Data Warehouse and Query Language for Hadoop

Programming

Hive

Edward Capriolo,
Dean Wampler &
Jason Rutherglen,

O'REILLY®

Programming Hive
by Edward Capriolo, Dean Wampler, and Jason Rutherglen

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://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Mike Loukides and Courtney Nash
Production Editors: Iris Febres and Rachel Steely
Proofreaders: Stacie Arellano and Kiel Van Horn
Indexer: Bob Pfahler
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

October 2012: First Edition.

Revision History for the First Edition:
2012-09-17 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449319335* for release details.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. *Programming Hive*, the image of a hornet's hive, and related trade dress are trademarks of O'Reilly Media, Inc.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and O'Reilly Media, Inc., was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

ISBN: 978-1-449-31933-5

[LSI]

1347905436

Table of Contents

CHAPTER 1
Introduction

From the early days of the Internet's mainstream breakout, the major search engines and ecommerce companies wrestled with ever-growing quantities of data. More recently, social networking sites experienced the same problem. Today, many organizations realize that the data they gather is a valuable resource for understanding their customers, the performance of their business in the marketplace, and the effectiveness of their infrastructure.

The *Hadoop* ecosystem emerged as a cost-effective way of working with such large data sets. It imposes a particular programming model, called *MapReduce*, for breaking up computation tasks into units that can be distributed around a cluster of commodity, server class hardware, thereby providing cost-effective, horizontal scalability. Underneath this computation model is a distributed file system called the *Hadoop Distributed Filesystem* (HDFS). Although the filesystem is "pluggable," there are now several commercial and open source alternatives.

However, a challenge remains; how do you move an existing data infrastructure to Hadoop, when that infrastructure is based on traditional relational databases and the *Structured Query Language* (SQL)? What about the large base of SQL users, both expert database designers and administrators, as well as casual users who use SQL to extract information from their data warehouses?

This is where *Hive* comes in. Hive provides an *SQL* dialect, called *Hive Query Language* (abbreviated *HiveQL* or just *HQL*) for querying data stored in a Hadoop cluster.

SQL knowledge is widespread for a reason; it's an effective, reasonably intuitive model for organizing and using data. Mapping these familiar data operations to the low-level MapReduce Java API can be daunting, even for experienced Java developers. Hive does this dirty work for you, so you can focus on the query itself. Hive translates most queries to MapReduce jobs, thereby exploiting the scalability of Hadoop, while presenting a familiar SQL abstraction. If you don't believe us, see "Java Versus Hive: The Word Count Algorithm" on page 10 later in this chapter.

Hive is most suited for *data warehouse* applications, where relatively static data is analyzed, fast response times are not required, and when the data is not changing rapidly.

Hive is not a full database. The design constraints and limitations of Hadoop and HDFS impose limits on what Hive can do. The biggest limitation is that Hive does not provide record-level update, insert, nor delete. You can generate new tables from queries or output query results to files. Also, because Hadoop is a batch-oriented system, Hive queries have higher latency, due to the start-up overhead for MapReduce jobs. Queries that would finish in seconds for a traditional database take longer for Hive, even for relatively small data sets.[1] Finally, Hive does not provide transactions.

So, Hive doesn't provide crucial features required for OLTP, *Online Transaction Processing*. It's closer to being an OLAP tool, *Online Analytic Processing*, but as we'll see, Hive isn't ideal for satisfying the "online" part of OLAP, at least today, since there can be significant latency between issuing a query and receiving a reply, both due to the overhead of Hadoop and due to the size of the data sets Hadoop was designed to serve.

If you need OLTP features for large-scale data, you should consider using a *NoSQL* database. Examples include *HBase*, a *NoSQL* database integrated with Hadoop,[2] *Cassandra*,[3] and *DynamoDB*, if you are using Amazon's Elastic MapReduce (EMR) or Elastic Compute Cloud (EC2).[4] You can even integrate Hive with these databases (among others), as we'll discuss in Chapter 17.

