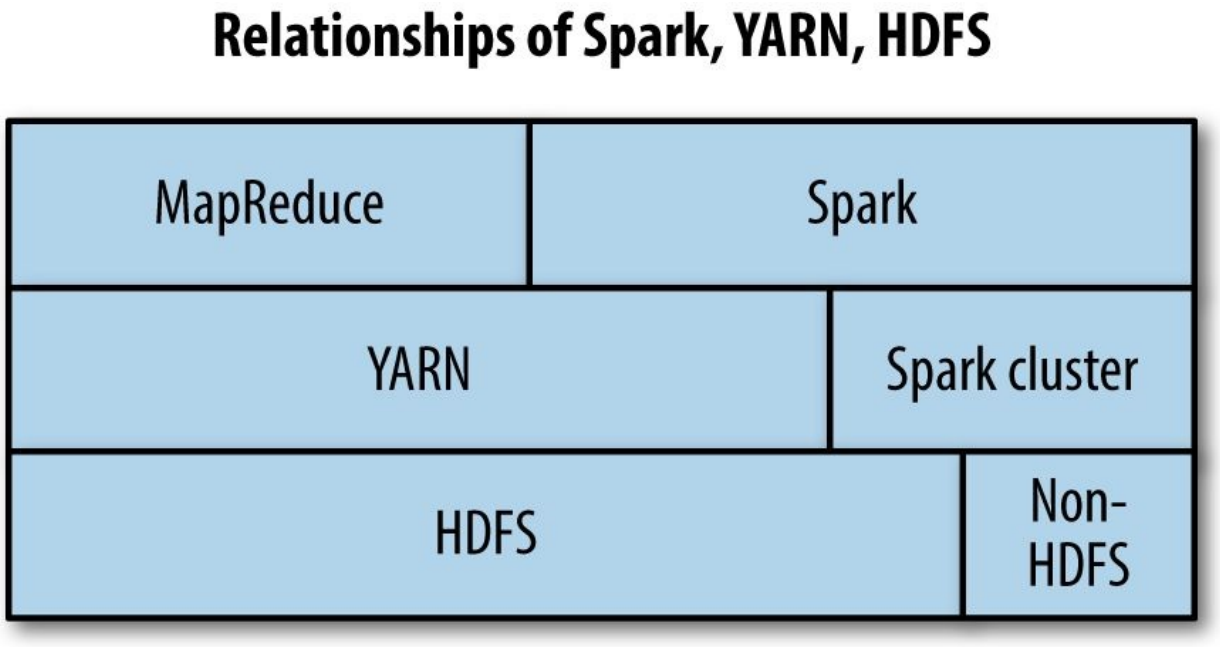

Figure P-2. Relationship between MapReduce, Spark, and HDFS

This relationship shows that there are many ways to run MapReduce and Spark using HDFS (and non-HDFS filesystems). In this book, I will use the following keywords and terminology:

- *MapReduce* refers to the general MapReduce framework paradigm.
- *MapReduce/Hadoop* refers to a specific implementation of the MapReduce framework using Hadoop.
- *Spark* refers to a specific implementation of Spark using HDFS as a persistent storage or a compute engine (note that Spark can run against any data store, but here we focus mostly on Hadoop's):

Spark can run without Hadoop using standalone cluster mode (which may use HDFS, NFS, or another medium as a persistent data store).

Spark can run with Hadoop using Hadoop's YARN or MapReduce framework.

Using this book, you will learn step by step the algorithms and tools you need to build MapReduce applications with Hadoop. MapReduce/Hadoop has become the programming model of choice for processing large data sets (such as log data, genome sequences, statistical applications, and social graphs). MapReduce can be used for any application that does not require tightly coupled parallel processing. Keep in mind that Hadoop is designed for MapReduce batch processing and is not an ideal solution for real-time processing. Do not expect to get your answers from Hadoop in 2 to 5 seconds; the smallest jobs might take 20+ seconds. Spark is a top-level Apache project that is well suited for near real-time processing, and will perform better with more RAM. With Spark, it is very possible to run a job (such as biomarker analysis or Cox regression) that processes 200 million records in 25 to 35 seconds by just using a cluster of 100 nodes. Typically, Hadoop jobs have a latency of 15 to 20 seconds, but this depends on the size and configuration of the Hadoop cluster.

An implementation of MapReduce (such as Hadoop) runs on a large cluster of commodity machines and is highly scalable. For example, a typical MapReduce computation processes many petabytes or terabytes of data on hundreds or thousands of machines. Programmers find MapReduce easy to use because it hides the messy details of parallelization, fault tolerance, data distribution, and load balancing, letting the programmers focus on writing the two key functions, `map()` and `reduce()`.

The following are some of the major applications of MapReduce/Hadoop/Spark:

- Query log processing
- Crawling, indexing, and search
- Analytics, text processing, and sentiment analysis
- Machine learning (such as Markov chains and the Naive Bayes classifier)

- Recommendation systems
- Document clustering and classification
- Bioinformatics (alignment, recalibration, germline ingestion, and DNA/RNA sequencing)
- Genome analysis (biomarker analysis, and regression algorithms such as linear and Cox)

What Is in This Book?

Each chapter of this book presents a problem and solves it through a set of MapReduce algorithms. MapReduce algorithms/solutions are complete recipes (including the MapReduce driver, mapper, combiner, and reducer programs). You can use the code directly in your projects (although sometimes you may need to cut and paste the sections you need). This book does not cover the theory behind the MapReduce framework, but rather offers practical algorithms and examples using MapReduce/Hadoop and Spark to solve tough big data problems. Topics covered include:

- Market Basket Analysis for a large set of transactions
- Data mining algorithms (K-Means, kNN, and Naive Bayes)
- DNA sequencing and RNA sequencing using huge genomic data
- Naive Bayes classification and Markov chains for data and market prediction
- Recommendation algorithms and pairwise document similarity
- Linear regression, Cox regression, and Pearson correlation
- Allelic frequency and mining DNA
- Social network analysis (recommendation systems, counting triangles, sentiment analysis)

You may cut and paste the provided solutions from this book to build your own

MapReduce applications and solutions using Hadoop and Spark. All the solutions have been compiled and tested. This book is ideal for anyone who knows some Java (i.e., can read and write basic Java programs) and wants to write and deploy MapReduce algorithms using Java/Hadoop/Spark. The general topic of MapReduce has been discussed in detail in an excellent book by Jimmy Lin and Chris Dyer[16]; again, the goal of this book is to provide concrete MapReduce algorithms and solutions using Hadoop and Spark. Likewise, this book will not discuss Hadoop itself in detail; Tom White's excellent book[31] does that very well.

This book will not cover how to install Hadoop or Spark; I am going to assume you already have these installed. Also, any Hadoop commands are executed relative to the directory where Hadoop is installed (the `$HADOOP_HOME` environment variable). This book is explicitly about presenting distributed algorithms using MapReduce/Hadoop and Spark. For example, I discuss APIs, cover command-line invocations for running jobs, and provide complete working programs (including the driver, mapper, combiner, and reducer).

What Is the Focus of This Book?

The focus of this book is to embrace the MapReduce paradigm and provide concrete problems that can be solved using MapReduce/Hadoop algorithms. For each problem presented, we will detail the `map()`, `combine()`, and `reduce()` functions and provide a complete solution, which has:

- A client, which calls the driver with proper input and output parameters.
- A driver, which identifies `map()` and `reduce()` functions, and identifies input and output.
- A mapper class, which implements the `map()` function.
- A combiner class (when possible), which implements the `combine()` function. We will discuss when it is possible to use a combiner.
- A reducer class, which implements the `reduce()` function.

