1.1 Data-management systems: a crash course

Relational database systems have been around for a few decades and have been hugely successful in solving data storage, serving, and processing problems over the years. Several large companies have built their systems using relational database systems, online transactional systems, as well as back-end analytics applications.

Online transaction processing (OLTP) systems are used by applications to record transactional information in real time. They're expected to return responses quickly, typically in milliseconds. For instance, the cash registers in retail stores record purchases and payments in real time as customers make them. Banks have large OLTP systems that they use to record transactions between users like transferring of funds and such. OLTP systems aren't limited to money transactions. Web companies like LinkedIn also have such applications—for instance, when users connect with other users. The term *transaction* in OLTP refers to transactions in the context of databases, not financial transactions.

Online analytical processing (OLAP) systems are used to answer analytical queries about the data stored in them. In the context of retailers, these would mean systems that generate daily, weekly, and monthly reports of sales and slice and dice the information to allow analysis of it from several different perspectives. OLAP falls in the domain of business intelligence, where data is explored, processed, and analyzed to glean information that could further be used to drive business decisions. For a company like LinkedIn, where the establishing of connections counts as transactions, analyzing the connectedness of the graph and generating reports on things like the number of average connections per user falls in the category of business intelligence; this kind of processing would likely be done using OLAP systems.

Relational databases, both open source and proprietary, have been successfully used at scale to solve both these kinds of use cases. This is clearly highlighted by the balance sheets of companies like Oracle, Vertica, Teradata, and others. Microsoft and IBM have their share of the pie too. All such systems provide full ACID[3] guarantees. Some scale better than others; some are open source, and others require you to pay steep licensing fees.

The internal design of relational databases is driven by relational math, and these systems require an up-front definition of schemas and types that the data will thereafter adhere to. Over time, SQL became the standard way of interacting with these systems, and it has been widely used for several years. SQL is arguably a lot easier to write and takes far less time than coding up custom access code in programming languages. But it might not be the best way to express the access patterns in every situation, and that's where issues like object-relational mismatch arose.

[3] For those who don't know (or don't remember), ACID is an acronym standing for *atomicity*, *consistency*, *isolation*, and *durability*. These are fundamental principles used to reason about data systems. See http://en.wikipedia.org/wiki/ACID for an introduction.

Any problem in computer science can be solved with a level of indirection. Solving problems like object-relational mismatch was no different and led to frameworks being built to alleviate the pain.

1.1.1 Hello, Big Data

Let's take a closer look at the term *Big Data*. To be honest, it's become something of a loaded term, especially now that enterprise marketing engines have gotten hold of it. We'll keep this discussion as grounded as possible.

What is Big Data? Several definitions are floating around, and we don't believe that any of them explains the term clearly. Some definitions say that Big Data means the data is large enough that you have to think about it in order to gain insights from it. Others say it's Big Data when it stops fitting on a single machine. These definitions are accurate in their own respect but not necessarily complete. Big Data, in our opinion, is a fundamentally different way of thinking about data and how it's used to drive business value. Traditionally, there were transaction recording (OLTP) and analytics (OLAP) on the recorded data. But not much was done to understand the reasons behind the transactions or what factors contributed to business taking place the way it did, or to come up with insights that could drive the customer's behavior directly. In the context of the earlier LinkedIn example, this could translate into finding missing connections based on user attributes, second-degree connections, and browsing behavior, and then prompting users to connect with people they may know. Effectively pursuing such initiatives typically requires working with a large amount of varied data.

This new approach to data was pioneered by web companies like Google and Amazon, followed by Yahoo! and Facebook. These companies also wanted to work with different kinds of data, and it was often unstructured or semistructured (such as logs of users' interactions with the website). This required the system to process several orders of magnitude more data. Traditional relational databases were able to scale up to a great extent for some use cases, but doing so often meant expensive licensing and/or complex application logic. But owing to the data models they provided, they didn't do a good job of working with evolving datasets that didn't adhere to the schemas defined up front. There was a need for systems that could work with different kinds of data formats and sources without requiring strict schema definitions up front, and do it at scale. The requirements were different enough that going back to the drawing board made sense to some of the internet pioneers, and that's what they did. This was the dawn of the world of *Big Data systems* and *NoSQL*. (Some might argue that it happened much later, but that's not the point. This did mark the beginning of a different way of thinking about data.)

As part of this innovation in data management systems, several new technologies were built. Each solved different use cases and had a different set of design assumptions and features. They had different data models, too.

How did we get to HBase? What fueled the creation of such a system? That's up next.

1.1.2 Data innovation

As we now know, many prominent internet companies, most notably Google, Amazon, Yahoo!, and Facebook, were on the forefront of this explosion of data. Some generated their own data, and others collected what was freely available; but managing these vastly different kinds of datasets became core to doing business. They all started by building on the technology available at the time, but the limitations of this technology became limitations on the continued growth and success of these businesses. Although data management technology wasn't core to the businesses, it became essential for *doing* business. The ensuing internal investment in technical research resulted in many new experiments in data technology.

Although many companies kept their research closely guarded, Google chose to talk about its successes. The publications that shook things up were the Google File System[4] and MapReduce papers.[5] Taken together, these papers represented a novel approach to the storage and processing of data. Shortly thereafter, Google published the Bigtable paper,[6] which provided a complement to the storage paradigm provided by its file system. Other companies built on this momentum, both the ideas and the habit of publishing their successful experiments. As Google's publications provided insight into indexing the internet, Amazon published Dynamo,[7] demystifying a fundamental component of the company's shopping cart.

