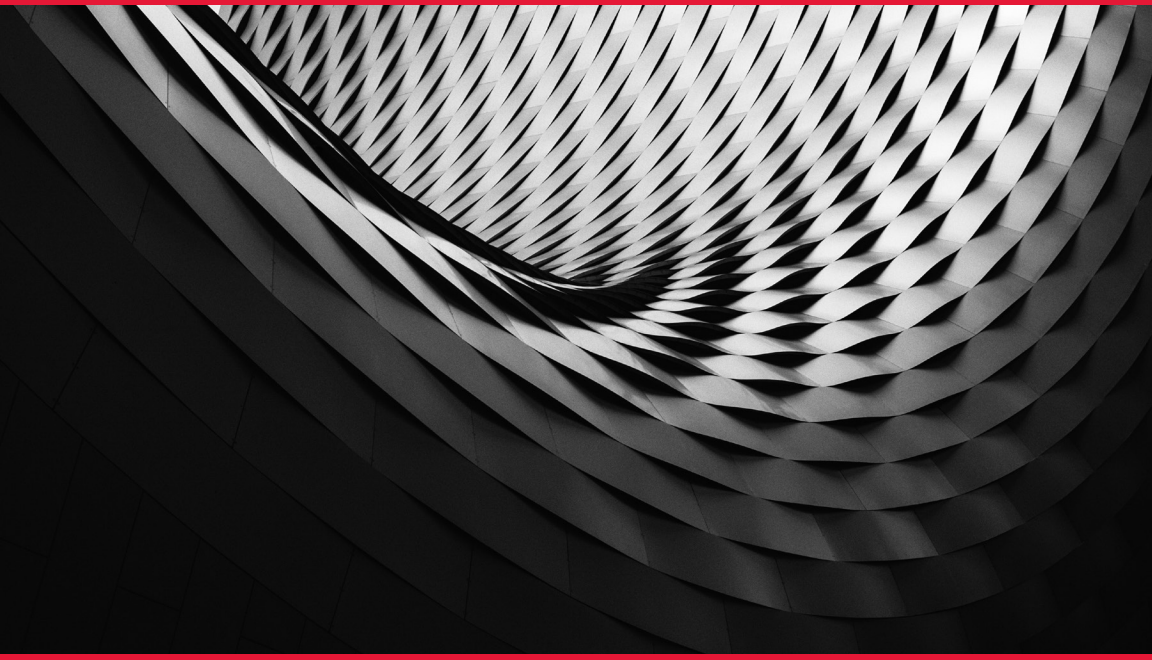
O'REILLY®

Hadoop: What You Need to Know

Hadoop Basics for the Enterprise Decision Maker



Donald Miner

Strata+ Hadoop WORLD

Make Data Work

strataconf.com

Presented by O'Reilly and Cloudera, Strata + Hadoop World is where cutting-edge data science and new business fundamentals intersect—and merge.

- Learn business applications of data technologies
- Develop new skills through trainings and in-depth tutorials
- Connect with an international community of thousands who work with data

Job # 15420

Hadoop: What You Need to Know

Hadoop Basics for the Enterprise Decision Maker

Donald Miner

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Hadoop: What You Need to Know
by Donald Miner

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*.

Editor: Marie Beaugureau
Production Editor: Kristen Brown
Proofreader: O'Reilly Production Services
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2016: First Edition

Revision History for the First Edition
2016-03-04: First Release

978-1-491-93730-3

[LSI]

For Griffin

Table of Contents

Hadoop: What You Need to Know

This report is written with the enterprise decision maker in mind. The goal is to give decision makers a crash course on what Hadoop is and why it is important. Hadoop technology can be daunting at first and it represents a major shift from traditional enterprise data warehousing and data analytics. Within these pages is an overview that covers just enough to allow you to make intelligent decisions about Hadoop in your enterprise.

From it's inception in 2006 at Yahoo! as a way to improve their search platform, to becoming an open source Apache project, to adoption as a defacto standard in large enterprises across the world, Hadoop has revolutionized data processing and enterprise data warehousing. It has given birth to dozens of successful startups and many companies have well documented Hadoop success stories. With this explosive growth comes a large amount of uncertainty, hype, and confusion but the dust is starting to settle and organizations are starting to better understand when it's appropriate and not appropriate to leverage Hadoop's revolutionary approach.

As you read on, we'll go over why Hadoop exists, why it is an important technology, basics on how it works, and examples of how you should probably be using it. By the end of this report you'll understand the basics of technologies like HDFS, MapReduce, and YARN, but won't get mired in the details.

An Introduction to Hadoop and the Hadoop Ecosystem

When you hear someone talk about Hadoop, they typically don't mean only the core Apache Hadoop project, but instead are referring to Apache Hadoop technology along with an ecosystem of other projects that work with Hadoop. An analogy to this is when someone tells you they are using Linux as their operating system: they aren't just using Linux, they are using thousands of applications that run on the Linux kernel as well.

Core Apache Hadoop

Core Hadoop is a software platform and framework for distributed computing of data. Hadoop is a platform in the sense that it is a long-running system that runs and executes computing tasks. Platforms make it easier for engineers to deploy applications and analytics because they don't have to rebuild all of the infrastructure from scratch for every task. Hadoop is a framework in the sense that it provides a layer of abstraction to developers of data applications and data analytics that hides a lot of the intricacies of the system.

The core Apache Hadoop project is organized into three major components that provide a foundation for the rest of the ecosystem:

HDFS (Hadoop Distributed File System)
: A filesystem that stores data across multiple computers (i.e., in a distributed manner); it is designed to be high throughput, resilient, and scalable

YARN (Yet Another Resource Negotiator)
: A management framework for Hadoop resources; it keeps track of the CPU, RAM, and disk space being used, and tries to make sure processing runs smoothly

MapReduce
: A generalized framework for processing and analyzing data in a distributed fashion

HDFS can manage and store large amounts of data over hundreds or thousands of individual computers. However, Hadoop allows you to both store lots of data *and* process lots of data with YARN and MapReduce, which is in stark contrast to traditional storage that just

stores data (e.g., NetApp or EMC) or supercomputers that just compute things (e.g., Cray).

The Hadoop Ecosystem

The Hadoop ecosystem is a collection of tools and systems that run alongside of or on top of Hadoop. Running "alongside" Hadoop means the tool or system has a purpose outside of Hadoop, but Hadoop users can leverage it. Running "on top of" Hadoop means that the tool or system leverages core Hadoop and can't work without it. Nobody maintains an official ecosystem list, and the ecosystem is constantly changing with new tools being adopted and old tools falling out of favor.

There are several Hadoop "distributions" (like there are Linux distributions) that bundle up core technologies into one supportable platform. Vendors such as Cloudera, Hortonworks, Pivotal, and MapR all have distributions. Each vendor provides different tools and services with their distributions, and the right vendor for your company depends on your particular use case and other needs.

A typical Hadoop "stack" consists of the Hadoop platform and framework, along with a selection of ecosystem tools chosen for a particular use case, running on top of a cluster of computers (Figure 1-1).