So, Hive is best suited for data warehouse applications, where a large data set is maintained and mined for insights, reports, etc.

Because most data warehouse applications are implemented using SQL-based relational databases, Hive lowers the barrier for moving these applications to Hadoop. People who know SQL can learn Hive easily. Without Hive, these users would need to learn new languages and tools to be productive again.

Similarly, Hive makes it easier for developers to port SQL-based applications to Hadoop, compared with other Hadoop languages and tools.

However, like most SQL dialects, HiveQL does not conform to the ANSI SQL standard and it differs in various ways from the familiar SQL dialects provided by Oracle, MySQL, and SQL Server. (However, it is closest to MySQL's dialect of SQL.)

1. However, for the big data sets Hive is designed for, this start-up overhead is trivial compared to the actual processing time.
2. See the Apache HBase website, *http://hbase.apache.org*, and *HBase: The Definitive Guide* by Lars George (O'Reilly).
3. See the Cassandra website, *http://cassandra.apache.org/*, and *High Performance Cassandra Cookbook* by Edward Capriolo (Packt).
4. See the DynamoDB website, *http://aws.amazon.com/dynamodb/*.

So, this book has a dual purpose. First, it provides a comprehensive, example-driven introduction to HiveQL for all users, from developers, database administrators and architects, to less technical users, such as business analysts.

Second, the book provides the in-depth technical details required by developers and Hadoop administrators to tune Hive query performance and to customize Hive with *user-defined functions*, custom data formats, etc.

We wrote this book out of frustration that Hive lacked good documentation, especially for new users who aren't developers and aren't accustomed to browsing project artifacts like bug and feature databases, source code, etc., to get the information they need. The Hive Wiki[5] is an invaluable source of information, but its explanations are sometimes sparse and not always up to date. We hope this book remedies those issues, providing a single, comprehensive guide to all the essential features of Hive and how to use them effectively.[6]

An Overview of Hadoop and MapReduce

If you're already familiar with Hadoop and the *MapReduce* computing model, you can skip this section. While you don't need an intimate knowledge of MapReduce to use Hive, understanding the basic principles of MapReduce will help you understand what Hive is doing behind the scenes and how you can use Hive more effectively.

We provide a brief overview of Hadoop and MapReduce here. For more details, see *Hadoop: The Definitive Guide* by Tom White (O'Reilly).

MapReduce

MapReduce is a computing model that decomposes large data manipulation *jobs* into individual *tasks* that can be executed in parallel across a cluster of servers. The results of the tasks can be joined together to compute the final results.

The *MapReduce* programming model was developed at Google and described in an influential paper called *MapReduce: simplified data processing on large clusters* (see the Appendix) on page 309. The *Google Filesystem* was described a year earlier in a paper called The Google filesystem on page 310. Both papers inspired the creation of Hadoop by Doug Cutting.

The term *MapReduce* comes from the two fundamental data-transformation operations used, *map* and *reduce*. A *map* operation converts the elements of a collection from one form to another. In this case, input key-value pairs are converted to zero-to-many

5. See *https://cwiki.apache.org/Hive/*.
6. It's worth bookmarking the wiki link, however, because the wiki contains some more obscure information we won't cover here.

output key-value pairs, where the input and output keys might be completely different and the input and output values might be completely different.

In *MapReduce*, all the key-pairs for a given key are sent to the same *reduce* operation. Specifically, the key and a collection of the values are passed to the reducer. The goal of "reduction" is to convert the collection to a value, such as summing or averaging a collection of numbers, or to another collection. A final key-value pair is emitted by the reducer. Again, the input versus output keys and values may be different. Note that if the job requires no reduction step, then it can be skipped.