It didn't take long for all these new ideas to begin condensing into open source implementations. In the years following, the data management space has come to host all manner of projects. Some focus on fast key-value stores, whereas others provide native data structures or document-based abstractions. Equally diverse are the intended access patterns and data volumes these technologies support. Some forego writing data to disk, sacrificing immediate persistence for performance. Most of these technologies don't hold ACID guarantees as sacred. Although proprietary products do exist, the vast majority of the technologies are open source projects. Thus, these technologies as a collection have come to be known as *NoSQL*.

Where does HBase fit in? HBase does qualify as a NoSQL store. It provides a *key-value API*, although with a twist not common in other key-value stores. It promises *strong consistency* so clients can see data immediately after it's written. HBase runs on *multiple nodes* in a cluster instead of on a single machine. It doesn't expose this detail to its clients. Your application code doesn't know if it's talking to 1 node or 100, which makes things simpler for everyone. HBase is designed for *terabytes to petabytes* of data, so it optimizes for this use case. It's a part of the *Hadoop ecosystem* and depends on some

[4] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google File System," Google Research Publications, http://research.google.com/archive/gfs.html.

[5] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Google Research Publications, http://research.google.com/archive/mapreduce.html.

[6] Fay Chang et al., "Bigtable: A Distributed Storage System for Structured Data," Google Research Publications, http://research.google.com/archive/bigtable.html.

[7] Werner Vogels, "Amazon's Dynamo," All Things Distributed, www.allthingsdistributed.com/2007/10/amazons_dynamo.html.

key features, such as data redundancy and batch processing, to be provided by other parts of Hadoop.

Now that you have some context for the environment at large, let's consider specifically the beginnings of HBase.

1.1.3 The rise of HBase

Pretend that you're working on an open source project for searching the web by crawling websites and indexing them. You have an implementation that works on a small cluster of machines but requires a lot of manual steps. Pretend too that you're working on this project around the same time Google publishes papers about its data-storage and -processing frameworks. Clearly, you would jump on these publications and spearhead an open source implementation based on them. Okay, maybe you wouldn't, and we surely didn't; but Doug Cutting and Mike Cafarella did.

Built out of Apache Lucene, Nutch was their open source web-search project and the motivation for the first implementation of Hadoop.[8] From there, Hadoop began to receive lots of attention from Yahoo!, which hired Cutting and others to work on it full time. From there, Hadoop was extracted out of Nutch and eventually became an Apache top-level project. With Hadoop well underway and the Bigtable paper published, the groundwork existed to implement an open source Bigtable on top of Hadoop. In 2007, Cafarella released code for an experimental, open source Bigtable. He called it HBase. The startup Powerset decided to dedicate Jim Kellerman and Michael Stack to work on this Bigtable analog as a way of contributing back to the open source community on which it relied.[9]

HBase proved to be a powerful tool, especially in places where Hadoop was already in use. Even in its infancy, it quickly found production deployment and developer support from other companies. Today, HBase is a top-level Apache project with thriving developer and user communities. It has become a core infrastructure component and is being run in production at scale worldwide in companies like StumbleUpon, Trend Micro, Facebook, Twitter, Salesforce, and Adobe.

HBase isn't a cure-all of data management problems, and you might include another technology in your stack at a later point for a different use case. Let's look at how HBase is being used today and the types of applications people have built using it. Through this discussion, you'll gain a feel for the kinds of data problems HBase can solve and has been used to tackle.

1.2 HBase use cases and success stories

Sometimes the best way to understand a software product is to look at how it's used. The kinds of problems it solves and how those solutions fit into a larger application architecture can tell you a lot about a product. Because HBase has seen a number of

[8] A short historical summary was published by Doug Cutting at http://cutting.wordpress.com/2009/08/10/joining-cloudera/.

[9] See Jim Kellerman's blog post at http://mng.bz/St47.

publicized production deployments, we can do just that. This section elaborates on some of the more common use cases that people have successfully used HBase to solve.

> NOTE Don't limit yourself to thinking that HBase can solve only these kinds of use cases. It's a nascent technology, and innovation in terms of use cases is what drives the development of the system. If you have a new idea that you think can benefit from the features HBase offers, try it. The community would love to help you during the process and also learn from your experiences. That's the spirit of open source software.

HBase is modeled after Google's Bigtable, so we'll start our exploration with the canonical Bigtable problem: storing the internet.

1.2.1 The canonical web-search problem: the reason for Bigtable's invention

Search is the act of locating information you care about: for example, searching for pages in a textbook that contain the topic you want to read about, or for web pages that have the information you're looking for. Searching for documents containing particular terms requires looking up indexes that map terms to the documents that contain them. To enable search, you have to build these indexes. This is precisely what Google and other search engines do. Their document corpus is the entire internet; the search terms are whatever you type in the search box.

Bigtable, and by extension HBase, provides storage for this corpus of documents. Bigtable supports row-level access so crawlers can insert and update documents individually. The search index can be generated efficiently via MapReduce directly against Bigtable. Individual document results can be retrieved directly. Support for all these access patterns was key in influencing the design of Bigtable. Figure 1.1 illustrates the critical role of Bigtable in the web-search application.