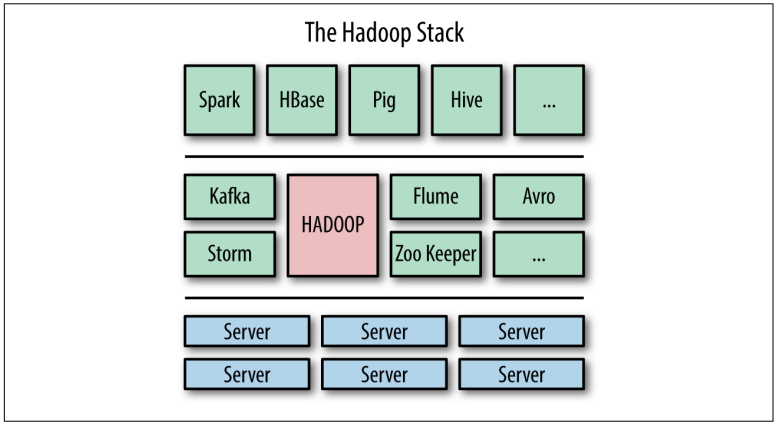


Figure 1-1. Hadoop (red) sits at the middle as the "kernel" of the Hadoop ecosystem (green). The various components that make up the ecosystem all run on a cluster of servers (blue).

Hadoop and its ecosystem represent a new way of doing things, as we'll look at next.

Hadoop Masks Being a Distributed System

Hadoop is a distributed system, which means it coordinates the usage of a cluster of multiple computational resources (referred to as servers, computers, or *nodes*) that communicate over a network. Distributed systems empower users to solve problems that cannot be solved by a single computer. A distributed system can store more data than can be stored on just one machine and process data much faster than a single machine can. However, this comes at the cost of increased complexity, because the computers in the cluster need to talk to one another, and the system needs to handle the increased chance of failure inherent in using more machines. These are some of the tradeoffs of using a distributed system. We don't use distributed systems because we want to...we use them because we have to.

Hadoop does a good job of hiding from its users that it is a distributed system by presenting a superficial view that looks very much like a single system (Figure 1-2). This makes the life of the user a whole lot easier because he or she can focus on analyzing data instead of manually coordinating different computers or manually planning for failures.

Take a look at this snippet of Hadoop MapReduce code written in Java (Example 1-1). Even if you aren't a Java programmer, I think you can still look through the code and get a general idea of what is going on. There is a point to this, I promise.

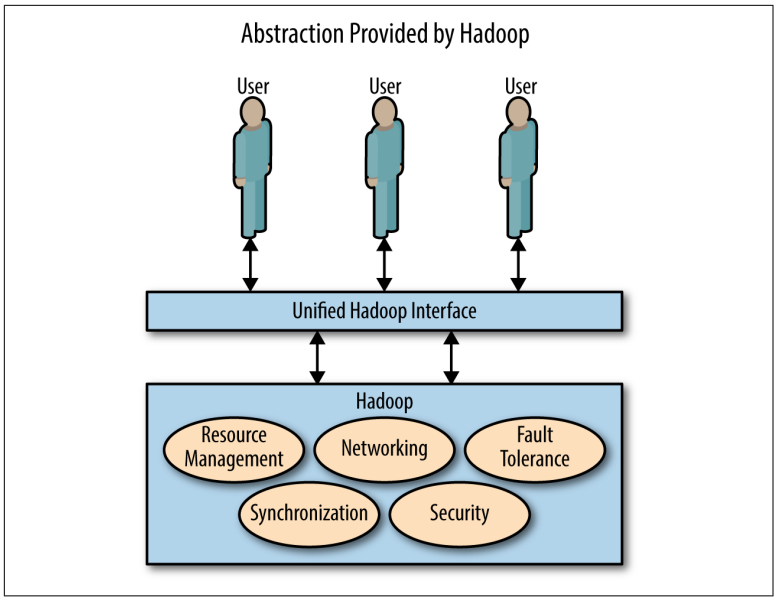


Figure 1-2. Hadoop hides the nasty details of distributed computing from users by providing a unified abstracted API on top of the distributed system underneath

Example 1-1. An example MapReduce job written in Java to count words

```
....
// This block of code defines the behavior of the map phase
public void map(Object key, Text value, Context context
) throws IOException, InterruptedException {
    // Split the line of text into words
    StringTokenizer itr = new StringTokenizer(value.toString());
    // Go through each word and send it to the reducers
    while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        // "I've seen this word once!"
        context.write(word, one);
    }
}
....
// This block of code defines the behavior of the reduce phase
public void reduce(Text key, Iterable<IntWritable> values,
Context context
) throws IOException, InterruptedException {
    int sum = 0;
    // For the word, count up the times we saw the word
    for (IntWritable val : values) {
```

```
        sum += val.get();
    }
    result.set(sum);
    // "I saw this word *result* number of times!"
    context.write(key, result);
}
....
```

This code is for word counting, the canonical example for MapReduce. MapReduce can do all sorts of fancy things, but in this relatively simple case it takes a body of text, and it will return the list of words seen in the text along with how many times each of those words was seen.

Nowhere in the code is there mention of the size of the cluster or how much data is being analyzed. The code in Example 1-1 could be run over a 10,000 node Hadoop cluster or on a laptop without any modifications. This same code could process 20 petabytes of website text or could process a single email (Figure 1-3).

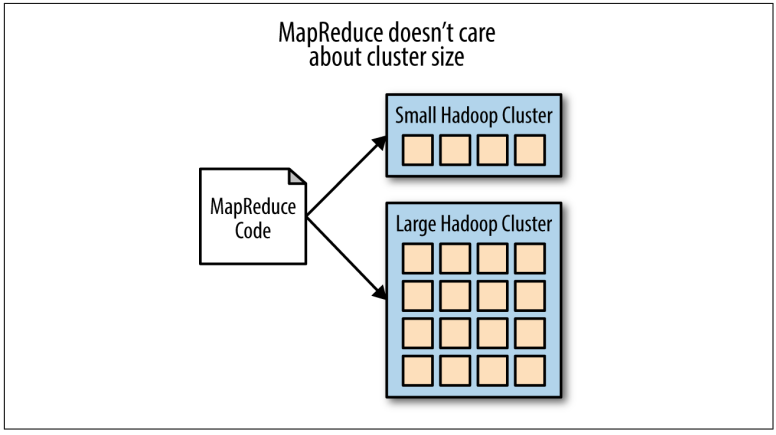


Figure 1-3. MapReduce code works the same and looks the same regardless of cluster size

This makes the code incredibly portable, which means a developer can test the MapReduce job on their workstation with a sample of data before shipping it off to the larger cluster. No modifications to the code need to be made if the nature or size of the cluster changes later down the road. Also, this abstracts away all of the complexities of a distributed system for the developer, which makes his or her life easier in several ways: there are fewer opportunities to make errors, fault tolerance is built in, there is less code to write, and so much

more—in short, a Ph.D in computer science becomes optional (I joke...mostly). The accessibility of Hadoop to the average software developer in comparison to previous distributed computing frameworks is one of the main reasons why Hadoop has taken off in terms of popularity.

Now, take a look at the series of commands in Example 1-2 that interact with HDFS, the filesystem that acts as the storage layer for Hadoop. Don't worry if they don't make much sense; I'll explain it all in a second.