An implementation infrastructure like the one provided by *Hadoop* handles most of the chores required to make jobs run successfully. For example, Hadoop determines how to decompose the submitted *job* into individual map and reduce *tasks* to run, it schedules those tasks given the available resources, it decides where to send a particular task in the cluster (usually where the corresponding data is located, when possible, to minimize network overhead), it monitors each task to ensure successful completion, and it restarts tasks that fail.

The *Hadoop Distributed Filesystem*, HDFS, or a similar distributed filesystem, manages data across the cluster. Each block is replicated several times (three copies is the usual default), so that no single hard drive or server failure results in data loss. Also, because the goal is to optimize the processing of very large data sets, HDFS and similar filesystems use very large block sizes, typically 64 MB or multiples thereof. Such large blocks can be stored contiguously on hard drives so they can be written and read with minimal seeking of the drive heads, thereby maximizing write and read performance.

To make MapReduce more clear, let's walk through a simple example, the *Word Count* algorithm that has become the "Hello World" of MapReduce.[7] Word Count returns a list of all the words that appear in a corpus (one or more documents) and the count of how many times each word appears. The output shows each word found and its count, one per line. By common convention, the word (output key) and count (output value) are usually separated by a tab separator.

Figure 1-1 shows how Word Count works in MapReduce.

There is a lot going on here, so let's walk through it from left to right.

Each *Input* box on the left-hand side of Figure 1-1 is a separate document. Here are four documents, the third of which is empty and the others contain just a few words, to keep things simple.

By default, a separate *Mapper* process is invoked to process each document. In real scenarios, large documents might be split and each split would be sent to a separate Mapper. Also, there are techniques for combining many small documents into a single *split* for a Mapper. We won't worry about those details now.

7. If you're not a developer, a "Hello World" program is the traditional first program you write when learning a new language or tool set.

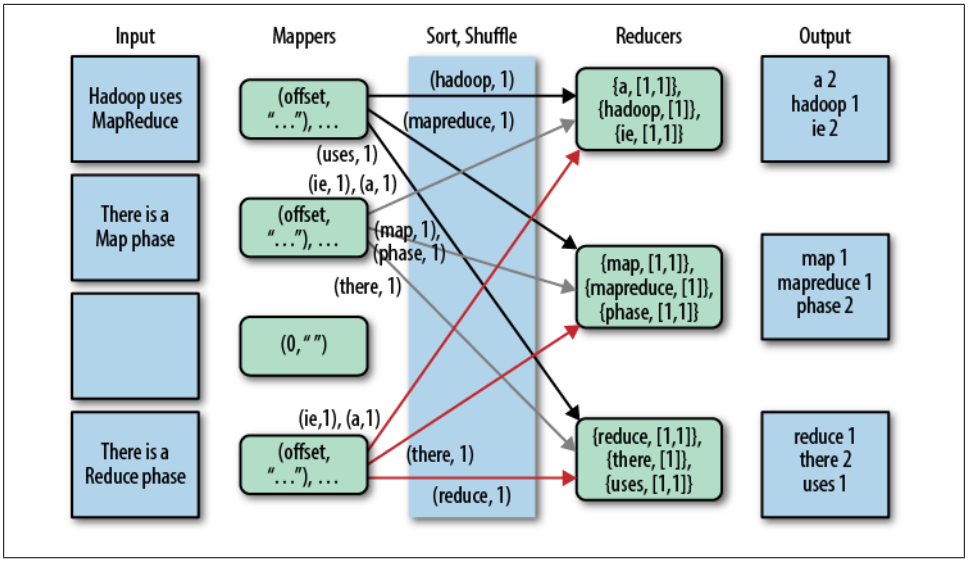

Figure 1-1. Word Count algorithm using MapReduce

The fundamental data structure for input and output in MapReduce is the key-value pair. After each Mapper is started, it is called repeatedly for each line of text from the document. For each call, the key passed to the mapper is the character offset into the document at the start of the line. The corresponding value is the text of the line.