> NOTE In the interest of brevity, this look at Bigtable doesn't do the original authors justice. We highly recommend the three papers on Google File System, MapReduce, and Bigtable as required reading for anyone curious about these technologies. You won't be disappointed.

With the canonical HBase example covered, let's look at other places where HBase has found purchase. The adoption of HBase has grown rapidly over the last couple of years. This has been fueled by the system becoming more reliable and performant, due in large part to the engineering effort invested by the various companies backing and using it. As more commercial vendors provide support, users are increasingly confident in using the system for critical applications. A technology designed to store a continuously updated copy of the internet turns out to be pretty good at other things internet-related. HBase has found a home filling a variety of roles in and around social-networking companies. From storing communications between individuals to communication analytics, HBase has become a critical infrastructure at Facebook, Twitter, and StumbleUpon, to name a few.

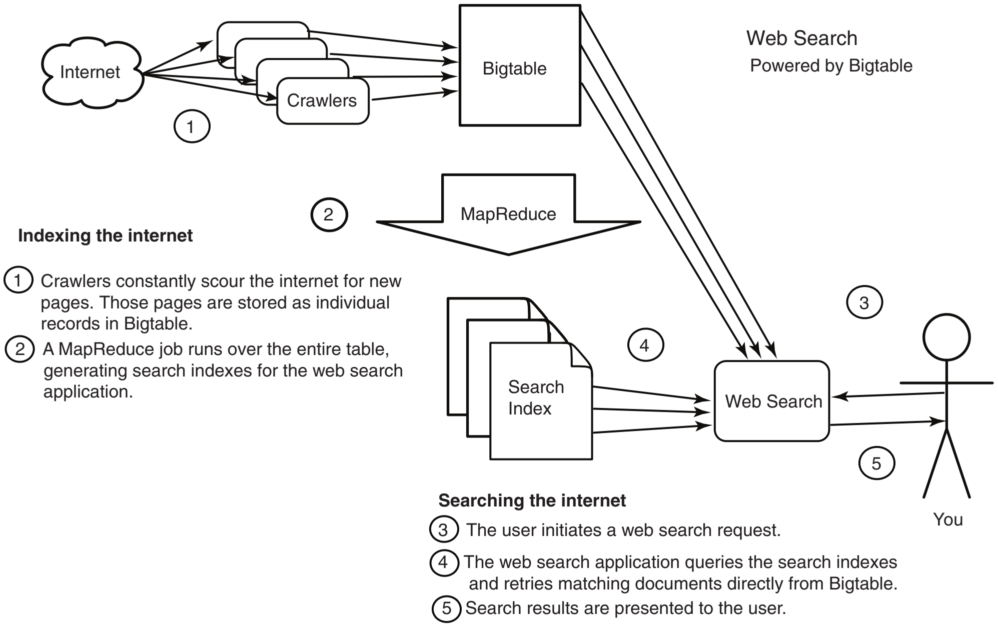

Internet
Crawlers
1
Bigtable
Web Search
Powered by Bigtable
2
MapReduce
Indexing the internet
1 Crawlers constantly scour the internet for new pages. Those pages are stored as individual records in Bigtable.
2 A MapReduce job runs over the entire table, generating search indexes for the web search application.
3
4
Search Index
Web Search
5
You
Searching the internet
3 The user initiates a web search request.
4 The web search application queries the search indexes and retries matching documents directly from Bigtable.
5 Search results are presented to the user.

Figure 1.1 Providing web-search results using Bigtable, simplified. The crawlers—applications collecting web pages—store their data in Bigtable. A MapReduce process scans the table to produce the search index. Search results are queried from Bigtable to display to the user.

HBase has been used in three major types of use cases but it's not limited to those. In the interest of keeping this chapter short and sweet, we'll cover the major use cases here.

1.2.2 Capturing incremental data

Data often trickles in and is added to an existing data store for further usage, such as analytics, processing, and serving. Many HBase use cases fall in this category—using HBase as the data store that captures incremental data coming in from various data sources. These data sources can be, for example, web crawls (the canonical Bigtable use case that we talked about), advertisement impression data containing information about which user saw what advertisement and for how long, or time series data generated from recording metrics of various kinds. Let's talk about a few successful use cases and the companies that are behind these projects.

CAPTURING METRICS: OPENTSDB

Web-based products serving millions of users typically have hundreds or thousands of servers in their back-end infrastructure. These servers spread across various functions—serving traffic, capturing logs, storing data, processing data, and so on. To keep the products up and running, it's critical to monitor the health of the servers as well as the software running on these servers (from the OS right up to the application the user is interacting with). Monitoring the entire stack at scale requires systems that can collect and store metrics of all kinds from these different sources. Every company has its own way of achieving this. Some use proprietary tools to collect and visualize metrics; others use open source frameworks.

StumbleUpon built an open source framework that allows the company to collect metrics of all kinds into a single system. Metrics being collected over time can be thought of as basically time-series data: that is, data collected and recorded over time. The framework that StumbleUpon built is called OpenTSDB, which stands for Open Time Series Database. This framework uses HBase at its core to store and access the collected metrics. The intention of building this framework was to have an extensible metrics collection system that could store and make metrics be available for access over a long period of time, as well as allow for all sorts of new metrics to be added as more features are added to the product. StumbleUpon uses OpenTSDB to monitor all of its infrastructure and software, including its HBase clusters. We cover OpenTSDB in detail in chapter 7 as a sample application built on top of HBase.