Example 1-2. Some sample HDFS commands

```
[1]$ hadoop fs -put hamlet.txt datz/hamlet.txt
[2]$ hadoop fs -put macbeth.txt data/macbeth.txt
[3]$ hadoop fs -mv datz/hamlet.txt data/hamlet.txt
[4]$ hadoop fs -ls data/
-rw-r-r- 1 don don 139k 2012-01-31 23:49 /user/don/data/
caesar.txt
-rw-r-r- 1 don don 180k 2013-09-25 20:45 /user/don/data/
hamlet.txt
-rw-r-r- 1 don don 117k 2013-09-25 20:46 /user/don/data/
macbeth.txt
[5]$ hadoop fs -cat /data/hamlet.txt | head
The Tragedie of Hamlet
Actus Primus. Scoena Prima.
Enter Barnardo and Francisco two Centinels.
Barnardo. Who's there?
Fran. Nay answer me: Stand & vnfold your selfe
Bar. Long liue the King
```

What the HDFS user did here is loaded two text files into HDFS, one for Hamlet (1) and one for Macbeth (2). The user made a typo at first (1) and fixed it with a "mv" command (3) by moving the file from `datz/` to `data/`. Then, the user lists what files are in the `data/` folder (4), which includes the two text files as well as the screenplay for Julius Caesar in `caesar.txt` that was loaded earlier. Finally, the user decides to take a look at the top few lines of Hamlet, just to make sure it's actually there (5).

Just as there are abstractions for writing code for MapReduce jobs, there are abstractions when writing commands to interact with HDFS—mainly that *nowhere in HDFS commands is there information about how or where data is stored.* When a user submits a Hadoop HDFS command, there are a lot of things that happen behind the scenes that the user is not aware of. All the user sees is

the results of the command without realizing that sometimes dozens of network communications needed to happen to retrieve the result.

For example, let's say a user wants to load several new files into HDFS. Behind the scenes, HDFS is taking each of these files, splitting them up into multiple blocks, distributing the blocks over several computers, replicating each block three times, and registering where they all are. The result of this replication and distribution is that if one of the Hadoop cluster's computers fails, not only won't the data be lost, but the user won't even notice any issues. There could have been a catastrophic failure in which an entire rack of computers shut down in the middle of a series of commands and the commands still would have been completed without the user noticing and without any data loss. This is the basis for Hadoop's *fault tolerance* (meaning that Hadoop can continue running even in the face of *some* isolated failures).

Hadoop abstracts parallelism (i.e., splitting up a computational task over more than one computer) by providing a distributed platform that manages typical tasks, such as resource management (YARN), data storage (HDFS), and computation (MapReduce). Without these components, you'd have to program fault tolerance and parallelism into every MapReduce job and HDFS command and that would be really hard to do.

Hadoop Scales Out Linearly

Hadoop does a good job maintaining *linear scalability*, which means that as certain aspects of the distributed system scale, other aspects scale 1-to-1. Hadoop does this in a way that scales *out* (not *up*), which means you can add to your existing system with newer or more powerful pieces. For example, scaling up your refrigerator means you buy a larger refrigerator and trash your old one; scaling out means you buy another refrigerator to sit beside your old one.

Some examples of scalability for Hadoop applications are shown in Figure 1-4.

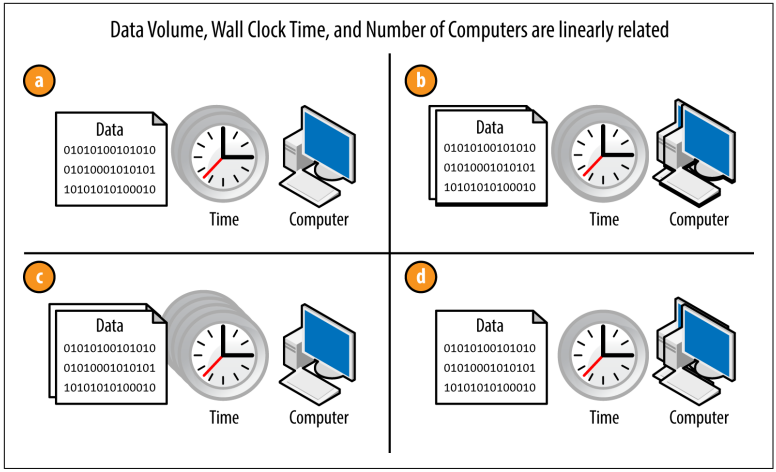


Figure 1-4. Hadoop linear scalability; by changing the amount of data or the number of computers, you can impact the amount of time you need to run a Hadoop application

Consider Figure 1-4a relative to these other setups:

- In Figure 1-4b, by doubling the amount of data and the number of computers from Figure 1-4a, we keep the amount of time the same. This rule is important if you want to keep your processing times the same as your data grows over time.
- In Figure 1-4c, by doubling the amount of data while keeping the number of computers the same, the amount of time it'll take to process this data doubles.
- Conversely from Figure 1-4c, in Figure 1-4d, by doubling the number of computers without changing the data size, the wall clock time is cut in half.

Some other more complex applications of the rules, as examples:

- If you store twice as much data and want to process data twice as fast, you need four times as many computers.
- If processing a month's worth of data takes an hour, processing a year's worth should take about twelve hours.
- If you turn off half of your cluster, you can store half as much data and processing will take twice as long.

These same rules apply to storage of data in HDFS. Doubling the number of computers means you can store twice as much data.

In Hadoop, the number of nodes, the amount of storage, and job runtime are intertwined in linear relationships. Linear relationships in scalability are important because they allow you to make accurate predictions of what you will need in the future and know that you won't blow the budget when the project gets larger. They also let you add computers to your cluster over time without having to figure out what to do with your old systems.

Recent discussions I've had with people involved in the Big Data team at Spotify—a popular music streaming service—provide a good example of this. Spotify has been able to make incremental additions to grow their main Hadoop cluster every year by predicting how much data they'll have next year. About three months before their cluster capacity will run out, they do some simple math and figure out how many nodes they will need to purchase to keep up with demand. So far they've done a pretty good job predicting the requirements ahead of time to avoid being surprised, and the simplicity of the math makes it easy to do.

Hadoop Runs on Commodity Hardware

You may have heard that Hadoop runs on commodity hardware, which is one of the major reasons why Hadoop is so groundbreaking and easy to get started with. Hadoop was originally built at Yahoo! to work on existing hardware they had that they could acquire easily. However, for today's Hadoop, commodity hardware may not be exactly what you think at first.

In Hadoop lingo, *commodity hardware* means that the hardware you build your cluster out of is nonproprietary and nonspecialized: plain old CPU, RAM, hard drives, and network. These are just Linux (typically) computers that can run the operating system, Java, and other unmodified tool sets that you can get from any of the large hardware vendors that sell you your web servers. That is, the computers are general purpose and don't need any sort of specific technology tied to Hadoop.

This is really neat because it allows significant flexibility in what hardware you use. You can buy from any number of vendors that are competing on performance and price, you can repurpose some

of your existing hardware, you can run it on your laptop computer, and never are you locked into a particular proprietary platform to get the job done. Another benefit is if you ever decide to stop using Hadoop later on down the road, you could easily resell the hardware because it isn't tailored to you, or you can repurpose it for other applications.

However, don't be fooled into thinking that commodity means inexpensive or consumer-grade. Top-of-the-line Hadoop clusters these days run serious hardware specifically customized to optimally run Hadoop workloads. One of the major differences between a typical Hadoop node's hardware and other server hardware is that there are more hard drives in a single chassis—typically between 12 and 24—in order to increase data throughput through parallelism. Clusters that use systems like HBase or have a high number of cores will also need a lot more RAM than your typical computer will have.

So, although "commodity" connotes inexpensive and easy to acquire, typical production Hadoop cluster hardware is just nonproprietary and nonspecialized, not necessarily generic and inexpensive.