One goal of this book is to provide step-by-step instructions for using Spark and

Hadoop as a solution for MapReduce algorithms. Another is to show how an output of one MapReduce job can be used as an input to another (this is called *chaining* or *pipelining* MapReduce jobs).

Who Is This Book For?

This book is for software engineers, software architects, data scientists, and application developers who know the basics of Java and want to develop MapReduce algorithms (in data mining, machine learning, bioinformatics, genomics, and statistics) and solutions using Hadoop and Spark. As I've noted, I assume you know the basics of the Java programming language (e.g., writing a class, defining a new class from an existing class, and using basic control structures such as the `while` loop and `if-then-else`).

More specifically, this book is targeted to the following readers:

- Data science engineers and professionals who want to do analytics (classification, regression algorithms) on big data. The book shows the basic steps, in the format of a cookbook, to apply classification and regression algorithms using big data. The book details the `map()` and `reduce()` functions by demonstrating how they are applied to real data, and shows where to apply basic design patterns to solve MapReduce problems. These MapReduce algorithms can be easily adapted across professions with some minor changes (for example, by changing the input format). All solutions have been implemented in Apache Hadoop/Spark so that these examples can be adapted in real-world situations.

- Software engineers and software architects who want to design machine learning algorithms such as Naive Bayes and Markov chain algorithms. The book shows how to build the model and then apply it to a new data set using MapReduce design patterns.

- Software engineers and software architects who want to use data mining algorithms (such as K-Means clustering and k-Nearest Neighbors) with MapReduce. Detailed examples are given to guide professionals in implementing similar algorithms.

- Data science engineers who want to apply MapReduce algorithms to clinical and biological data (such as DNA sequencing and RNA sequencing). This book clearly explains practical algorithms suitable for bioinformaticians and clinicians. It presents the most relevant regression/analytical algorithms used for different biological data types. The majority of these algorithms have been

deployed in real-world production systems.

- Software architects who want to apply the most important optimizations in a MapReduce/distributed environment.

This book assumes you have a basic understanding of Java and Hadoop's HDFS. If you need to become familiar with Hadoop and Spark, the following books will offer you the background information you will need:

- *Hadoop: The Definitive Guide* by Tom White (O'Reilly)
- *Hadoop in Action* by Chuck Lam (Manning Publications)
- *Hadoop in Practice* by Alex Holmes (Manning Publications)
- *Learning Spark* by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia (O'Reilly)

Online Resources

Two websites accompany this book:

https://github.com/mahmoudparsian/data-algorithms-book/

At this GitHub site, you will find links to the source code (organized by chapter), shell scripts (for running MapReduce/Hadoop and Spark programs), sample input files for testing, and some extra content that isn't in the book, including a couple of bonus chapters.

http://mapreduce4hackers.com

At this site, you will find links to extra source files (not mentioned in the book) plus some additional content that is not in the book. Expect more coverage of MapReduce/Hadoop/Spark topics in the future.

What Software Is Used in This Book?

When developing solutions and examples for this book, I used the software and programming environments listed in Table P-3.

Table P-3. Software/programming

environments used in this book

Software	Version
Java programming language (JDK7)	1.7.0_67
Operating system: Linux CentOS	6.3
Operating system: Mac OS X	10.9
Apache Hadoop	2.5.0, 2.6.0
Apache Spark	1.1.0, 1.3.0, 1.4.0
Eclipse IDE	Luna

All programs in this book were tested with Java/JDK7, Hadoop 2.5.0, and Spark (1.1.0, 1.3.0, 1.4.0). Examples are given in mixed operating system environments (Linux and OS X). For all examples and solutions, I engaged basic text editors (such as vi, vim, and TextWrangler) and compiled them using the Java command-line compiler (`javac`).

In this book, shell scripts (such as bash scripts) are used to run sample MapReduce/Hadoop and Spark programs. Lines that begin with a `$` or `#` character indicate that the commands must be entered at a terminal prompt (such as bash).

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

NOTE

This element signifies a general note.

Using Code Examples

As mentioned previously, supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/mahmoudparsian/data-algorithms-book/* and *http://www.mapreduce4hackers.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Data Algorithms* by Mahmoud Parsian (O'Reilly). Copyright 2015 Mahmoud Parsian, 978-1-491-90618-7."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Books Online

Safari Books Online is an on-demand digital library that delivers expert content in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of plans and pricing for enterprise, government, education, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons,

Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds more. For more information about Safari Books Online, please visit us online.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (in the United States or Canada)
- 707-829-0515 (international or local)
- 707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *http://bit.ly/data_algorithms*.

To comment or ask technical questions about this book, send email to *bookquestions@oreilly.com*.

For more information about our books, courses, conferences, and news, see our website at *http://www.oreilly.com*.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://www.youtube.com/oreillymedia*

Acknowledgments

To each reader: a big thank you for reading my book. I hope that this book is useful and serves you well.

Thank you to my editor at O’Reilly, Ann Spencer, for believing in my book project, supporting me in reorganizing the chapters, and suggesting a new title (originally, I proposed a title of *MapReduce for Hackers*). Also, I want to thank Mike Loukides (VP of Content Strategy for O’Reilly Media) for believing in and supporting my book project.

Thank you so much to my editor, Marie Beaugureau, data and development editor at O’Reilly, who has worked with me patiently for a long time and supported me during every phase of this project. Marie’s comments and suggestions have been very useful and helpful.

A big thank you to Rachel Monaghan, copyeditor, for her superb knowledge of book editing and her valuable comments and suggestions. This book is more readable because of her. Also, I want to say a big thank you to Matthew Hacker, production editor, who has done a great job in getting this book through production. Thanks to Rebecca Demarest (O’Reilly’s illustrator) and Dan Fauxsmith (Director of Publishing Services for O’Reilly) for polishing the artwork. Also, I want to say thank you to Rachel Head (as proofreader), Judith McConville (as indexer), David Futato (as interior designer), and Ellie Volckhausen (as cover designer).

Thanks to my technical reviewers, Cody Koeninger, Kun Lu, Neera Vats, Dr. Phanendra Babu, Willy Bruns, and Mohan Reddy. Your comments were useful, and I have incorporated your suggestions as much as possible. Special thanks to Cody for providing detailed feedback.

A big thank you to Jay Flatley (CEO of Illumina), who has provided a tremendous opportunity and environment in which to unlock the power of the genome. Thank you to my dear friends Saeid Akhtari (CEO, NextBio) and Dr. Satnam Alag (VP of Engineering at Illumina) for believing in me and supporting me for the past five years.

Thanks to my longtime dear friend, Dr. Ramachandran Krishnaswamy (my Ph.D. advisor), for his excellent guidance and for providing me with the environment to work on computer science.

Thanks to my dear parents (mother Monireh Azemoun and father Bagher Parsian) for making education their number one priority. They have supported me tremendously. Thanks to my brother, Dr. Ahmad Parsian, for helping me to understand mathematics. Thanks to my sister, Nayer Azam Parsian, for helping

me to understand compassion.

Last, but not least, thanks to my dear family — Behnaz, Maral, and Yaseen — whose encouragement and support throughout the writing process means more than I can say.

Comments and Questions for This Book

I am always interested in your feedback and comments regarding the problems and solutions described in this book. Please email comments and questions for this book to mahmoud.parsian@yahoo.com. You can also find me at *http://www.mapreduce4hackers.com*.