CAPTURING USER-INTERACTION DATA: FACEBOOK AND STUMBLEUPON

Metrics captured for monitoring are one category. There are also metrics about user interaction with a product. How do you keep track of the site activity of millions of people? How do you know which site features are most popular? How do you use one page view to directly influence the next? For example, who saw what, and how many times was a particular button clicked? Remember the Like button in Facebook and the Stumble and +1 buttons in StumbleUpon? Does this smell like a counting problem? They increment a counter every time a user *likes* a particular topic.

StumbleUpon had its start with MySQL, but as the service became more popular, that technology choice failed it. The online demand of this increasing user load was too much for the MySQL clusters, and ultimately StumbleUpon chose HBase to replace those clusters. At the time, HBase didn't directly support the necessary features. StumbleUpon implemented atomic increment in HBase and contributed it back to the project.

Facebook uses the counters in HBase to count the number of times people *like* a particular page. Content creators and page owners can get near real-time metrics about how many users *like* their pages. This allows them to make more informed decisions about what content to generate. Facebook built a system called Facebook Insights, which needs to be backed by a scalable storage system. The company looked at various options, including RDBMS, in-memory counters, and Cassandra, before settling on HBase. This way, Facebook can scale horizontally and provide the service to millions of users as well as use its existing experience in running large-scale HBase clusters. The system handles tens of billions of events per day and records hundreds of metrics.

TELEMETRY: MOZILLA AND TREND MICRO

Operational and software-quality data includes more than just metrics. Crash reports are an example of useful software-operational data that can be used to gain insights into the quality of the software and plan the development roadmap. This isn't necessarily related to web servers serving applications. HBase has been successfully used to capture and store crash reports that are generated from software crashes on users' computers.

The Mozilla Foundation is responsible for the Firefox web browser and Thunderbird email client. These tools are installed on millions of computers worldwide and run on a wide variety of OSs. When one of these tools crashes, it may send a crash report back to Mozilla in the form of a bug report. How does Mozilla collect these reports? What use are they once collected? The reports are collected via a system called Socorro and are used to direct development efforts toward more stable products. Socorro's data storage and analytics are built on HBase.[10]

The introduction of HBase enabled basic analysis over far more data than was previously possible. This analysis was used to direct Mozilla's developer focus to great effect, resulting in the most bug-free release ever.

Trend Micro provides internet security and threat-management services to corporate clients. A key aspect of security is awareness, and log collection and analysis are critical for providing that awareness in computer systems. Trend Micro uses HBase to manage its web reputation database, which requires both row-level updates and support for batch processing with MapReduce. Much like Mozilla's Socorro, HBase is also used to collect and analyze log activity, collecting billions of records every day. The flexible schema in HBase allows data to easily evolve over time, and Trend Micro can add new attributes as analysis processes are refined.

ADVERTISEMENT IMPRESSIONS AND CLICKSTREAM

Over the last decade or so, online advertisements have become a major source of revenue for web-based products. The model has been to provide free services to users but have ads linked to them that are targeted to the user using the service at the time. This kind of targeting requires detailed capturing and analysis of user-interaction data to understand the user's profile. The ad to be displayed is then selected based on that profile. Fine-grained user-interaction data can lead to building better models, which in turn leads to better ad targeting and hence more revenue. But this kind of data has two properties: it comes in the form of a continuous stream, and it can be easily partitioned based on the user. In an ideal world, this data should be available to use as soon as it's generated, so the user-profile models can be improved continuously without delay—that is, in an online fashion.

Online vs. offline systems

The terms *online* and *offline* have come up a couple times. For the uninitiated, these terms describe the conditions under which a software system is expected to perform. Online systems have low-latency requirements. In some cases, it's better for these systems to respond with no answer than to take too long producing the correct answer. You can think of a system as online if there's a user at the other end impatiently tapping their foot. Offline systems don't have this low-latency requirement. There's a user waiting for an answer, but that response isn't expected immediately.

[10] Laura Thomson, "Moving Socorro to HBase," Mozilla WebDev, http://mng.bz/L2k9.

> *(continued)*
> The intent to be an online or an offline system influences many technology decisions when implementing an application. HBase is an online system. Its tight integration with Hadoop MapReduce makes it equally capable of offline access as well.

These factors make collecting user-interaction data a perfect fit for HBase, and HBase has been successfully used to capture raw clickstream and user-interaction data incrementally and then process it (clean it, enrich it, use it) using different processing mechanisms (MapReduce being one of them). If you look for companies that do this, you'll find plenty of examples.

1.2.3 Content serving

One of the major use cases of databases traditionally has been that of serving content to users. Applications that are geared toward serving different types of content are backed by databases of all shapes, sizes, and colors. These applications have evolved over the years, and so have the databases they're built on top of. A vast amount of content of varied kinds is available that users want to consume and interact with. In addition, accessibility to such applications has grown, owing to this burgeoning thing called the internet and an even more rapidly growing set of devices that can connect to it. The various kinds of devices lead to another requirement: different devices need the same content in different formats.

That's all about users *consuming* content. In another entirely different use case, users *generate* content: tweets, Facebook posts, Instagram pictures, and micro blogs are just a few examples.