Don't get too scared...the nice thing about Hadoop is that it'll work great on high-end hardware or low-end hardware, but be aware that you get what you pay for. That said, paying an unnecessary premium for the best of the best is often not as effective as spending the same amount of money to simply acquire more computers.

Hadoop Handles Unstructured Data

If you take another look at how we processed the text in Example 1-1, we used Java code. The possibilities for analysis of that text are endless because you can simply write Java to process the data in place. This is a fundamental difference from relational databases, where you need to first transform your data into a series of predictable tables with columns that have clearly defined data types (also known as performing extract, transform, load, or ETL). For that reason, the relational model is a paradigm that just doesn't handle unstructured data well.

What this means for you is that you can *analyze data you couldn't analyze before* using relational databases, because they struggle with unstructured data. Some examples of unstructured data that organi-

zations are trying to parse today range from scanned PDFs of paper documents, images, audio files, and videos, among other things. This is a big deal! Unstructured data is some of the hardest data to process but it also can be some of the most valuable, and Hadoop allows you to extract value from it.

However, it's important to know what you are getting into. Processing unstructured data with a programming language and a distributed computing framework like MapReduce isn't as easy as using SQL to query a relational database. This is perhaps the highest "cost" of Hadoop—it requires more emphasis on code for more tasks. For organizations considering Hadoop, this needs to be clear, as a different set of human resources is required to work with the technology in comparison to relational database projects. The important thing to note here is that with Hadoop we *can* process unstructured data (when we couldn't before), but that doesn't mean that it's easy.

Keep in mind that Hadoop isn't only used to handle unstructured data. Plenty of people use Hadoop to process very structured data (e.g., CSV files) because they are leveraging the other reasons why Hadoop is awesome, such as its ability to handle scale in terms of processing power or storage while being on commodity hardware. Hadoop can also be useful in processing fields in structured data that contain freeform text, email content, customer feedback, messages, and other non-numerical pieces of data.

In Hadoop You Load Data First and Ask Questions Later

Schema-on-read is a term popularized by Hadoop and other NoSQL systems. It means that the nature of the data (the schema) is inferred on-the-fly while the data is being read off the filesystem for analysis, instead of when the data is being loaded into the system for storage. This is in contrast to *schema-on-write*, which means that the schema is encoded when the data is stored in the analytic platform (Figure 1-5). Relational databases are schema-on-write because you have to tell the database the nature of the data in order to store it. In ETL (the process of bringing data into a relational database), the important letter in this acronym is T: transforming data from its original form into a form the database can understand and store. Hadoop, on the other hand, is schema-on-read because the MapReduce job makes sense of the data as the data is being read.

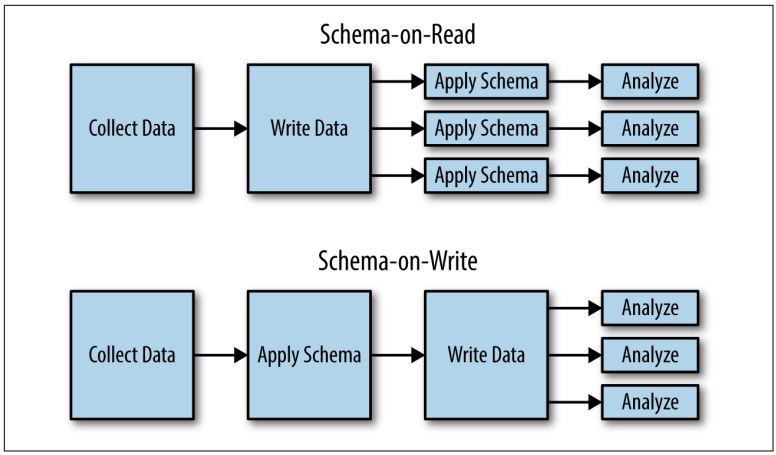


Figure 1-5. Schema-on-read differs from schema-on-write by writing data to the data store before interpreting the schema or transforming it in any way. The upside is the interpretation of the nature of data is pushed until later, but the downside is that it needs to be interpreted every time the data is analyzed.

The Benefits of Schema-on-Read

Schema-on-read seems like a pretty basic idea but it is a drastic departure from the "right way" of doing things in relational databases. With it comes one fundamental and drastic benefit: the ability to easily load and keep the data in its original and raw form, which is the unmodified bytes coming out of your applications and systems. Why would you want to keep this around? At first glance you might not take data sources like these too seriously, but the power of keeping the original data around cannot be ignored as long as you can handle the work associated with processing it.

Ability to explore data sooner

The first benefit in dealing with raw data is it solves a fundamental chicken-and-egg problem of data processing: *You don't know what to do with your data until you do something with your data.* To break the chain, you need to explore your data...but you can't do effective data exploration without putting your data in a place where it can be processed. Hadoop and schema-on-read allow you to load data into Hadoop's HDFS in its original form without much thought about how you might process it. Then, with MapReduce, you can write code to break apart the data and analyze it on the fly. In a SQL data-

base, you need to think about the kind of processing you will do, which will motivate how you write your schemas and set up your ETL processes. Hadoop allows you to skip that step entirely and get working on your data a lot sooner.

Schema and ETL flexibility

The next benefit you will notice is the cost of changing your mind is minimal: *once you do some data exploration and figure out how you might extract value from the data, you may decide to go in a different direction than originally anticipated*. If you had been following a schema-on-write paradigm and storing this data in a SQL database, this may mean going back to the drawing board and coming up with new ETL processes and new schemas—or even worse, trying to shoehorn the new processes into the old. Then, you would have faced the arduous process of reingesting all of the data back into the system. With schema-on-read in Hadoop, the raw data can stay the same but you just decide to cook it in a different way by modifying your MapReduce jobs. No data rewrite is necessary and no changes need to be made to existing processes that are doing their job well. A fear of commitment in data processing is important and Hadoop lets you avoid committing to one way of doing things until much later in the project, when you will likely know more about your problems or use cases.

Raw data retains all potential value

The other benefit is that *you can change your mind about how important certain aspects of your data are*. Over time, the value of different pieces of your data will change. If you only process the pieces of data that you care about right now, you will be punished later. Unfortunately, having the foresight into what will be important to you later is impossible, so your best bet is to keep the raw data around...just in case. I recommend that you become a data hoarder because you never know when that obscure piece of data might be really useful. One of the best examples of this is forensic analysis in cybersecurity, during which a data analyst researches a threat or breach after the fact. Before and during an attack, it's not clear what data will be useful. The only recourse is to store as much data as possible and organize it well so that when something does happen it can be investigated efficiently.

Another reason raw data is great: everyone loves to think that their code is perfect, but everyone also knows this is not the case. ETL processes can end up being massive and complicated chunks of code that—like everything else—are susceptible to bugs. Being able to keep your raw data around allows you to *go back and retroactively reprocess data after a bug has been fixed*. Meanwhile, downtime might be necessary in a schema-on-write system because data has to go back and be fixed and rewritten. This benefit isn't just about being resilient to mistakes—it also lets you be more agile by allowing you to take on more risk in the early stages of processing in order to move on to the later and more interesting stages more quickly.

Ability to treat the data however you want

Finally, the best part: *schema-on-read and using raw data isn't mandatory*! Nothing is stopping you from having elaborate ETL pipelines feeding your Hadoop cluster and reaping all of the benefits of schema on write. The decision can be made on a case by case basis and there's always an option to do both. On the flip side, a relational database will have a hard time doing schema-on-read and you really are locked into one way of doing things.