In Word Count, the character offset (key) is discarded. The value, the line of text, is tokenized into words, using one of several possible techniques (e.g., splitting on white-space is the simplest, but it can leave in undesirable punctuation). We'll also assume that the Mapper converts each word to lowercase, so for example, "FUN" and "fun" will be counted as the same word.

Finally, for each word in the line, the mapper outputs a key-value pair, with the word as the key and the number 1 as the value (i.e., the count of "one occurrence"). Note that the output *types* of the keys and values are different from the input types.

Part of Hadoop's magic is the *Sort and Shuffle* phase that comes next. Hadoop sorts the key-value pairs by key and it "shuffles" all pairs with the same key to the same *Reducer*. There are several possible techniques that can be used to decide which reducer gets which range of keys. We won't worry about that here, but for illustrative purposes, we have assumed in the figure that a particular alphanumeric partitioning was used. In a real implementation, it would be different.

For the mapper to simply output a count of 1 every time a word is seen is a bit wasteful of network and disk I/O used in the sort and shuffle. (It does minimize the memory used in the Mappers, however.) One optimization is to keep track of the count for each word and then output only one count for each word when the Mapper finishes. There

are several ways to do this optimization, but the simple approach is logically correct and sufficient for this discussion.

The inputs to each *Reducer* are again key-value pairs, but this time, each key will be one of the words found by the mappers and the value will be a *collection* of all the counts emitted by all the mappers for that word. Note that the type of the key and the type of the value collection elements are the same as the types used in the Mapper's output. That is, the key type is a character string and the value collection element type is an integer.

To finish the algorithm, all the reducer has to do is add up all the counts in the value collection and write a final key-value pair consisting of each word and the count for that word.

Word Count isn't a toy example. The data it produces is used in spell checkers, language detection and translation systems, and other applications.

Hive in the Hadoop Ecosystem

The Word Count algorithm, like most that you might implement with Hadoop, is a little involved. When you actually implement such algorithms using the Hadoop Java API, there are even more low-level details you have to manage yourself. It's a job that's only suitable for an experienced Java developer, potentially putting Hadoop out of reach of users who aren't programmers, even when they understand the algorithm they want to use.

In fact, many of those low-level details are actually quite repetitive from one job to the next, from low-level chores like wiring together Mappers and Reducers to certain data manipulation constructs, like filtering for just the data you want and performing SQL-like joins on data sets. There's a real opportunity to eliminate reinventing these idioms by letting "higher-level" tools handle them automatically.

That's where Hive comes in. It not only provides a familiar programming model for people who know SQL, it also eliminates lots of boilerplate and sometimes-tricky coding you would have to do in Java.

This is why Hive is so important to Hadoop, whether you are a DBA or a Java developer. Hive lets you complete a lot of work with relatively little effort.

Figure 1-2 shows the major "modules" of Hive and how they work with Hadoop.

There are several ways to interact with Hive. In this book, we will mostly focus on the CLI, *command-line interface*. For people who prefer graphical user interfaces, commercial and open source options are starting to appear, including a commercial product from Karmasphere (*http://karmasphere.com*), Cloudera's open source *Hue* (*https://github.com/cloudera/hue*), a new "Hive-as-a-service" offering from Qubole (*http://qubole.com*), and others.