The bottom line is that users consume and generate a lot of content. HBase is being used to back applications that allow a large number of users interacting with them to either consume or generate content.

A content management system (CMS) allows for storing and serving content, as well as managing everything from a central location. More users and more content being generated translates into a requirement for a more scalable CMS solution. Lily,[11] a scalable CMS, uses HBase as its back end, along with other open source frameworks such as Solr to provide a rich set of functionality.

Salesforce provides a hosted CRM product that exposes rich relational database functionality to customers through a web browser interface. Long before Google was publishing papers about its proto-NoSQL systems, the most reasonable choice to run a large, carefully scrutinized database in production was a commercial RDBMS. Over the years, Salesforce has scaled that approach to do hundreds of millions of transactions per day, through a combination of database sharding and cutting-edge performance engineering.

[11] Lily Content Management System: www.lilyproject.org.

When looking for ways to expand its database arsenal to include distributed database systems, Salesforce evaluated the full spectrum of NoSQL technologies before deciding to implement HBase.[12] The primary factor in the choice was consistency. Bigtable-style systems are the only architectural approach that combines seamless horizontal scalability with strong record-level consistency. Additionally, Salesforce already used Hadoop for doing large offline batch processing, so the company was able to take advantage of in-house expertise in running and administering systems on the Hadoop stack.

URL SHORTENERS

URL shorteners gained a lot of popularity in the recent past, and many of them cropped up. StumbleUpon has its own, called su.pr. Su.pr uses HBase as its back end, and that allows it to scale up—shorten URLs and store tons of short URLs and their mapping to the longer versions.

SERVING USER MODELS

Often, the content being served out of HBase isn't consumed directly by users, but is instead used to make decisions about what should be served. It's metadata that is used to enrich the user's interaction.

Remember the user profiles we talked about earlier in the context of ad serving? Those profiles (or models) can also be served out of HBase. Such models can be of various kinds and can be used for several different use cases, from deciding what ad to serve to a particular user, to deciding price offers in real time when users shop on an e-commerce portal, to adding context to user interaction and serving back information the user asked for while searching for something on a search engine. There are probably many such use cases that aren't publicly talked about, and mentioning them could get us into trouble.

Runa[13] serves user models that are used to make real-time price decisions and make offers to users during their engagement with an e-commerce portal. The models are fine-tuned continuously with the help of new user data that comes in.

1.2.4 Information exchange

The world is becoming more connected by the day, with all sorts of social networks cropping up.[14] One of the key aspects of these social networks is the fact that users can interact using them. Sometimes these interactions are in groups (small and large alike); other times, the interaction is between two individuals. Think of hundreds of millions of people having conversations over these networks. They aren't happy with just the ability to communicate with people far away; they also want to look at the history of all their communication with others. Luckily for social network companies, storage is cheap, and innovations in Big Data systems allow them to use the cheap storage to their advantage.[15]

[12] This statement is based on personal conversations with some of the engineers at Salesforce.

[13] www.runa.com.

[14] Some might argue that connecting via social networks doesn't mean being more social. That's a philosophical discussion, and we'll stay out of it. It has nothing to do with HBase, right?

[15] Plus all those ad dollars need to be put to use somehow.

3 Distributed HBase, HDFS, and MapReduce

This chapter covers

- HBase as a distributed storage system
- When to use MapReduce instead of the key-value API
- MapReduce concepts and workflow
- How to write MapReduce applications with HBase
- How to use HBase for map-side joins in MapReduce
- Examples of using HBase with MapReduce

As you've realized, HBase is built on Apache Hadoop. What may not yet be clear to you is why. Most important, what benefits do we, as application developers, enjoy from this relationship? HBase depends on Hadoop for two separate concerns. Hadoop MapReduce provides a distributed computation framework for high-throughput *data access*. The Hadoop Distributed File System (HDFS) gives HBase a storage layer providing *availability* and *reliability*. In this chapter, you'll see how TwitBase is able to take advantage of this data access for bulk processing and how HBase uses HDFS to guarantee availability and reliability.

To begin this chapter, we'll show you why MapReduce is a valuable alternative access pattern for processing data in HBase. Then we'll describe Hadoop MapReduce in general. With this knowledge, we'll tie it all back into HBase as a distributed system. We'll show you how to use HBase from MapReduce jobs and explain some useful tricks you can do with HBase from MapReduce. Finally, we'll show you how HBase provides availability, reliability, and durability for your data. If you're a seasoned Hadooper and know a bunch about MapReduce and HDFS, you can jump straight to section 3.3 and dive into learning about distributed HBase.

The code used in this chapter is a continuation of the TwitBase project started in the previous chapter and is available at https://github.com/hbaseinaction/twitbase.

3.1 A case for MapReduce

Everything you've seen so far about HBase has a focus on *online* operations. You expect every `Get` and `Put` to return results in milliseconds. You carefully craft your `Scans` to transfer as little data as possible over the wire so they'll complete as quickly as possible. You emphasize this behavior in your schema designs, too. The `twits` table's rowkey is designed to maximize physical data locality and minimize the time spent scanning records.

Not all computation must be performed online. For some applications, *offline* operations are fine. You likely don't care if the monthly site traffic summary report is generated in four hours or five hours, as long as it completes before the business owners are ready for it. Offline operations have performance concerns as well. Instead of focusing on individual request latency, these concerns often focus on the entire computation in aggregate. MapReduce is a computing paradigm built for offline (or batch) processing of large amounts of data in an efficient manner.