Hadoop is Open Source

The pros and cons of free and open source software is still a contentious issue. My intent isn't to distract, but mentioning some of the benefits of Hadoop being free and open source is important.

First, it's free. Free isn't always about cost; it is more about barrier to entry. Not having to talk to a sales executive from a large software vendor and have consultants come in to set you up in order to run your first pilot is rather appealing if you have enough skill in-house. The time and cost it takes to run an exploratory pilot is significantly reduced just because there isn't someone trying to protect the proprietary nature of the code. Hadoop and its open source documentation can be downloaded off of the Apache website or from the websites of one of many vendors that support Hadoop. No questions asked.

The next thing that the no-cost nature of Hadoop provides is ease of worldwide adoption. In this day and age, most of the technological *de facto* standards are open source. There are many reasons why

that's the case, but one of them is certainly that there is a lower barrier to widespread adoption for open-source projects—and widespread adoption is a good thing. If more people are doing what you are doing, it's easier to find people who know how to do it and it's easier to learn from others both online and in your local community. Hadoop is no exception here either. Hadoop is now being taught in universities, other companies local to yours are probably using it, there's a proliferation of Hadoop user groups, and there is no shortage of documentation or discussion online.

As with the other reasons in this list of Hadoop benefits, you can have it both ways: if you want to take advantage of some of the benefits of commercial software, there are several vendors that support and augment Hadoop to provide you with additional value or support if you want to pay for it. Hadoop being open source gives you the choice to do it yourself or do it with help. You aren't locked in one way or another.

The Hadoop Distributed File System Stores Data in a Distributed, Scalable, Fault-Tolerant Manner

The Hadoop Distributed File System (HDFS) gives you a way to store a lot of data in a distributed fashion. It works with the other components of Hadoop to serve up data files to systems and frameworks. Currently, some clusters are in the hundreds of petabytes of storage (a petabyte is a thousand terabytes or a million gigabytes).

The NameNode Runs the Show and the DataNodes Do All the Work

HDFS is implemented as a "master and slave" architecture that is made up of a NameNode (the master) and one or more data nodes (the slaves). The data on each data node is managed by a program called a DataNode, and there is one DataNode program for each data node. There are other services like the Checkpoint node and NameNode standbys, but the two important players for understanding how HDFS works are the NameNode and the DataNode.

The NameNode does three major things: it knows where data is, it tells clients where to send data, and tells clients where to retrieve data from. A *client* is any program that connects to HDFS to interact with it in some way. Client code is embedded in MapReduce, custom applications, HBase, and anything else that uses HDFS. Behind the scenes, DataNodes do the heavy lifting of transferring data to clients and storing data on disk.

Transferring data to and from HDFS is a two-step process:

1. The client connects to the NameNode and asks which DataNode it can get the data from or where it should send the data.
2. The client then connects to the DataNode the NameNode indicated and receives or sends the data directly to the DataNode, without the NameNode's involvement.

This is done in a fault-tolerant and scalable way. An example of the communication behind the scenes for a HDFS command is shown in Figure 1-6.

There are all kinds of things going on behind the scenes, but the major takeaway is that the NameNode is solely a coordinator. Large volumes of data are not being sent back and forth to the single NameNode; instead, the data is spread out across the several DataNodes. This approach makes throughput scalable to the point of the network bandwidth, not just the bandwidth of a single node.

The NameNode also keeps track of the number of copies of your data (covered in more detail in "HDFS Stores Files in Three Places" on page 18) and will tell DataNodes to make more copies if needed.

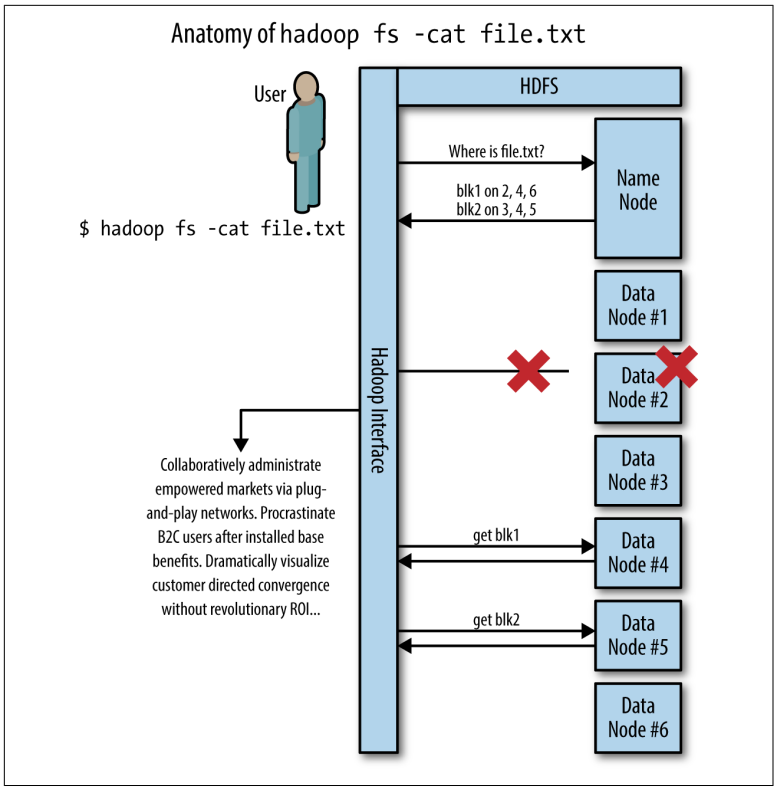


Figure 1-6. Anatomy of an HDFS command, `hadoop fs -cat file.txt`. The HDFS file "file.txt" is split into two blocks due to its size, blk1 and blk2. These blocks are stored on a number of nodes across the cluster. DataNode #2 is having some sort of catastrophic failure and the operation of getting blk1 fails. Hadoop then tries to get blk1 from DataNode #4 instead and succeeds.

HDFS Stores Files in Three Places

By default, HDFS stores three copies of your files scattered across the cluster somewhat randomly. These are not backup copies; they are first-class files and can all be accessed by clients. As a user you'll never notice that there are three copies. Even if some failure occurs and there are temporarily only two copies, you'll never know because it is all handled behind the scenes by the NameNode and the DataNodes (Figure 1-6).

When the Google File System (GFS) paper was released, which HDFS was based on, most people were really confused about some of the choices made in the GFS, but over time they started making sense. Storing three copies of files is one of the curiosities that got attention. Why did Google decide to do that?

First, storing three copies of the data is really inefficient: 33% storage efficiency. For each three-terabyte hard drive you add to your system, you're only able to store an additional one terabyte of data. When we are talking about tens of petabytes of data, that can get really expensive. There are numerous distributed storage approaches that store data in a redundant manner over multiple computers that achieve efficiencies upwards of 80%.

However, upon further examination there are benefits to the replication that outweigh the costs. First let's look at network bandwidth and a concept called *data locality*. Data locality is the practice of processing data where it resides without transferring it over a network. We'll talk about how important data locality is later on, but for now just be aware that it's important.

Now, let's contrast the way data is broken up and stored in GFS/HDFS versus traditional distributed storage systems. Traditional storage systems use data striping or parity bits to achieve high efficiency. With striping, a file is split up into alternating little pieces of data and moved onto separate storage devices. The chunking approach taken by HDFS, on the other hand, doesn't split up the data in a fine-grained manner—instead, it chunks the data into contiguous blocks.