— Mahmoud Parsian

Sunnyvale, California

March 26, 2015

Chapter 1. Secondary Sort: Introduction

A *secondary sort problem* relates to sorting values associated with a key in the reduce phase. Sometimes, it is called *value-to-key conversion*. The secondary sorting technique will enable us to sort the values (in ascending or descending order) passed to each reducer. I will provide concrete examples of how to achieve secondary sorting in ascending or descending order.

The goal of this chapter is to implement the Secondary Sort design pattern in MapReduce/Hadoop and Spark. In software design and programming, a *design pattern* is a reusable algorithm that is used to solve a commonly occurring problem. Typically, a design pattern is not presented in a specific programming language but instead can be implemented by many programming languages.

The MapReduce framework automatically sorts the keys generated by mappers. This means that, before starting reducers, all intermediate key-value pairs generated by mappers must be sorted by key (and not by value). Values passed to each reducer are not sorted at all; they can be in any order. What if you also want to sort a reducer's values? MapReduce/Hadoop and Spark do not sort values for a reducer. So, for those applications (such as time series data) in which you want to sort your reducer data, the Secondary Sort design pattern enables you to do so.

First we'll focus on the MapReduce/Hadoop solution. Let's look at the MapReduce paradigm and then unpack the concept of the secondary sort:

- $\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow \texttt{list}(\texttt{key}_2, \texttt{value}_2)$
- $\texttt{reduce}(\texttt{key}_2, \texttt{list}(\texttt{value}_2)) \rightarrow \texttt{list}(\texttt{key}_3, \texttt{value}_3)$

First, the `map()` function receives a key-value pair input, ($\texttt{key}_1$, $\texttt{value}_1$). Then it outputs any number of key-value pairs, ($\texttt{key}_2$, $\texttt{value}_2$). Next, the `reduce()` function receives as input another key-value pair, ($\texttt{key}_2$, $\texttt{list}(\texttt{value}_2)$), and outputs any number of ($\texttt{key}_3$, $\texttt{value}_3$) pairs.

Now consider the following key-value pair, ($\texttt{key}_2$, $\texttt{list}(\texttt{value}_2)$), as an input for

a reducer:

- $\texttt{list}(\texttt{value}_2) = (V_1, V_2, ..., V_n)$

where there is no ordering between reducer values $(V_1, V_2, ..., V_n)$.

The goal of the Secondary Sort pattern is to give *some ordering* to the values received by a reducer. So, once we apply the pattern to our MapReduce paradigm, then we will have:

- $\texttt{SORT}(V_1, V_2, ..., V_n) = (S_1, S_2, ..., S_n)$

- $\texttt{list}(\texttt{value}_2) = (S_1, S_2, ..., S_n)$

where:

- $S_1 < S_2 < ... < S_n$ (ascending order), or

- $S_1 > S_2 > ... > S_n$ (descending order)

Here is an example of a secondary sorting problem: consider the temperature data from a scientific experiment. A dump of the temperature data might look something like the following (columns are `year`, `month`, `day`, and daily `temperature`, respectively):

```
2012, 01, 01, 5
2012, 01, 02, 45
2012, 01, 03, 35
2012, 01, 04, 10
...
2001, 11, 01, 46
2001, 11, 02, 47
2001, 11, 03, 48
2001, 11, 04, 40
...
2005, 08, 20, 50
2005, 08, 21, 52
2005, 08, 22, 38
2005, 08, 23, 70
```

Suppose we want to output the temperature for every `year-month` with the values sorted in ascending order. Essentially, we want the reducer values iterator to be sorted. Therefore, we want to generate something like this output (the first column is `year-month` and the second column is the sorted temperatures):

```
2012-01:  5, 10, 35, 45, ...
2001-11: 40, 46, 47, 48, ...
2005-08: 38, 50, 52, 70, ...
```

Solutions to the Secondary Sort Problem

There are at least two possible approaches for sorting the reducer values. These solutions may be applied to both the MapReduce/Hadoop and Spark frameworks:

- The first approach involves having the reducer read and buffer all of the values for a given *key* (in an array data structure, for example), then doing an in-reducer sort on the values. This approach will not scale: since the reducer will be receiving all values for a given *key*, this approach might cause the reducer to run out of memory (`java.lang.OutOfMemoryError`). On the other hand, this approach can work well if the number of values is small enough that it will not cause an out-of-memory error.

- The second approach involves using the MapReduce framework for sorting the reducer values (this does not require in-reducer sorting of values passed to the reducer). This approach consists of "creating a composite key by adding a part of, or the entire value to, the natural key to achieve your sorting objectives." For the details on this approach, see Java Code Geeks. This option is scalable and will not generate out-of-memory errors. Here, we basically offload the sorting to the MapReduce framework (sorting is a paramount feature of the MapReduce/Hadoop framework).
  This is a summary of the second approach:

  1. Use the *Value-to-Key Conversion* design pattern: form a composite intermediate key, (K, V_1), where V_1 is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V_1) into a reducer key, simply create a composite key (for details, see the `DateTemperaturePair` class). In our example, V_1 is the `temperature` data.

  2. Let the MapReduce execution framework do the sorting (rather than sorting in memory, let the framework sort by using the cluster nodes).

  3. Preserve state across multiple key-value pairs to handle processing; you can achieve this by having proper mapper output partitioners (for example, we partition the mapper's output by the natural key).

Implementation Details

To implement the secondary sort feature, we need additional plug-in Java classes. We have to tell the MapReduce/Hadoop framework:

- How to sort reducer keys
- How to partition keys passed to reducers (custom partitioner)
- How to group data that has arrived at each reducer

Sort order of intermediate keys

To accomplish secondary sorting, we need to take control of the sort order of intermediate keys and the control order in which reducers process keys. First, we inject a value (`temperature` data) into the composite key, and then we take control of the sort order of intermediate keys. The relationships between the natural key, composite key, and key-value pairs are depicted in Figure 1-1.