3.1.1 Latency vs. throughput

The concept of this duality between online and offline concerns has come up a couple times now. This duality exists in traditional relational systems too, with Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). Different database systems are optimized for different access patterns. To get the best performance for the least cost, you need to use the right tool for the job. The same system that handles fast real-time queries isn't necessarily optimized for batch operations on large amounts of data.

Consider the last time you needed to buy groceries. Did you go to the store, buy a single item, and return it to your pantry, only to return to the store for the next item? Well sure, you may do this sometimes, but it's not ideal, right? More likely you made a shopping list, went to the store, filled up your cart, and brought everything home. The entire trip took longer, but the time you spent away from home was shorter per item than taking an entire trip per item. In this example, the time in transit dominates the time spent shopping for, purchasing, and unpacking the groceries. When buying multiple things at once, the average time spent per item purchased is much lower. Making the shopping list results in higher throughput. While in the store, you'll need a bigger cart to accommodate that long shopping list; a small hand basket won't cut it. Tools that work for one approach aren't always sufficient for another.

We think about data access in much the same way. Online systems focus on minimizing the time it takes to access one piece of data—the round trip of going to the store to buy a single item. Response latency measured on the 95th percentile is generally the most important metric for online performance. Offline systems are optimized for access in the aggregate, processing as much as we can all at once in order to maximize throughput. These systems usually report their performance in number of units processed per second. Those units might be requests, records, or megabytes. Regardless of the unit, it's about overall processing time of the task, not the time of an individual unit.

3.1.2 Serial execution has limited throughput

You wrapped up the last chapter by using Scan to look at the most recent twits for a user of TwitBase. Create the start rowkey, create the end rowkey, and execute the scan. That works for exploring a single user, but what if you want to calculate a statistic over all users? Given your user base, perhaps it would be interesting to know what percentage of twits are about Shakespeare. Maybe you'd like to know how many users have mentioned *Hamlet* in their twits.

How can you look at all the twits from all the users in the system to produce these metrics? The Scan object will let you do that:

```
HTableInterface twits = pool.getTable(TABLE_NAME);
Scan s = new Scan();
ResultScanner results = twits.getScanner(s);
for(Result r : results) {
  ... // process twits
}
```

This block of code asks for *all* the data in the table and returns it for your client to iterate through. Does it have a bit of code-smell to you? Even before we explain the inner workings of iterating over the items in the ResultScanner instance, your intuition should flag this as a bad idea. Even if the machine running this loop could process 10 MB/sec, churning through 100 GB of twits would take nearly 3 hours!

3.1.3 Improved throughput with parallel execution

What if you could parallelize this problem—that is, split your gigabyte of twits into pieces and process all the pieces in parallel? You could turn 3 hours into 25 minutes by spinning up 8 threads and running them all in parallel. A laptop has 8 cores and can easily hold 100 GB of data, so assuming it doesn't run out of memory, this should be pretty easy.

Embarrassingly parallel

Many problems in computing are inherently parallel. Only because of incidental concerns must they be written in a serial fashion. Such concerns could be any of programming language design, storage engine implementation, library API, and so on. The challenge falls to you as an algorithm designer to see these situations for what they are. Not all problems are easily parallelizable, but you'll be surprised by how many are once you start to look.

The code for distributing the work over different threads might look something like this:

```
int numSplits = 8;                                                  Split
Split[] splits = split(startrow, endrow, numSplits);             <- work
List<Future<?>> workers = new ArrayList<Future<?>>(numSplits);
ExecutorService es = Executors.newFixedThreadPool(numSplits);
for (final Split split : splits) {                                  Distribute
  workers.add(es.submit(new Runnable() {                            work
    public void run() {
      HTableInterface twits = pool.getTable(TABLE_NAME);
      Scan s = new Scan(split.start, split.end);
      ResultScanner results = twits.getScanner(s);
      for(Result r : results) {
        ...                                       <- Do
      }                                              work
    }
  }));
}
for(Future<?> f : workers) {
  f.get();
  ...                                <- Aggregate
}                                       work
es.shutdownNow();
```

That's not bad, but there's one problem. People are using TwitBase, and before long you'll have 200 GB of twits—and then 1 TB and then 50 TB! Fitting all that data on your laptop's hard drive is a serious challenge, and it's running desperately low on cores. What do you do? You can settle for waiting longer for the computation to finish, but that solution won't last forever as hours quickly turn into days. Parallelizing the computation worked well last time, so you might as well throw more computers at it. Maybe you can buy 20 cheapish servers for the price of 10 fancy laptops.

Now that you have the computing power, you still need to deal with splitting the problem across those machines. Once you've solved that problem, the aggregation step will require a similar solution. And all this while you've assumed everything works as expected. What happens if a thread gets stuck or dies? What if a hard drive fails or the machine suffers random RAM corruption? It would be nice if the workers could restart where they left off in case one of the splits kills the program. How does the aggregation keep track of which splits have finished and which haven't? How do the results get shipped back for aggregation? Parallelizing the problem was pretty easy, but the rest of this distributed computation is painful bookkeeping. If you think about it, the bookkeeping would be required for every algorithm you write. The solution is to make that into a framework.