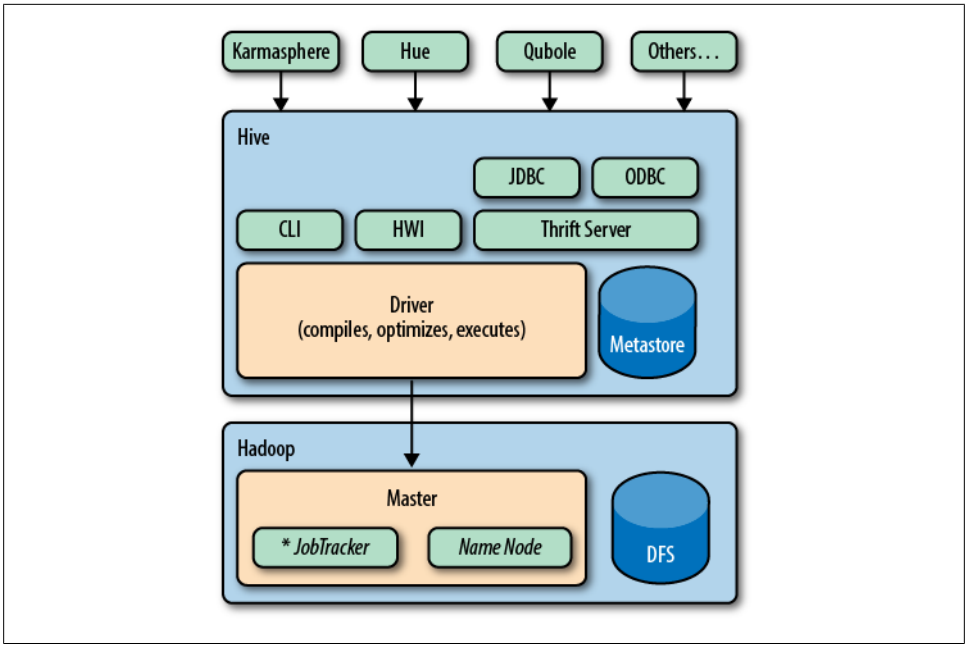

Figure 1-2. Hive modules

Bundled with the Hive distribution is the CLI, a simple web interface called *Hive web interface* (HWI), and programmatic access through JDBC, ODBC, and a Thrift server (see Chapter 16).

All commands and queries go to the Driver, which compiles the input, optimizes the computation required, and executes the required steps, usually with MapReduce jobs.

When MapReduce jobs are required, Hive doesn't generate Java MapReduce programs. Instead, it uses built-in, generic Mapper and Reducer modules that are driven by an XML file representing the "job plan." In other words, these generic modules function like mini language interpreters and the "language" to drive the computation is encoded in XML.

Hive communicates with the *JobTracker* to initiate the MapReduce job. Hive does not have to be running on the same master node with the JobTracker. In larger clusters, it's common to have edge nodes where tools like Hive run. They communicate remotely with the JobTracker on the master node to execute jobs. Usually, the data files to be processed are in HDFS, which is managed by the *NameNode*.

The Metastore is a separate relational database (usually a MySQL instance) where Hive persists table schemas and other system metadata. We'll discuss it in detail in Chapter 2.

While this is a book about Hive, it's worth mentioning other higher-level tools that you should consider for your needs. Hive is best suited for data warehouse applications, where real-time responsiveness to queries and record-level inserts, updates, and deletes

are not required. Of course, Hive is also very nice for people who know SQL already. However, some of your work may be easier to accomplish with alternative tools.

Pig

The best known alternative to Hive is Pig (see *http://pig.apache.org*), which was developed at Yahoo! about the same time Facebook was developing Hive. Pig is also now a top-level Apache project that is closely associated with Hadoop.

Suppose you have one or more sources of input data and you need to perform a complex set of transformations to generate one or more collections of output data. Using Hive, you might be able to do this with nested queries (as we'll see), but at some point it will be necessary to resort to temporary tables (which you have to manage yourself) to manage the complexity.

Pig is described as a *data flow* language, rather than a query language. In Pig, you write a series of declarative statements that define *relations* from other relations, where each new relation performs some new data transformation. Pig looks at these declarations and then builds up a sequence of MapReduce jobs to perform the transformations until the final results are computed the way that you want.

This step-by-step "flow" of data can be more intuitive than a complex set of queries. For this reason, Pig is often used as part of ETL (Extract, Transform, and Load) processes used to ingest external data into a Hadoop cluster and transform it into a more desirable form.