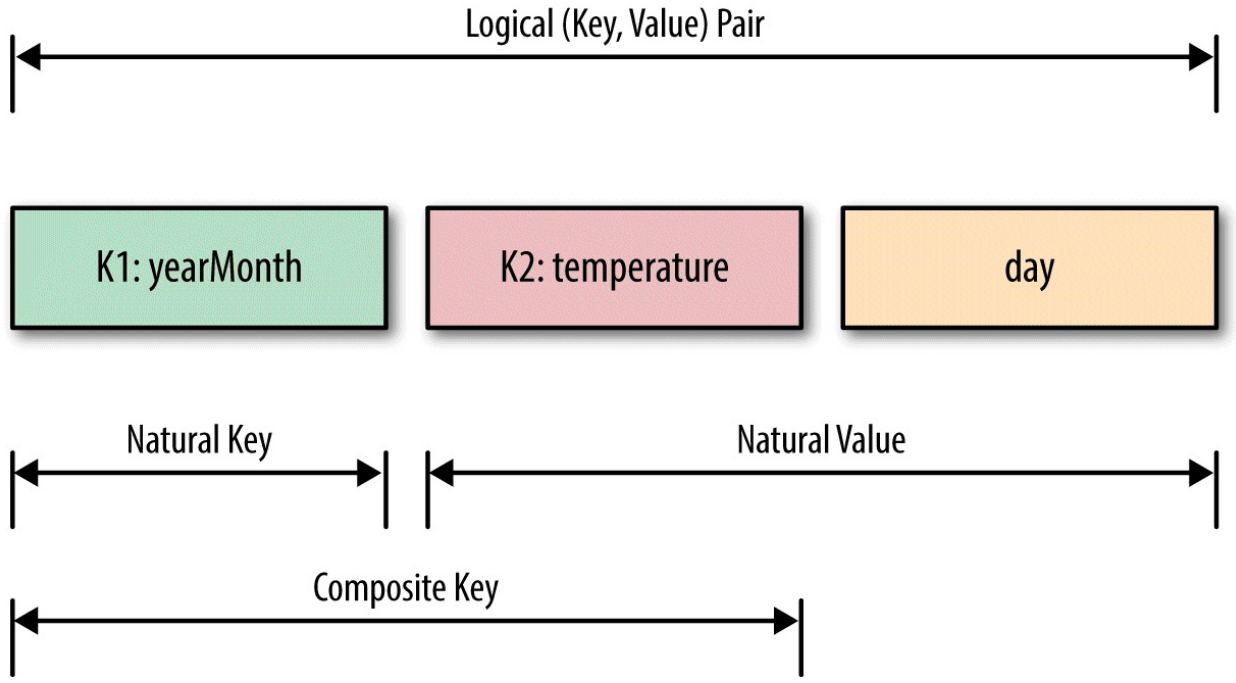

Figure 1-1. Secondary sorting keys

The main question is what value we should add to the natural key to accomplish the secondary sort. The answer is the `temperature` data field (because we want the reducers' values to be sorted by `temperature`). So, we have to indicate how

`DateTemperaturePair` objects should be sorted using the `compareTo()` method. We need to define a proper data structure for holding our key and value, while also providing the sort order of intermediate keys. In Hadoop, for custom data types (such as `DateTemperaturePair`) to be persisted, they have to implement the `Writable` interface; and if we are going to compare custom data types, then they have to implement an additional interface called `WritableComparable` (see Example 1-1).

Example 1-1. DateTemperaturePair class

```
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
...
public class DateTemperaturePair
   implements Writable, WritableComparable<DateTemperaturePair> {

    private Text yearMonth = new Text();                // natural key
    private Text day = new Text();
    private IntWritable temperature = new IntWritable(); // secondary key

    ...

    @Override
    /**
     * This comparator controls the sort order of the keys.
     */
    public int compareTo(DateTemperaturePair pair) {
        int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
        if (compareValue == 0) {
            compareValue = temperature.compareTo(pair.getTemperature());
        }
        //return compareValue;    // sort ascending
        return -1*compareValue;   // sort descending
    }
    ...
}
```

Custom partitioner

In a nutshell, the partitioner decides which mapper's output goes to which reducer based on the mapper's output key. For this, we need two plug-in classes: a custom partitioner to control which reducer processes which keys, and a custom `Comparator` to sort reducer values. The custom partitioner ensures that all data with the same key (the natural key, not including the composite key with the `temperature` value) is sent to the same reducer. The custom `Comparator` does sorting so that the natural key (`year-month`) groups the data once it arrives at the reducer.

Example 1-2. DateTemperaturePartitioner class

```
import org.apache.hadoop.io.Text;
```

```
import org.apache.hadoop.mapreduce.Partitioner;

public class DateTemperaturePartitioner
   extends Partitioner<DateTemperaturePair, Text> {

    @Override
    public int getPartition(DateTemperaturePair pair,
                            Text text,
                            int numberOfPartitions) {
        // make sure that partitions are non-negative
        return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
     }
}
```

Hadoop provides a plug-in architecture for injecting the custom partitioner code into the framework. This is how we do so inside the driver class (which submits the MapReduce job to Hadoop):

```
import org.apache.hadoop.mapreduce.Job;
...
Job job = ...;
...
job.setPartitionerClass(TemperaturePartitioner.class);
```

Grouping comparator

In Example 1-3, we define the comparator (`DateTemperatureGroupingComparator` class) that controls which keys are grouped together for a single call to the `Reducer.reduce()` function.

Example 1-3. DateTemperatureGroupingComparator class

```
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.io.WritableComparator;

public class DateTemperatureGroupingComparator
   extends WritableComparator {

    public DateTemperatureGroupingComparator() {
        super(DateTemperaturePair.class, true);
    }

    @Override
    /**
     * This comparator controls which keys are grouped
     * together into a single call to the reduce() method
     */
    public int compare(WritableComparable wc1, WritableComparable wc2) {
        DateTemperaturePair pair = (DateTemperaturePair) wc1;
        DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
        return pair.getYearMonth().compareTo(pair2.getYearMonth());
    }
}
```

Hadoop provides a plug-in architecture for injecting the grouping comparator code into the framework. This is how we do so inside the driver class (which

submits the MapReduce job to Hadoop):

```
job.setGroupingComparatorClass(YearMonthGroupingComparator.class);
```

Data Flow Using Plug-in Classes

To help you understand the `map()` and `reduce()` functions and custom plug-in classes, Figure 1-2 illustrates the data flow for a portion of input.

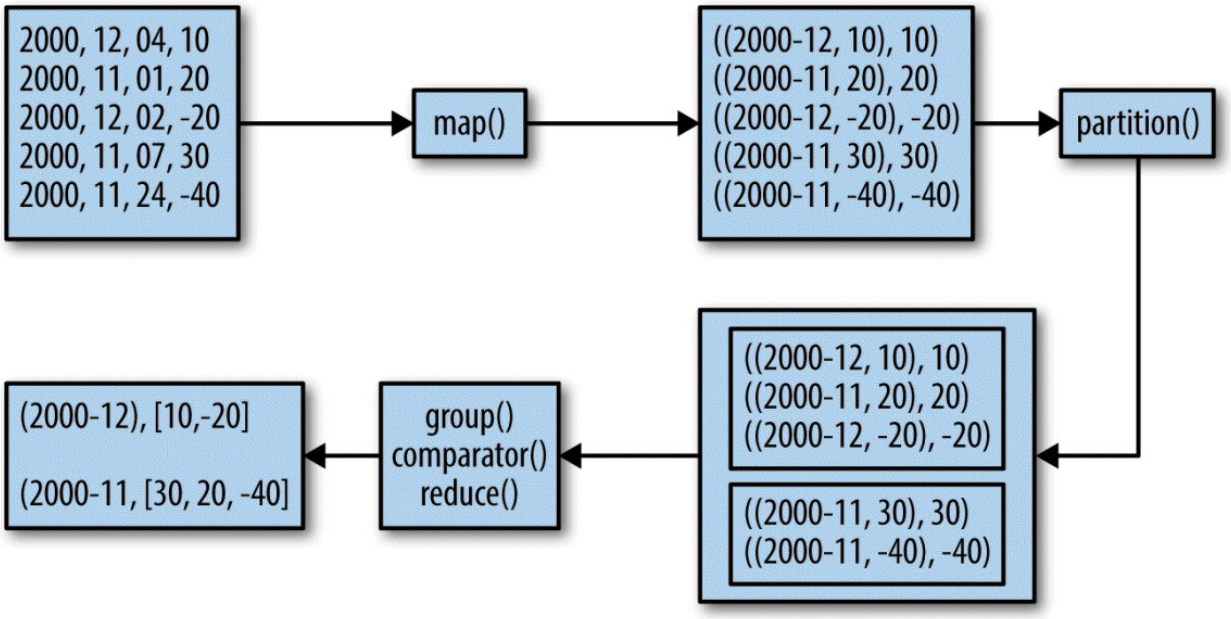

Figure 1-2. Secondary sorting data flow

The mappers create (*K*,*V*) pairs, where *K* is a composite key of `(year,month,temperature)` and *V* is `temperature`. The `(year,month)` part of the composite key is the natural key. The partitioner plug-in class enables us to send all natural keys to the same reducer and the grouping comparator plug-in class enables temperatures to arrive sorted at reducers. The Secondary Sort design pattern uses MapReduce's framework for sorting the reducers' values rather than collecting them all and then sorting them in memory. The Secondary Sort design pattern enables us to "scale out" no matter how many reducer values we want to sort.