3.1.4 MapReduce: maximum throughput with distributed parallelism

Enter Hadoop. Hadoop gives us two major components that solve this problem. The *Hadoop Distributed File System (HDFS)* gives all these computers a common, shared file system where they can all access the data. This removes a lot of the pain from farming out the data to the workers and aggregating the work results. Your workers can access the input data from HDFS and write out the processed results to HDFS, and all the others can see them. *Hadoop MapReduce* does all the bookkeeping we described, splitting up the workload and making sure it gets done. Using MapReduce, all you write are the *Do Work* and *Aggregate Work* bits; Hadoop handles the rest. Hadoop refers to the Do Work part as the *Map Step*. Likewise, Aggregate Work is the *Reduce Step*. Using Hadoop MapReduce, you have something like this instead:

```
public class Map
  extends Mapper<LongWritable, Text, Text, LongWritable> {

  protected void map(LongWritable key,
                     Text Value,
                     Context context) {
    ...                  Do
  }                      work
}
```

This code implements the `map` task. This function expects `long` keys and `Text` instances as input and writes out pairs of `Text` to `LongWritable`. `Text` and `LongWritable` are Hadoop-speak for `String` and `Long`, respectively.

Notice all the code you're not writing. There are no split calculations, no Futures to track, and no thread pool to clean up after. Better still, this code can run anywhere on the Hadoop cluster! Hadoop distributes your worker logic around the cluster according to resource availability. Hadoop makes sure every machine receives a unique slice of the `twits` table. Hadoop ensures no work is left behind, even if workers crash.

Your Aggregate Work code is shipped around the cluster in a similar fashion. The Hadoop harness looks something like this:

```
public class Reduce
  extends Reducer<Text, LongWritable, Text, LongWritable> {

  protected void reduce(Text key,
                        Iterable<LongWritable> vals,
                        Context context) {
    ...                   Aggregate
  }                       work
}
```

Here you see the reduce task. The function receives the [String,Long] pairs output from map and produces new [String,Long] pairs. Hadoop also handles collecting the output. In this case, the [String,Long] pairs are written back to the HDFS. You could have just as easily written them back to HBase. HBase provides TableMapper and TableReducer classes to help with that.

You've just seen when and why you'll want to use MapReduce instead of programming directly against the HBase client API. Now let's take a quick look at the MapReduce framework. If you're already familiar with Hadoop MapReduce, feel free to skip down to section 3.3: "HBase in distributed mode."

3.2 An overview of Hadoop MapReduce

In order to provide you with a general-purpose, reliable, fault-tolerant distributed computation harness, MapReduce constrains how you implement your program. These constraints are as follows:

- All computations are implemented as either map or reduce tasks.
- Each task operates over a portion of the total input data.
- Tasks are defined primarily in terms of their input data and output data.
- Tasks depend on their input data and don't communicate with other tasks.

Hadoop MapReduce enforces these constraints by requiring that programs be implemented with map and reduce functions. These functions are composed into a *Job* and run as a unit: first the mappers and then the reducers. Hadoop runs as many simultaneous tasks as it's able. Because there are no runtime dependencies between concurrent tasks, Hadoop can run them in any order as long as the map tasks are run before the reduce tasks. The decisions of how many tasks to run and which tasks to run are up to Hadoop.

Exceptions to every rule

As far as Hadoop MapReduce is concerned, the points outlined previously are more like guidelines than rules. MapReduce is *batch-oriented*, meaning most of its design principles are focused on the problem of distributed *batch processing of large amounts* of data. A system designed for the distributed, real-time processing of an event stream might take a different approach.

On the other hand, Hadoop MapReduce can be abused for any number of other workloads that fit within these constraints. Some workloads are I/O heavy, others are computation heavy. The Hadoop MapReduce framework is a reliable, fault-tolerant job execution framework that can be used for both kinds of jobs. But MapReduce *is* optimized for I/O intensive jobs and makes several optimizations around minimizing network bottlenecks by reducing the amount of data that needs to be transferred over the wire.

3.2.1 MapReduce data flow explained

Implementing programs in terms of Map and Reduce Steps requires a change in how you tackle a problem. This can be quite an adjustment for developers accustomed to other common kinds of programming. Some people find this change so fundamental that they consider it a *change of paradigm*. Don't worry! This claim may or may not be true. We'll make it as easy as possible to think in MapReduce. MapReduce is all about processing large amounts of data in parallel, so let's break down a MapReduce problem in terms of the flow of data.

For this example, let's consider a log file from an application server. Such a file contains information about how a user spends time using the application. Its contents look like this:

```
Date        Time   UserID Activity    TimeSpent

01/01/2011  18:00  user1  load_page1  3s
01/01/2011  18:01  user1  load_page2  5s
01/01/2011  18:01  user2  load_page1  2s
01/01/2011  18:01  user3  load_page1  3s
01/01/2011  18:04  user4  load_page3  10s
01/01/2011  18:05  user1  load_page3  5s
01/01/2011  18:05  user3  load_page5  3s
01/01/2011  18:06  user4  load_page4  6s
01/01/2011  18:06  user1  purchase    5s
01/01/2011  18:10  user4  purchase    8s
01/01/2011  18:10  user1  confirm     9s
01/01/2011  18:10  user4  confirm     11s
01/01/2011  18:11  user1  load_page3  3s
```