Traditional storage systems that use striping may be good at retrieving a single file with a high number of IOPS (input and output operations per second), but are fundamentally flawed for large-scale data analysis. Here is the problem: if you want to read a file that has been split out by striping and analyze the contents, you need to rematerialize the file from all of the different locations it resides in within the cluster, which requires network usage and makes data locality impossible. On the other hand, if you store the file in contiguous chunks but instead store three copies of it (how GFS/HDFS does it), when a client is ready to analyze the file, the file doesn't need to be rematerialized; the processing can work on that one contiguous chunk in isolation. The client can analyze that file on the same computer on which the data resides without using the network to trans-

fer data (data locality). Data locality dramatically reduces the strain on the network, and thus allows for more data processing–in other words, greater scalability. When MapReduce is processing thousands of files at once, this scalability greatly outweighs the price of storage density and high IOPS.

If you are set on storing a file as a single sequential chunk instead of in stripes (e.g., in HDFS), you need to store complete copies somewhere else in case that chunk gets lost in a failure. But why three copies? How about two? Or four? Three copies is empirically a sweet spot in two ways: fault tolerance and performance. The fault tolerance aspect is obvious: GFS/HDFS can lose two computers in short order and not only will the data not be lost, the NameNode will be able to command DataNode processes to replicate the remaining copy to get back up to three copies. The benefits to performance may not be as intuitive: with more copies, each piece of data is more available to clients since they can access any of the three copies. This smooths out hot spots (i.e., computers that are overwhelmed because they host files that are getting used a lot) by spreading the traffic for a single file over multiple computers. This is important when running multiple jobs over the same data at the same time. To answer the original question: two replicas is a bit too risky for most people and you may notice performance degradation; four replicas is really conservative and doesn't improve performance enough to make it worthwhile.

Files in HDFS Can't Be Edited

HDFS does a good job of storing large amounts of data over large numbers of computers, but in order to achieve its throughput potential and scalability, some functionality had to be sacrificed—namely the ability to edit files. Most first-time users of HDFS are disappointed that it can't be used as a normal storage device (e.g., network-attached storage or a USB thumb drive), which would allow them to use first-class applications (even something like Microsoft Word) on data in HDFS. Instead, you are limited by the Hadoop commands and a programming interface. By sacrificing this feature, the designers of HDFS kept HDFS simple and good at what it needs to do.

Once you write a file, that's it: the file will have that content until it gets deleted. This makes things easier for HDFS in a number of ways, but mostly it means that HDFS doesn't need to make

synchronized changes, which is pretty hard to do in a distributed system.

For all intents and purposes, a file in HDFS is the atomic building block of a data set, not the records inside the file. The implication of this is that you have to design your data ingest flow and your applications in a way that doesn't edit files, but instead adds files to a folder. If the use case really requires editing records, check out the BigTable-based key/value stores HBase and Accumulo.

The NameNode Can Be HDFS's Achilles' Heel

With great power comes great responsibility. The NameNode is really important because it knows where everything is. Unfortunately, given its importance there are a number of ways it can fall short that you need to be aware of.

SPOF—it's fun to say, but unfortunately it has been one of the major criticisms of Hadoop. SPOF stands for *Single Point Of Failure* and is a very bad thing to have in a distributed system. In traditional HDFS, the NameNode is a SPOF because if it fails or becomes unavailable, HDFS as a whole becomes unavailable. This does *not* mean that the data stored within HDFS has been lost, merely that it is unavailable for retrieval because it's impossible to find out where it is. It's like losing your phone book but phone numbers still work.

There are ways of reducing the chances of the NameNode becoming unavailable (referred to as implementing *NameNode High Availability*, or *NameNode HA*). However, the downside of doing that is added complexity, and when things break they are harder to fix. Eric Baldeschweiler, one of the fathers of Hadoop at Yahoo! and a founder of Hortonworks (a Hadoop vendor), told me that sometimes it doesn't make sense to worry about configuring the NameNode for high availability, because using the regular NameNode configuration is simple to implement and easy to fix. It may break more often, but it takes far less time to fix when it does break. With enough metrics and monitoring, and a staff that has fixed NameNodes before, a recovery could take an order of minutes. It's just a tradeoff you have to consider, but by no means think that NameNode HA is an absolute requirement for production clusters. However, significant engineering efforts are being applied to NameNode HA, and it gets easier and more reliable with every new Hadoop release.

YARN Allocates Cluster Resources for Hadoop

YARN became an official part of Hadoop in early 2012 and was the major new feature in Hadoop's new 2.x release. YARN fills an important gap in Hadoop and has helped it expand its usefulness into new areas. This gap was in resource negotiation: how to divvy up cluster resources (such as compute and memory) over a pool of computers.

Before YARN, most resource negotiation was handled at the individual computer's system level by the individual operating system. This has a number of pitfalls because the operating system does not understand the grand scheme of things at the cluster level and therefore is unable to make the best decisions about which applications and tasks (e.g., MapReduce or Storm) get which resources. YARN fills this gap by spreading out tasks and workloads more intelligently over the cluster and telling each individual computer what it should be running and how many resources should be given to it.

Getting YARN up and running is part of the standard install of Hadoop clusters and is as synonymous with "running Hadoop" as HDFS is, in the sense that pretty much every Hadoop cluster is running YARN these days.

Developers Don't Need to Know Much About YARN

The best thing about YARN is that YARN does everything in the background and doesn't require much interaction with Hadoop developers on a day-to-day basis to accomplish tasks. For the most part, developers work with the MapReduce APIs and other frameworks' APIs—the resource negotiation is handled behind the scenes. The framework itself is interacting with YARN and asking for resources, and then utilizing them without requiring developers to do anything.

Usually the most time spent with YARN is when there is a problem of some sort (such as applications not getting enough resources or the cluster not being utilized enough). Fixing these problems falls within the responsibilities of the system administrator, not the developer.

YARN really pays off in the long run by making developers' lives easier (which improves productivity) and by reducing the barrier of entry to Hadoop.

MapReduce is a Framework for Analyzing Data

MapReduce is a generalized framework for analyzing data stored in HDFS over the computers in a cluster. It allows the analytic developer to write code that is completely ignorant of the nature of the massive distributed system underneath it. This is incredibly useful because analytic developers just want to analyze data—they don't want to worry about things like network protocols and fault tolerance every time they run a job.

MapReduce itself is a paradigm for distributed computing described in a 2004 paper by engineers at Google. The authors of the paper described a general-purpose method to analyze large amounts of data in a distributed fashion on commodity hardware, in a way that masks a lot of the complexities of distributed systems.

Hadoop's MapReduce is just one example implementation of the MapReduce concept (the one that was originally implemented by Yahoo!). The CouchDB, MongoDB, and Riak NoSQL databases also have native MapReduce implementations for querying data that have nothing to do with Hadoop's MapReduce.

Map and Reduce Use the Notion of Shared Nothing

Unlike the other names you may have come across in Hadoop—e.g., Pig, Hive, ZooKeeper, and so on—MapReduce is actually describing a process: it stands for two major processing phases called "map" and "reduce" (surprise!). In a MapReduce job, the analytic developer writes custom code for the map phase and the reduce phase. Then, the Hadoop system takes that code and applies it to the data in a parallel and scalable way. The map phase performs record-by-record alterations by taking a close look at each record, but only that record. The reduce phase performs the higher-level groupings, counting, and summarizations on groups of records. The ability to customize code for the map and reduce phases is what gives Hadoop the flexibility to answer the questions that couldn't be answered before. With both map and reduce, you can do just about anything with your data.