MapReduce/Hadoop Solution to Secondary Sort

This section provides a complete MapReduce implementation of the secondary sort problem using the Hadoop framework.

Input

The input will be a set of files, where each record (line) will have the following format:

```
Format:
        <year><,><month><,><day><,><temperature>

Example:
        2012, 01, 01, 35
        2011, 12, 23, -4
```

Expected Output

The expected output will have the following format:

```
Format:
        <year><-><month>: <temperature1><,><temperature2><,> ...
        where temperature1 <= temperature2 <= ...

Example:
        2012-01:  5, 10, 35, 45, ...
        2001-11: 40, 46, 47, 48, ...
        2005-08: 38, 50, 52, 70, ...
```

map() Function

The `map()` function parses and tokenizes the input and then injects the value (`temperature`) into the reducer key, as shown in Example 1-4.

Example 1-4. map() for secondary sorting

```
/**
 * @param key is generated by Hadoop (ignored here)
 * @param value has this format: "YYYY,MM,DD,temperature"
 */
map(key, value) {
   String[] tokens = value.split(",");
   // YYYY = tokens[0]
   // MM = tokens[1]
   // DD = tokens[2]
   // temperature = tokens[3]
   String yearMonth = tokens[0] + tokens[1];
   String day = tokens[2];
   int temperature = Integer.parseInt(tokens[3]);
   // prepare reducer key
   DateTemperaturePair reducerKey = new DateTemperaturePair();
   reducerKey.setYearMonth(yearMonth);
```

```
17    reducerKey.setDay(day);
18    reducerKey.setTemperature(temperature); // inject value into key
19    // send it to reducer
20    emit(reducerKey, temperature);
21 }
```

reduce() Function

The reducer's primary function is to concatenate the values (which are already sorted through the Secondary Sort design pattern) and emit them as output. The `reduce()` function is given in Example 1-5.

Example 1-5. reduce() for secondary sorting

```
1 /**
2  * @param key is a DateTemperaturePair object
3  * @param value is a list of temperatures
4  */
5 reduce(key, value) {
6    StringBuilder sortedTemperatureList = new StringBuilder();
7    for (Integer temperature : value) {
8       sortedTemperatureList.append(temperature);
9       sortedTemperatureList.append(",");
10   }
11   emit(key, sortedTemperatureList);
12 }
```

Hadoop Implementation Classes

The classes shown in Table 1-1 are used to solve the problem.

Table 1-1. Classes used in MapReduce/Hadoop solution

Class name	Class description
`SecondarySortDriver`	The driver class; defines input/output and registers plug-in classes
`SecondarySortMapper`	Defines the `map()` function
`SecondarySortReducer`	Defines the `reduce()` function
`DateTemperatureGroupingComparator`	Defines how keys will be grouped together
`DateTemperaturePair`	Defines paired date and temperature as a Java object
`DateTemperaturePartitioner`	Defines custom partitioner

How is the value injected into the key? The first comparator (the `DateTemperaturePair.compareTo()` method) controls the sort order of the keys, while the second comparator (the

`DateTemperatureGroupingComparator.compare()` method) controls which keys are grouped together into a single call to the `reduce()` method. The combination of these two comparators allows you to set up jobs that act like you've defined an order for the values.

The `SecondarySortDriver` is the driver class, which registers the custom plug-in classes (`DateTemperaturePartitioner` and `DateTemperatureGroupingComparator`) with the MapReduce/Hadoop framework. This driver class is presented in Example 1-6.

Example 1-6. SecondarySortDriver class

```
public class SecondarySortDriver extends Configured implements Tool {
    public int run(String[] args) throws Exception {
        Configuration conf = getConf();
        Job job = new Job(conf);
        job.setJarByClass(SecondarySortDriver.class);
        job.setJobName("SecondarySortDriver");

        Path inputPath = new Path(args[0]);
        Path outputPath = new Path(args[1]);
        FileInputFormat.setInputPaths(job, inputPath);
        FileOutputFormat.setOutputPath(job, outputPath);

        job.setOutputKeyClass(TemperaturePair.class);
        job.setOutputValueClass(NullWritable.class);

        job.setMapperClass(SecondarySortingTemperatureMapper.class);
        job.setReducerClass(SecondarySortingTemperatureReducer.class);
        job.setPartitionerClass(TemperaturePartitioner.class);
        job.setGroupingComparatorClass(YearMonthGroupingComparator.class);

        boolean status = job.waitForCompletion(true);
        theLogger.info("run(): status="+status);
        return status ? 0 : 1;
    }

    /**
    * The main driver for the secondary sort MapReduce program.
    * Invoke this method to submit the MapReduce job.
    * @throws Exception when there are communication
    * problems with the job tracker.
    */
    public static void main(String[] args) throws Exception {
        // Make sure there are exactly 2 parameters
        if (args.length != 2) {
            throw new IllegalArgumentException("Usage: SecondarySortDriver" +
                                               " <input-path> <output-path>");
        }

        //String inputPath = args[0];
        //String outputPath = args[1];
        int returnStatus = ToolRunner.run(new SecondarySortDriver(), args);
        System.exit(returnStatus);
    }

}
```

Sample Run of Hadoop Implementation

Input

```
# cat sample_input.txt
2000,12,04, 10
2000,11,01,20
2000,12,02,-20
2000,11,07,30
2000,11,24,-40
2012,12,21,30
2012,12,22,-20
2012,12,23,60
2012,12,24,70
2012,12,25,10
2013,01,22,80
2013,01,23,90
2013,01,24,70
2013,01,20,-10
```

HDFS input

```
# hadoop fs -mkdir /secondary_sort
# hadoop fs -mkdir /secondary_sort/input
# hadoop fs -mkdir /secondary_sort/output
# hadoop fs -put sample_input.txt /secondary_sort/input/
# hadoop fs -ls /secondary_sort/input/
Found 1 items
-rw-r--r-- 1 ... 128  ...  /secondary_sort/input/sample_input.txt
```

The script

```
# cat run.sh
export JAVA_HOME=/usr/java/jdk7
export BOOK_HOME=/home/mp/data-algorithms-book
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/secondary_sort/input
OUTPUT=/secondary_sort/output
$HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
PROG=org.dataalgorithms.chap01.mapreduce.SecondarySortDriver
$HADOOP_HOME/bin/hadoop jar $APP_JAR $PROG $INPUT $OUTPUT
```