A drawback of Pig is that it uses a custom language not based on SQL. This is appropriate, since it is not designed as a query language, but it also means that Pig is less suitable for porting over SQL applications and experienced SQL users will have a larger learning curve with Pig.

Nevertheless, it's common for Hadoop teams to use a combination of Hive and Pig, selecting the appropriate tool for particular jobs.

Programming Pig by Alan Gates (O'Reilly) provides a comprehensive introduction to Pig.

HBase

What if you need the database features that Hive doesn't provide, like row-level updates, rapid query response times, and transactions?

HBase is a distributed and scalable data store that supports row-level updates, rapid queries, and row-level transactions (but not multirow transactions).

HBase is inspired by Google's *Big Table*, although it doesn't implement all Big Table features. One of the important features HBase supports is column-oriented storage, where columns can be organized into *column families*. Column families are physically

stored together in a distributed cluster, which makes reads and writes faster when the typical query scenarios involve a small subset of the columns. Rather than reading entire rows and discarding most of the columns, you read only the columns you need.

HBase can be used like a key-value store, where a single key is used for each row to provide very fast reads and writes of the row's columns or column families. HBase also keeps a configurable number of versions of each column's values (marked by timestamps), so it's possible to go "back in time" to previous values, when needed.

Finally, what is the relationship between HBase and Hadoop? HBase uses HDFS (or one of the other distributed filesystems) for durable file storage of data. To provide row-level updates and fast queries, HBase also uses in-memory caching of data and local files for the append log of updates. Periodically, the durable files are updated with all the append log updates, etc.

HBase doesn't provide a query language like SQL, but Hive is now integrated with HBase. We'll discuss this integration in "HBase" on page 222.

For more on HBase, see the HBase website, and *HBase: The Definitive Guide* by Lars George.

Cascading, Crunch, and Others

There are several other "high-level" languages that have emerged outside of the Apache Hadoop umbrella, which also provide nice abstractions on top of Hadoop to reduce the amount of low-level boilerplate code required for typical jobs. For completeness, we list several of them here. All are JVM (Java Virtual Machine) libraries that can be used from programming languages like Java, Clojure, Scala, JRuby, Groovy, and Jython, as opposed to tools with their own languages, like Hive and Pig.

Using one of these programming languages has advantages and disadvantages. It makes these tools less attractive to nonprogrammers who already know SQL. However, for developers, these tools provide the full power of a *Turing complete* programming language. Neither Hive nor Pig are Turing complete. We'll learn how to extend Hive with Java code when we need additional functionality that Hive doesn't provide (Table 1-1).

Table 1-1. Alternative higher-level libraries for Hadoop

Name	URL	Description
Cascading	*http://cascading.org*	Java API with Data Processing abstractions. There are now many Domain Specific Languages (DSLs) for Cascading in other languages, e.g., Scala, Groovy, JRuby, and Jython.
Cascalog	*https://github.com/nathanmarz/cascalog*	A Clojure DSL for Cascading that provides additional functionality inspired by Datalog for data processing and query abstractions.
Crunch	*https://github.com/cloudera/crunch*	A Java and Scala API for defining data flow pipelines.

Because Hadoop is a batch-oriented system, there are tools with different distributed computing models that are better suited for *event stream* processing, where closer to "real-time" responsiveness is required. Here we list several of the many alternatives (Table 1-2).

Table 1-2. Distributed data processing tools that don't use MapReduce

Name	URL	Description
Spark	*http://www.spark-project.org/*	A distributed computing framework based on the idea of distributed data sets with a Scala API. It can work with HDFS files and it offers notable performance improvements over Hadoop MapReduce for many computations. There is also a project to port Hive to Spark, called Shark (*http://shark.cs.berkeley.edu/*).
Storm	*https://github.com/nathanmarz/storm*	A real-time event stream processing system.
Kafka	*http://incubator.apache.org/kafka/index.html*	A distributed publish-subscribe messaging system.