This section is going to dive into the technical weeds a bit deeper than the other portions of this report, but it describes a couple of really important points about what makes Hadoop and MapReduce different. Try not to get too held up if you don't fully comprehend

the technical details, just focus on understanding that MapReduce has advantage in linear scalability.

In programming, *map* typically means applying some function (such as absolute value or uppercase) to every element in a list, and then returning that series of outcomes as a list. See some examples of map outside of a Hadoop context in Example 1-3.

Example 1-3. Two examples of map in Python

```
# apply the "abs" function (absolute value) to the list of numbers
map(abs, [1, -3, -5, 0, 4])
# returns [1, 3, 5, 0, 4]

# apply the uppercase function to the list of strings
map(lambda s: s.upper(), ['map', 'AND', 'Reduce'])
# returns ['MAP', 'AND', 'REDUCE']
```

There is something really interesting to note about map: the order in which map functions are applied doesn't really matter. In the first example above, we could have applied the abs function to the third element in the list, then the first, then the second, then the fifth, then the fourth and the result would have been the same as if it was done in some other order.

Now, suppose you wanted to apply a map function to a list of a trillion integers (which on a single computer would take a long time, so it would help to split it across several computers). Since the order in which the items get applied doesn't matter, you avoid one of the largest challenges in distributed systems: getting different computers to agree on when things are done and in what order. This concept in distributed systems is called *shared nothing*, which basically means that each node doing work is independent of the others and can do its job without sharing anything with the other data nodes.

The result is a very scalable system because you can add more data nodes without adding a lot of overhead. Shared nothing, along with data locality, provides a foundation for theoretically limitless linear scalability and is why MapReduce works so well.

So, the map phase of a MapReduce job is a shared nothing approach to processing data stored in HDFS. A MapReduce developer writes a piece of custom code that looks at one record at a time (the equivalent to abs or the lambda expression in Example 1-3) and MapReduce figures out how to run that code on every record in the data

set. But unlike the map in Python from Example 1-3, MapReduce's map is expected to output a *key/value pair*—essentially a set of linked data with the *key* being a unique identifier and the *value* being the data itself. Behind the scenes, the MapReduce framework will group together the pairs with the same key and send all of the values with the same key to the same reduce function.

Reduce in MapReduce takes results from the map phase, groups them together by key, and performs aggregation operations such as summations, counts, and averages over the key groups. The term *reduce*, like *map*, is a programming term that is used in several languages. The traditional reduce operation takes a function as a parameter (like map). But unlike map, the function takes two parameters instead of one. It applies the parameter function to the first two elements of the list, takes the result, then reapplies the function to that outcome with the next element—then reapplies the function to that outcome with the next element, and so on. See some examples outside of a Hadoop context in Example 1-4.

Example 1-4. Examples of reduce in Python

```
# sum the numbers using a function that adds two numbers
reduce(lambda x, y: x + y, [1, 2, 3, 3])
# returns 9

# find the largest number by using the +max+ function
reduce(max, [1, 2, 3, 3])
# returns 3

# concatenate the list of strings and add a space between each token
reduce(lambda x, y: x + ' ' + y, ['map', 'and', 'reduce'])
# returns 'map and reduce'
```

In Hadoop, reduce is a bit more general in how it does its job than its traditional counterpart seen in the Python examples, but the concept is the same: take a list of items and run some sort of processing over that list as a whole. Sometimes that means distilling down to a summation (like the max example), or just reformatting the list in a new form that isn't really reducing the size of the output (like the string concatenation example).

To summarize how MapReduce works, here are the map and reduce functions explained without code. This summary uses the canonical example of MapReduce: "word count" (which you saw Java code for

earlier in Example 1-1). This program goes through a corpus of text and counts the words seen in that text:

Map

- Take a line of text as input.
- Split up the text by whitespace (e.g., spaces, tabs, or newlines).
- Normalize each word (i.e., remove capitalization, punctuation, and if you are getting fancy spell checking or Porter stemming).
- Output each word as a key and the number one as a value, which means "we saw the word 'Hadoop' once."

Reduce

- Take a key and a bunch of ones as input.
- Sum the ones to get the count for that word.
- Output the sum, which means "we saw the word 'Hadoop' X-number of times."

There are a ton of other things going on behind the scenes that Hadoop is handling for you. Most notably, I skipped over how MapReduce gets data from the map functions to the reduce functions (which is called the "shuffle and sort"), but this is meant as a high-level introduction and you can find this information in either *Hadoop: The Definitive Guide* (Tom White, O'Reilly) or *MapReduce Design Patterns* (Donald Miner and Adam Shook, O'Reilly) if you think you'd like to know more. The main takeaway here is that MapReduce uses a shared nothing approach to achieve extreme *scalability in computation.* The tradeoff is that the analytic developer has to write their job in terms of map and reduce and can't just write a general-purpose program. MapReduce knows how to parallelize map and reduce functions, but not anything else.

Data Locality is One of the Main Reasons Hadoop is Scalable

Shared nothing lets us scale compute well, but once you fix one bottleneck, you move on to the next one. In this case, the next bottleneck is the scalability of the network fabric in the cluster.

In many traditional data processing systems, storage (where the data is at rest) and compute (where the data gets processed) have been kept separate. In these systems, data would be moved over a very fast network to computers that would then process it. Moving data over a network is really expensive in terms of time, in comparison to, say...not going over the network at all. The alternative is to process the data in place, where it's stored (which is referred to as *data locality*; see Figure 1-7).

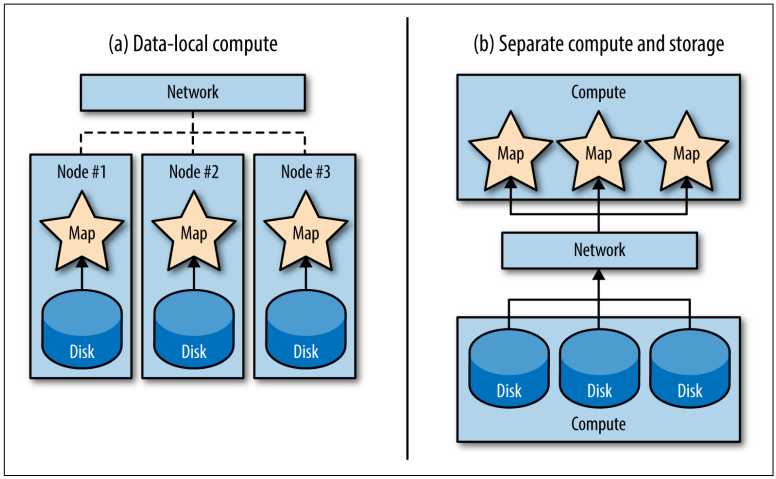


Figure 1-7. Data locality is a radical shift from separating storage and compute, which at one point had been a popular way of handling large-scale data systems. (a) In data-local compute, the map tasks are pulling data from the local disks and not using the network at all. (b) In separate compute and storage, all of the data passes through the network when you wish to process it.

Data locality seems like a really straightforward idea, but the concept of not moving data over the network and processing it in place is actually pretty novel. Hadoop was one of the first general-purpose and widely available systems that took that approach. It turns out that shipping the instructions for computation (i.e., a compiled program containing code for map and reduce functions, which is relatively small) over the network is much faster than shipping a petabyte of data to where the program is.