Log of sample run

```
# ./run.sh
...
Deleted hdfs://localhost:9000/secondary_sort/output
13/02/27 19:39:54 INFO input.FileInputFormat: Total input paths to process : 1
...
13/02/27 19:39:54 INFO mapred.JobClient: Running job: job_201302271939_0001
13/02/27 19:39:55 INFO mapred.JobClient:  map 0% reduce 0%
13/02/27 19:40:10 INFO mapred.JobClient:  map 100% reduce 0%
13/02/27 19:40:22 INFO mapred.JobClient:  map 100% reduce 10%
...
```

```
13/02/27 19:41:10 INFO mapred.JobClient:  map 100% reduce 90%
13/02/27 19:41:16 INFO mapred.JobClient:  map 100% reduce 100%
13/02/27 19:41:21 INFO mapred.JobClient: Job complete: job_201302271939_0001
...
13/02/27 19:41:21 INFO mapred.JobClient:   Map-Reduce Framework
...
13/02/27 19:41:21 INFO mapred.JobClient:     Reduce input records=14
13/02/27 19:41:21 INFO mapred.JobClient:     Reduce input groups=4
13/02/27 19:41:21 INFO mapred.JobClient:     Combine output records=0
13/02/27 19:41:21 INFO mapred.JobClient:     Reduce output records=4
13/02/27 19:41:21 INFO mapred.JobClient:     Map output records=14
13/02/27 19:41:21 INFO SecondarySortDriver: run(): status=true
13/02/27 19:41:21 INFO SecondarySortDriver: returnStatus=0
```

Inspecting the output

```
# hadoop fs -cat /secondary_sort/output/p*
2013-01  90,80,70,-10
2000-12  10,-20
2000-11  30,20,-40
2012-12  70,60,30,10,-20
```

How to Sort in Ascending or Descending Order

You can easily control the sorting order of the values (ascending or descending) by using the `DateTemperaturePair.compareTo()` method as follows:

```
1 public int compareTo(DateTemperaturePair pair) {
2     int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
3     if (compareValue == 0) {
4        compareValue = temperature.compareTo(pair.getTemperature());
5     }
6     //return compareValue; // sort ascending
7     return -1*compareValue; // sort descending
8 }
```

Spark Solution to Secondary Sort

To solve a secondary sorting problem in Spark, we have at least two options:

Option #1

> Read and buffer all of the values for a given *key* in an `Array` or `List` data structure and then do an in-reducer sort on the values. This solution works if you have a small set of values (which will fit in memory) per reducer key.

Option #2

> Use the Spark framework for sorting the reducer values (this option does not require in-reducer sorting of values passed to the reducer). This

approach involves “creating a composite key by adding a part of, or the entire value to, the natural key to achieve your sorting objectives.” This option always scales (because you are not limited by the memory of a commodity server).

Time Series as Input

To demonstrate secondary sorting, let’s use time series data:

```
name time value
  x    2    9
  y    2    5
  x    1    3
  y    1    7
  y    3    1
  x    3    6
  z    1    4
  z    2    8
  z    3    7
  z    4    0
  p    2    6
  p    4    7
  p    1    9
  p    6    0
  p    7    3
```

Expected Output

Our expected output is as follows. Note that the values of reducers are grouped by name and sorted by time:

```
name   t1   t2  t3   t4   t5 ...
x =>  [3,   9,  6]
y =>  [7,   5,  1]
z =>  [4,   8,  7,   0]
p =>  [9,   6,  7,   0,   3]
```

Option 1: Secondary Sorting in Memory

Since Spark has a very powerful and high-level API, I will present the entire solution in a single Java class. The Spark API is built upon the basic abstraction concept of the RDD (resilient distributed data set). To fully utilize Spark’s API, we have to understand RDDs. An `RDD<T>` (i.e., an RDD of type `T`) *object* represents an immutable, partitioned collection of elements (of type `T`) that can be operated on in parallel. The `RDD<T>` *class* contains the basic MapReduce operations available on all RDDs, such as `map()`, `filter()`, and `persist()`,

while the `JavaPairRDD<K,V>` class contains MapReduce operations such as `mapToPair()`, `flatMapToPair()`, and `groupByKey()`. In addition, Spark's `PairRDDFunctions` contains operations available only on RDDs of key-value pairs, such as `reduce()`, `groupByKey()`, and `join()`. (For details on RDDs, see Spark's API and Appendix B of this book.) Therefore, `JavaRDD<T>` is a list of objects of type `T`, and `JavaPairRDD<K,V>` is a list of objects of type `Tuple2<K,V>` (where each tuple represents a key-value pair).

The Spark-based algorithm is listed next. Although there are 10 steps, most of them are trivial and some are provided for debugging purposes only:

1. We import the required Java/Spark classes. The main Java classes for MapReduce are given in the `org.apache.spark.api.java` package. This package includes the following classes and interfaces:
   - `JavaRDDLike` (interface)
   - `JavaDoubleRDD`
   - `JavaPairRDD`
   - `JavaRDD`
   - `JavaSparkContext`
   - `StorageLevels`
2. We pass input data as arguments and validate.
3. We connect to the Spark master by creating a `JavaSparkContext` object, which is used to create new RDDs.
4. Using the context object (created in step 3), we create an RDD for the input file; the resulting RDD will be a `JavaRDD<String>`. Each element of this RDD will be a record of time series data: `<name><,><time><,><value>`.
5. Next we want to create key-value pairs from a `JavaRDD<String>`, where the key is the `name` and the value is a pair of (`time`, `value`). The resulting RDD will be a `JavaPairRDD<String, Tuple2<Integer, Integer>>`.
6. To validate step 5, we collect all values from the `JavaPairRDD<>` and print

them.

7. We group `JavaPairRDD<>` elements by the key (`name`). To accomplish this, we use the `groupByKey()` method.
   The result will be the RDD:

   ```
   JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>>
   ```

   Note that the resulting list (`Iterable<Tuple2<Integer, Integer>>`) is unsorted. In general, Spark's `reduceByKey()` is preferred over `groupByKey()` for performance reasons, but here we have no other option than `groupByKey()` (since `reduceByKey()` does not allow us to sort the values in place for a given key).

8. To validate step 7, we collect all values from the `JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>>` and print them.

9. We sort the reducer's values to get the final output. We accomplish this by writing a custom `mapValues()` method. We just sort the values (the key remains the same).

10. To validate the final result, we collect all values from the sorted `JavaPairRDD<>` and print them.

A solution for option #1 is implemented by a single driver class: `SecondarySorting` (see Example 1-7). All steps, 1–10, are listed inside the class definition, which will be presented in the following sections. Typically, a Spark application consists of a driver program that runs the user's `main()` function and executes various parallel operations on a cluster. Parallel operations will be achieved through the extensive use of RDDs. For further details on RDDs, see Appendix B.

Example 1-7. SecondarySort class overall structure

```
1 // Step 1: import required Java/Spark classes
2 public class SecondarySort {
3   public static void main(String[] args) throws Exception {
4     // Step 2: read input parameters and validate them
5     // Step 3: connect to the Spark master by creating a JavaSparkContext
6     // object (ctx)
6     // Step 4: use ctx to create JavaRDD<String>
7     // Step 5: create key-value pairs from JavaRDD<String>, where
8     // key is the {name} and value is a pair of (time, value)
```

```
    // Step 6: validate step 5-collect all values from JavaPairRDD<>
    // and print them
    // Step 7: group JavaPairRDD<> elements by the key ({name})
    // Step 8: validate step 7-collect all values from JavaPairRDD<>
    // and print them
    // Step 9: sort the reducer's values; this will give us the final output
    // Step 10: validate step 9-collect all values from JavaPairRDD<>
    // and print them

    // done
    ctx.close();
    System.exit(0);
  }
}
```

Step 1: Import required classes

As shown in Example 1-8, the main Spark package for the Java API is `org.apache.spark.api.java`, which includes the `JavaRDD`, `JavaPairRDD`, and `JavaSparkContext` classes. `JavaSparkContext` is a factory class for creating new RDDs (such as `JavaRDD` and `JavaPairRDD` objects).