Finally, it's important to consider when you *don't* need a full cluster (e.g., for smaller data sets or when the time to perform a computation is less critical). Also, many alternative tools are easier to use when prototyping algorithms or doing exploration with a subset of data. Some of the more popular options are listed in Table 1-3.

Table 1-3. Other data processing languages and tools

Name	URL	Description
R	*http://r-project.org/*	An open source language for statistical analysis and graphing of data that is popular with statisticians, economists, etc. It's not a distributed system, so the data sizes it can handle are limited. There are efforts to integrate R with Hadoop.
Matlab	*http://www.mathworks.com/products/matlab/index.html*	A commercial system for data analysis and numerical methods that is popular with engineers and scientists.
Octave	*http://www.gnu.org/software/octave/*	An open source clone of MatLab.
Mathematica	*http://www.wolfram.com/mathematica/*	A commercial data analysis, symbolic manipulation, and numerical methods system that is also popular with scientists and engineers.
SciPy, NumPy	*http://scipy.org*	Extensive software package for scientific programming in Python, which is widely used by data scientists.

Java Versus Hive: The Word Count Algorithm

If you are not a Java programmer, you can skip to the next section.

If you are a Java programmer, you might be reading this book because you'll need to support the Hive users in your organization. You might be skeptical about using Hive for your own work. If so, consider the following example that implements the Word

Count algorithm we discussed above, first using the Java MapReduce API and then using Hive.

It's very common to use Word Count as the first Java MapReduce program that people write, because the algorithm is simple to understand, so you can focus on the API. Hence, it has become the "Hello World" of the Hadoop world.

The following Java implementation is included in the Apache Hadoop distribution.[8] If you don't know Java (and you're still reading this section), don't worry, we're only showing you the code for the size comparison:

```
package org.myorg;

import java.io.IOException;
import java.util.*;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.conf.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;

public class WordCount {

 public static class Map extends Mapper<LongWritable, Text, Text, IntWritable> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
      throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
 }

 public static class Reduce extends Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterable<IntWritable> values, Context context)
      throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        context.write(key, new IntWritable(sum));
    }
```

8. Apache Hadoop word count: *http://wiki.apache.org/hadoop/WordCount*.

```
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();

    Job job = new Job(conf, "wordcount");

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    job.setMapperClass(Map.class);
    job.setReducerClass(Reduce.class);

    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.waitForCompletion(true);
  }

}
```

That was 63 lines of Java code. We won't explain the API details.[9] Here is the *same* calculation written in HiveQL, which is just 8 lines of code, and does not require compilation nor the creation of a "JAR" (Java ARchive) file:

```
CREATE TABLE docs (line STRING);

LOAD DATA INPATH 'docs' OVERWRITE INTO TABLE docs;

CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
  (SELECT explode(split(line, '\s')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
```

We'll explain all this HiveQL syntax later on.

9. See *Hadoop: The Definitive Guide* by Tom White for the details.

In both examples, the files were tokenized into words using the simplest possible approach; splitting on whitespace boundaries. This approach doesn't properly handle punctuation, it doesn't recognize that singular and plural forms of words are the same word, etc. However, it's good enough for our purposes here.[10]

The virtue of the Java API is the ability to customize and fine-tune every detail of an algorithm implementation. However, most of the time, you just don't need that level of control and it slows you down considerably when you have to manage all those details.

If you're not a programmer, then writing Java MapReduce code is out of reach. However, if you already know SQL, learning Hive is relatively straightforward and many applications are quick and easy to implement.

What's Next

We described the important role that Hive plays in the Hadoop ecosystem. Now let's get started!

10. There is one other minor difference. The Hive query hardcodes a path to the data, while the Java code takes the path as an argument. In Chapter 2, we'll learn how to use Hive *variables* in scripts to avoid hardcoding such details.