Data locality fundamentally removes the network as a bottleneck for linear scalability. Little programs are moved around to an arbitrarily large number of computers that each store their own data—and

data, for the most part, doesn't need to leave the computer it rests on. Together with shared nothing, there isn't much left to prevent Hadoop from scaling to thousands or tens of thousands of nodes.

This is where we come full circle back to HDFS. HDFS is built to support MapReduce, and without some key design decisions MapReduce wouldn't be able to achieve data locality. Namely, storing files as large contiguous chunks allows map tasks to process data from a single disk instead of having to pull from several disks or computers over the network. Also, because HDFS creates three replicas, it drastically increases the chances that a node will be available to host local computation, even if one or two of the replica nodes are busy with other tasks.

There Are Alternatives to MapReduce

MapReduce in many cases is not abstract enough for its users, which you may find funny since that is its entire point: to abstract away a number of the details in distributed computation. However, computer scientists love building abstractions on abstractions and were able to make MapReduce a lot easier to use with frameworks like Apache Crunch, Cascading, and Scalding (data pipeline-based computation tools that abstract the usage of map and reduce in MapReduce), and higher-level languages like Apache Pig (a dataflow language) and Apache Hive (a SQL-like language). In many cases, MapReduce can be considered the assembly language of MapReduce-based computation.

As with everything else, this abstraction comes at a cost, and masking too many details can be dangerous. In the worst case, code can get convoluted and performance may be affected. Sometimes using MapReduce code is the best way to achieve the needed control and power. However, be careful of the power user that wants to always do things in MapReduce...there is a time and place for the higher-level abstractions.

MapReduce is "Slow"

One recurring complaint with MapReduce is that it is slow. Well, what does slow mean?

First, MapReduce has a decent amount of spin-up time, meaning that it has a fixed time cost to do anything and that fixed cost is nontrivial. A MapReduce job that does very little other except load and

write data might take upward of 30 to 45 seconds to execute. This latency makes it hard to do any sort of ad hoc or iterative analysis because waiting 45 seconds for every iteration becomes taxing on the developer. It also prevents the development of on-demand analytics that run in response to user actions.

Second, MapReduce writes and reads to disk a lot (upwards of seven times per MapReduce job) and interacting with disk is one of the most expensive operations a computer can do. Most of the writing to disk helps with fault tolerance at scale and helps avoid having to restart a MapReduce job if just a single computer dies (which is a downside of some of the more aggressive parallel computational approaches).

MapReduce works best when its jobs are scheduled to run periodically and have plenty of time, also known as running in *batch*. MapReduce is really good at batch because it has enormous potential for throughput and aggregating large amounts of data in a linearly scalable way. For example, it is common to once a day run collaborative filtering recommendation engines, build reports on logs, and train models. These use cases are appropriate for batch because they are over large amounts of data and the results only need to be updated now and then. But, batch is not the only thing people want to do.

Nowadays, there's a lot of interest in real-time *stream* processing, which is fundamentally different from batch. One of the early free and open source solutions to real-time stream processing was Apache Storm, which was originally an internal system at Twitter. Other technologies that are used to do streaming include Spark Streaming (a microbatch approach to streaming that is part of the Spark platform), and Apache Samza (a streaming framework built on top of Apache Kafka). Storm and the other streaming technologies are good at processing data in motion in a scalable and robust way, so when used in conjunction with MapReduce, they can cover the batch-induced blind spot.

So why not do everything in a streaming fashion if it handles real time? The streaming paradigm has issues storing long-term state or doing aggregations over large amounts of data. Put plainly, it's bad at things batch is good at, which is why the two technologies are so complementary. The combination of streaming and batch (particularly MapReduce and Storm) is the foundation for an architectural pattern called the *Lambda Architecture*, which is a two-pronged

approach where you pass the same data through a batch system and a streaming system. The streaming system handles the most recent data for keeping real-time applications happy, while the batch system periodically goes through and gets accurate results on long-term aggregations. You can read more about it at http://lambda-architecture.net.

Fundamentally, the frameworks we mentioned in the past section (Pig, Hive, Crunch, Scalding, and Cascading) are built on MapReduce and have the same limitations as MapReduce. There have been a couple of free and open source approaches for alternative execution engines that are better suited for interactive ad hoc analysis while also being higher level. Cloudera's Impala was one of the first of these and implements SQL through a *Massively Parallel Processing (MPP)* approach to data processing, which has been used by numerous proprietary distributed databases in the past decade (Greenplum, Netezza, Teradata, and many more). Apache Tez is an incremental improvement to generalize how MapReduce passes between phases of MapReduce. Spark uses memory intelligently so that iterative analytics run much faster without much downside over MapReduce. These are just some of the frameworks getting the most attention.

The thing to take away from this conversation is that MapReduce is the 18-wheeler truck hauling data across the country: it has impressive and scalable throughput in delivering large batches. It's not going to win in a quarter-mile race against a race car (the Impalas and Sparks of the world), but in the long haul it can get a lot of work done. Sometimes it makes sense to own both a 18-wheeler and a race car so that you can handle both situations well, but don't be fooled into thinking that MapReduce can be easily overtaken by something "faster."

Summary

The high-level points to remember from this report are:

- Hadoop consists of HDFS, MapReduce, YARN, and the Hadoop ecosystem.
- Hadoop is a distributed system, but tries to hide that fact from the user.
- Hadoop scales out linearly.

- Hadoop doesn't need special hardware.
- You can analyze unstructured data with Hadoop.
- Hadoop is schema-on-read, not schema-on-write.
- Hadoop is open source.
- HDFS can store a lot of data with minimal risk of data loss.
- HDFS has some limitations that you'll have to work around.
- YARN spreads out tasks and workloads intelligently over a cluster.
- YARN does everything in the background and doesn't require much interaction with Hadoop developers.
- MapReduce can process a lot of data by using a shared nothing approach that takes advantage of data locality.
- There are technologies that do some things better than MapReduce, but never everything better than MapReduce.

Further Reading

If you feel the need to take it a bit deeper, I suggest you check out some of the following books:

Hadoop: The Definitive Guide *by Tom White*
: Chapters 1, 2, 3, 4, and 5 in the fourth edition cover a similar scope as this report, but in more detail and speak more to the developer audience. The chapters after that get into more details of implementing Hadoop solutions.

Hadoop Operations *by Eric Sammer*
: This book gives a good overview of Hadoop from more of a system administrator's standpoint. Again, Chapters 1, 2, and 3 mirror the scope of this report and then it starts to go into more detail.

MapReduce Design Patterns *by yours truly, Donald Miner (shameless plug)*
: If you are interested in learning more about MapReduce in particular, this book may seem like it is for advanced readers (well, it is). But if you need more details on the capabilities of MapReduce, this is a good start.

About the Author

Donald Miner is founder of the data science firm Miner & Kasch and specializes in Hadoop enterprise architecture and applying machine learning to real-world business problems. He is the author of the O'Reilly book *MapReduce Design Patterns*. He has architected and implemented dozens of mission-critical and large-scale Hadoop systems within the U.S. government and Fortune 500 companies. He has applied machine learning techniques to analyze data across several verticals, including financial, retail, telecommunications, healthcare, government intelligence, and entertainment. His PhD is from the University of Maryland, Baltimore County, where he focused on machine learning and multiagent systems. He lives in Maryland with his wife and two young sons.