Example 1-8. Step 1: Import required classes

```
// Step 1: import required Java/Spark classes
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;

import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.Collections;
import java.util.Comparator;
```

Step 2: Read input parameters

This step, demonstrated in Example 1-9, reads the HDFS input file (Spark may read data from HDFS and other persistent stores, such as a Linux filesystem), which might look like */dir1/dir2/myfile.txt*.

Example 1-9. Step 2: Read input parameters

```
   // Step 2: read input parameters and validate them
   if (args.length < 1) {
      System.err.println("Usage: SecondarySort <file>");
      System.exit(1);
   }
   String inputPath = args[0];
   System.out.println("args[0]: <file>="+args[0]);
```

Step 3: Connect to the Spark master

To work with RDDs, first you need to create a `JavaSparkContext` object (as shown in Example 1-10), which is a factory for creating `JavaRDD` and `JavaPairRDD` objects. It is also possible to create a `JavaSparkContext` object by injecting a `SparkConf` object into the `JavaSparkContext`'s class constructor. This approach is useful when you read your cluster configurations from an XML file. In a nutshell, the `JavaSparkContext` object has the following responsibilities:

- Initializes the application driver.
- Registers the application driver to the cluster manager. (If you are using the Spark cluster, then this will be the Spark master; if you are using YARN, then it will be YARN's resource manager.)
- Obtains a list of executors for executing your application driver.

Example 1-10. Step 3: Connect to the Spark master

```
1    // Step 3: connect to the Spark master by creating a JavaSparkContext object
2    final JavaSparkContext ctx = new JavaSparkContext();
```

Step 4: Use the JavaSparkContext to create a JavaRDD

This step, illustrated in Example 1-11, reads an HDFS file and creates a `JavaRDD<String>` (which represents a set of records where each record is a `String` object). By definition, Spark's RDDs are *immutable* (i.e., they cannot be altered or modified). Note that Spark's RDDs are the basic abstraction for parallel execution. Note also that you may use `textFile()` to read HDFS or non-HDFS files.

Example 1-11. Step 4: Create JavaRDD

```
1    // Step 4: use ctx to create JavaRDD<String>
2    // input record format: <name><,><time><,><value>
3    JavaRDD<String> lines = ctx.textFile(inputPath, 1);
```

Step 5: Create key-value pairs from the JavaRDD

This step, shown in Example 1-12, implements a mapper. Each record (from the `JavaRDD<String>` and consisting of `<name><,><time><,><value>`) is converted to a key-value pair, where the key is a `name` and the value is a `Tuple2(time, value)`.

Example 1-12. Step 5: Create key-value pairs from JavaRDD

```
1    // Step 5: create key-value pairs from JavaRDD<String>, where
```

```
  // key is the {name} and value is a pair of (time, value).
  // The resulting RDD will be a JavaPairRDD<String, Tuple2<Integer, Integer>>.
  // Convert each record into Tuple2(name, time, value).
  // PairFunction<T, K, V>
  //     T => Tuple2(K, V) where T is input (as String),
  //     K=String
  //     V=Tuple2<Integer, Integer>
  JavaPairRDD<String, Tuple2<Integer, Integer>> pairs =
        lines.mapToPair(new PairFunction<
                                        String,                  // T
                                        String,                  // K
                                        Tuple2<Integer, Integer> // V
                                       >() {
    public Tuple2<String, Tuple2<Integer, Integer>> call(String s) {
      String[] tokens = s.split(","); // x,2,5
      System.out.println(tokens[0] + "," + tokens[1] + "," + tokens[2]);
      Integer time = new Integer(tokens[1]);
      Integer value = new Integer(tokens[2]);
      Tuple2<Integer, Integer> timevalue =
      new Tuple2<Integer, Integer>(time, value);
      return new Tuple2<String, Tuple2<Integer, Integer>>(tokens[0], timevalue);
    }
  });
```

Step 6: Validate step 5

To debug and validate your steps in Spark (as shown in Example 1-13), you may use `JavaRDD.collect()` and `JavaPairRDD.collect()`. Note that `collect()` is used for debugging and educational purposes (but avoid using `collect()` for debugging purposes in production clusters; doing so will impact performance). Also, you may use `JavaRDD.saveAsTextFile()` for debugging as well as creating your desired outputs.

Example 1-13. Step 6: Validate step 5

```
  // Step 6: validate step 5-collect all values from JavaPairRDD<>
  // and print them
  List<Tuple2<String, Tuple2<Integer, Integer>>> output = pairs.collect();
  for (Tuple2 t : output) {
     Tuple2<Integer, Integer> timevalue = (Tuple2<Integer, Integer>) t._2;
     System.out.println(t._1 + "," + timevalue._1 + "," + timevalue._1);
  }
```

Step 7: Group JavaPairRDD elements by the key (name)

We implement the reducer operation using `groupByKey()`. As you can see in Example 1-14, it is much easier to implement the reducer through Spark than MapReduce/Hadoop. Note that in Spark, in general, `reduceByKey()` is more efficient than `groupByKey()`. Here, however, we cannot use `reduceByKey()`.

Example 1-14. Step 7: Group JavaPairRDD elements

```
  // Step 7: group JavaPairRDD<> elements by the key ({name})
  JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> groups =
```

```
3          pairs.groupByKey();
```

Step 8: Validate step 7

This step, shown in Example 1-15, validates the previous step by using the `collect()` function, which gets all values from the `groups` RDD.

Example 1-15. Step 8: Validate step 7

```
1    // Step 8: validate step 7-we collect all values from JavaPairRDD<>
2    // and print them
2    System.out.println("===DEBUG1===");
3    List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>>> output2 =
4          groups.collect();
5    for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output2) {
6       Iterable<Tuple2<Integer, Integer>> list = t._2;
7       System.out.println(t._1);
8       for (Tuple2<Integer, Integer> t2 : list) {
9          System.out.println(t2._1 + "," + t2._2);
10      }
11      System.out.println("=====");
12   }
```

The following shows the output of this step. As you can see, the reducer values are not sorted:

```
y
2,5
1,7
3,1
=====
x
2,9
1,3
3,6
=====
z
1,4
2,8
3,7
4,0
=====
p
2,6
4,7
6,0
7,3
1,9
=====
```

Step 9: Sort the reducer's values in memory

This step, shown in Example 1-16, uses another powerful Spark method, `mapValues()`, to just sort the values generated by reducers. The `mapValues()` method enables us to convert (K, V_1) into (K, V_2), where V_2 is a sorted V_1. One

important note about Spark's RDD is that it is immutable and cannot be altered/updated by any means. For example, in this step, to sort our values, we have to copy them into another list first. Immutability applies to the RDD itself and its elements.

Example 1-16. Step 9: sort the reducer's values in memory

```
// Step 9: sort the reducer's values; this will give us the final output.
// Option #1: worked
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// Pass each value in the key-value pair RDD through a map function
// without changing the keys;
// this also retains the original RDD's partitioning.
JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> sorted =
      groups.mapValues(
        new Function<Iterable<Tuple2<Integer, Integer>>,  // input
                     Iterable<Tuple2<Integer, Integer>>   // output
                    >() {
  public Iterable<Tuple2<Integer, Integer>> call(Iterable<Tuple2<Integer,
                                                                 Integer>> s) {
    List<Tuple2<Integer, Integer>> newList = new ArrayList<Tuple2<Integer,
                                                                  Integer>>(s);
    Collections.sort(newList, new TupleComparator());
    return newList;
  }
});
```

Step 10: output final result

The `collect()` method collects all of the RDD's elements into a `java.util.List` object. Then we iterate through the `List` to get all the final elements (see Example 1-17).

Example 1-17. Step 10: Output final result

```
// Step 10: validate step 9-collect all values from JavaPairRDD<>
// and print them
System.out.println("===DEBUG2=");
List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>>> output3 =
      sorted.collect();
for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output3) {
   Iterable<Tuple2<Integer, Integer>> list = t._2;
   System.out.println(t._1);
   for (Tuple2<Integer, Integer> t2 : list) {
      System.out.println(t2._1 + "," + t2._2);
   }
   System.out.println("=====");
}
```

Spark Sample Run

As far as Spark/Hadoop is concerned, you can run a Spark application in three different modes:[1]

Standalone mode

This is the default setup. You start the Spark master on a master node and a "worker" on every slave node, and submit your Spark application to the Spark master.

YARN client mode

In this mode, you do not start a Spark master or worker nodes. Instead, you submit the Spark application to YARN, which runs the Spark driver in the client Spark process that submits the application.

YARN cluster mode

In this mode, you do not start a Spark master or worker nodes. Instead, you submit the Spark application to YARN, which runs the Spark driver in the ApplicationMaster in YARN.

Next, we will cover how to submit the secondary sort application in the standalone and YARN cluster modes.

Running Spark in standalone mode

The following subsections provide the input, script, and log output of a sample run of our secondary sort application in Spark's standalone mode.

HDFS input

```
# hadoop fs -cat /mp/timeseries.txt
x,2,9
y,2,5
x,1,3
y,1,7
y,3,1
x,3,6
z,1,4
z,2,8
z,3,7
z,4,0
p,2,6
p,4,7
p,1,9
p,6,0
p,7,3
```

The script

```
# cat run_secondarysorting.sh
#!/bin/bash
```

```
export JAVA_HOME=/usr/java/jdk7
export SPARK_HOME=/home/hadoop/spark-1.1.0
export SPARK_MASTER=spark://myserver100:7077
BOOK_HOME=/home/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/home/hadoop/testspark/timeseries.txt
# Run on a Spark standalone cluster
prog=org.dataalgorithms.chap01.spark.SparkSecondarySort
$SPARK_HOME/bin/spark-submit \
  --class $prog \
  --master $SPARK_MASTER \
  --executor-memory 2G \
  --total-executor-cores 20 \
  $APP_JAR \
  $INPUT
```

Log of the run

```
# ./run_secondarysorting.sh
args[0]: <file>=/mp/timeseries.txt
...
===  DEBUG STEP 5 ===
...
x,2,2
y,2,2
x,1,1
y,1,1
y,3,3
x,3,3
z,1,1
z,2,2
z,3,3
z,4,4
p,2,2
p,4,4
p,1,1
p,6,6
p,7,7
===  DEBUG STEP 7 ===
14/06/04 08:42:54 INFO spark.SparkContext: Starting job: collect
  at SecondarySort.java:96
14/06/04 08:42:54 INFO scheduler.DAGScheduler: Registering RDD 2
  (mapToPair at SecondarySort.java:75)
...
14/06/04 08:42:55 INFO scheduler.DAGScheduler: Stage 1
  (collect at SecondarySort.java:96) finished in 0.273 s
14/06/04 08:42:55 INFO spark.SparkContext: Job finished:
  collect at SecondarySort.java:96, took 1.587001929 s
z
1,4
2,8
3,7
4,0
=====
p
2,6
4,7
1,9
6,0
7,3
=====
x
```

```
2,9
1,3
3,6
=====
y
2,5
1,7
3,1
=====
===  DEBUG STEP 9 ===
14/06/04 08:42:55 INFO spark.SparkContext: Starting job: collect
  at SecondarySort.java:158
...
14/06/04 08:42:55 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 3.0,
  whose tasks have all completed, from pool
14/06/04 08:42:55 INFO spark.SparkContext: Job finished: collect at
  SecondarySort.java:158, took 0.074271723 s
z
1,4
2,8
3,7
4,0
=====
p
1,9
2,6
4,7
6,0
7,3
=====
x
1,3
2,9
3,6
=====
y
1,7
2,5
3,1
=====
```

Typically, you save the final result to HDFS. You can accomplish this by adding the following line of code after creating your “sorted” RDD:

```
sorted.saveAsTextFile("/mp/output");
```

Then you may view the output as follows:

```
# hadoop fs -ls /mp/output/
Found 2 items
-rw-r--r--   3 hadoop root,hadoop      0 2014-06-04 10:49 /mp/output/_SUCCESS
-rw-r--r--   3 hadoop root,hadoop    125 2014-06-04 10:49 /mp/output/part-00000

# hadoop fs -cat /mp/output/part-00000
(z,[(1,4), (2,8), (3,7), (4,0)])
(p,[(1,9), (2,6), (4,7), (6,0), (7,3)])
(x,[(1,3), (2,9), (3,6)])
(y,[(1,7), (2,5), (3,1)])
```

Running Spark in YARN cluster mode

The script to submit our Spark application in YARN cluster mode is as follows:

```
# cat run_secondarysorting_yarn.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7
export HADOOP_HOME=/usr/local/hadoop-2.5.0
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export SPARK_HOME=/home/hadoop/spark-1.1.0
BOOK_HOME=/home/mp/data-algorithms-book
APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar
INPUT=/mp/timeseries.txt
prog=org.dataalgorithms.chap01.spark.SparkSecondarySort
$SPARK_HOME/bin/spark-submit \
  --class $prog \
  --master yarn-cluster \
  --executor-memory 2G \
  --num-executors 10 \
  $APP_JAR \
  $INPUT
```

Option #2: Secondary Sorting Using the Spark Framework

In the solution for option #1, we sorted reducer values in memory (using Java's `Collections.sort()` method), which might not scale if the reducer values will not fit in a commodity server's memory. Next we will implement option #2 for the MapReduce/Hadoop framework. We cannot achieve this in the current Spark (Spark-1.1.0) framework, because currently Spark's shuffle is based on a hash, which is different from MapReduce's sort-based shuffle. So, you should implement sorting explicitly using an RDD operator. If we had a partitioner by a natural key (`name`) that preserved the order of the RDD, that would be a viable solution — for example, if we sorted by `(name, time)`, we would get:

```
(p,1),(1,9)
(p,4),(4,7)
(p,6),(6,0)
(p,7),(7,3)

(x,1),(1,3)
(x,2),(2,9)
(x,3),(3,6)

(y,1),(1,7)
(y,2),(2,5)
(y,3),(3,1)

(z,1),(1,4)
(z,2),(2,8)
(z,3),(3,7)
(z,4),(4,0)
```

There is a partitioner (represented as an abstract class, `org.apache.spark.Partitioner`), but it does not preserve the order of the original RDD elements. Therefore, option #2 cannot be implemented by the current version of Spark (1.1.0).

Further Reading on Secondary Sorting

To support secondary sorting in Spark, you may extend the `JavaPairRDD` class and add additional methods such as `groupByKeyAndSortValues()`. For further work on this topic, you may refer to the following:

- Support sorting of values in addition to keys (i.e., secondary sort)
- *https://github.com/tresata/spark-sorted*

Chapter 2 provides a detailed implementation of the Secondary Sort design pattern using the MapReduce and Spark frameworks.

[1] For details, see the Spark documentation.