Let's calculate the amount of time each user spends using the application. A basic implementation might be to iterate through the file, summing the values of `TimeSpent` for each user. Your program could have a single `HashMap` (or `dict`, for you Pythonistas) with `UserID` as the key and summed `TimeSpent` as the value. In simple pseudo-code, that program might look like this:

```
agg = {}
for line in file:
  record = split(line)                          <- Do work
  agg[record["UserID"]] += record["TimeSpent"]     <- Aggregate work
report(agg)
```

This looks a lot like the serial example from the previous section, doesn't it? Like the serial example, its throughput is limited to a single thread on a single machine. MapReduce is for distributed parallelism. The first thing to do when parallelizing a problem is break it up. Notice that each line in the input file is processed independently from all the other lines. The only time when data from different lines is seen together is during the aggregation step. That means this input file can be parallelized by any number of lines, processed independently, and aggregated to produce exactly the same result. Let's split it into four pieces and assign those pieces to four different machines, as per figure 3.1.

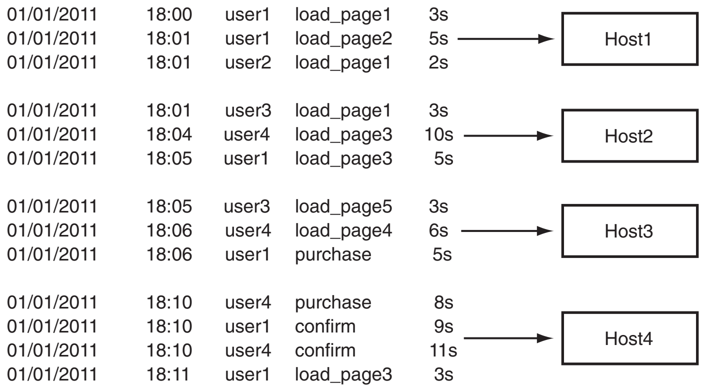

01/01/2011 18:00 user1 load_page1 3s
01/01/2011 18:01 user1 load_page2 5s
01/01/2011 18:01 user2 load_page1 2s
Host1
01/01/2011 18:01 user3 load_page1 3s
01/01/2011 18:04 user4 load_page3 10s
01/01/2011 18:05 user1 load_page3 5s
Host2
01/01/2011 18:05 user3 load_page5 3s
01/01/2011 18:06 user4 load_page4 6s
01/01/2011 18:06 user1 purchase 5s
Host3
01/01/2011 18:10 user4 purchase 8s
01/01/2011 18:10 user1 confirm 9s
01/01/2011 18:10 user4 confirm 11s
01/01/2011 18:11 user1 load_page3 3s
Host4

Figure 3.1 Splitting and assigning work. Each record in the log file can be processed independently, so you split the input file according to the number of workers available.

Look closely at these divisions. Hadoop doesn't know anything about this data other than that it's line-oriented. In particular, there's no effort made to group according to `UserID`. This is an important point we'll address shortly.

Now the work is divided and assigned. How do you rewrite the program to work with this data? As you saw from the `map` and `reduce` stubs, MapReduce operates in terms of key-value pairs. For line-oriented data like this, Hadoop provides pairs of `[line number:line]`. While walking through the MapReduce workflow, we refer in general to this first set of key-value pairs as `[k1,v1]`. Let's start by writing the Map Step, again in pseudo-code:

```
def map(line_num, line):
  record = split(line)
  emit(record["UserID"], record["TimeSpent"])
```

Do work

The Map Step is defined in terms of the lines from the file. For each line in its portion of the file, this Map Step splits the line and produces a new key-value pair of `[UserID:TimeSpent]`. In this pseudo-code, the function `emit` handles reporting the produced pairs back to Hadoop. As you likely guessed, we'll refer to the second set of key-value pairs as `[k2,v2]`. Figure 3.2 continues where the previous figure left off.

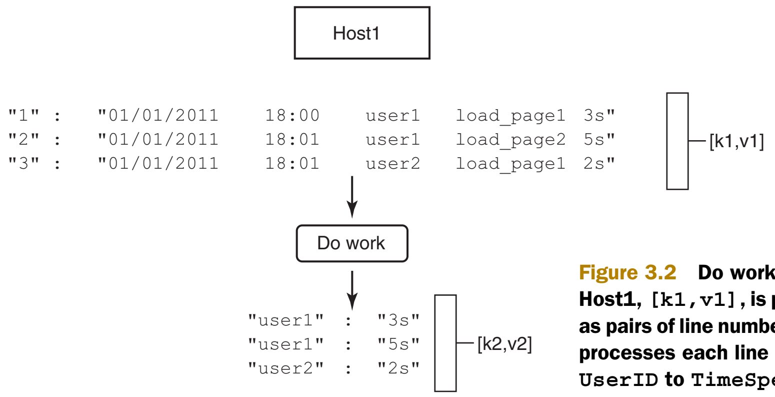

Host1
"1" : "01/01/2011 18:00 user1 load_page1 3s"
"2" : "01/01/2011 18:01 user1 load_page2 5s"
"3" : "01/01/2011 18:01 user2 load_page1 2s"
[k1,v1]
Do work
"user1" : "3s"
"user1" : "5s"
"user2" : "2s"
[k2,v2]

Figure 3.2 Do work. The data assigned to Host1, `[k1,v1]`, is passed to the Map Step as pairs of line number to line. The Map Step processes each line and produces pairs of `UserID` to `TimeSpent`, `[k2,v2]`.

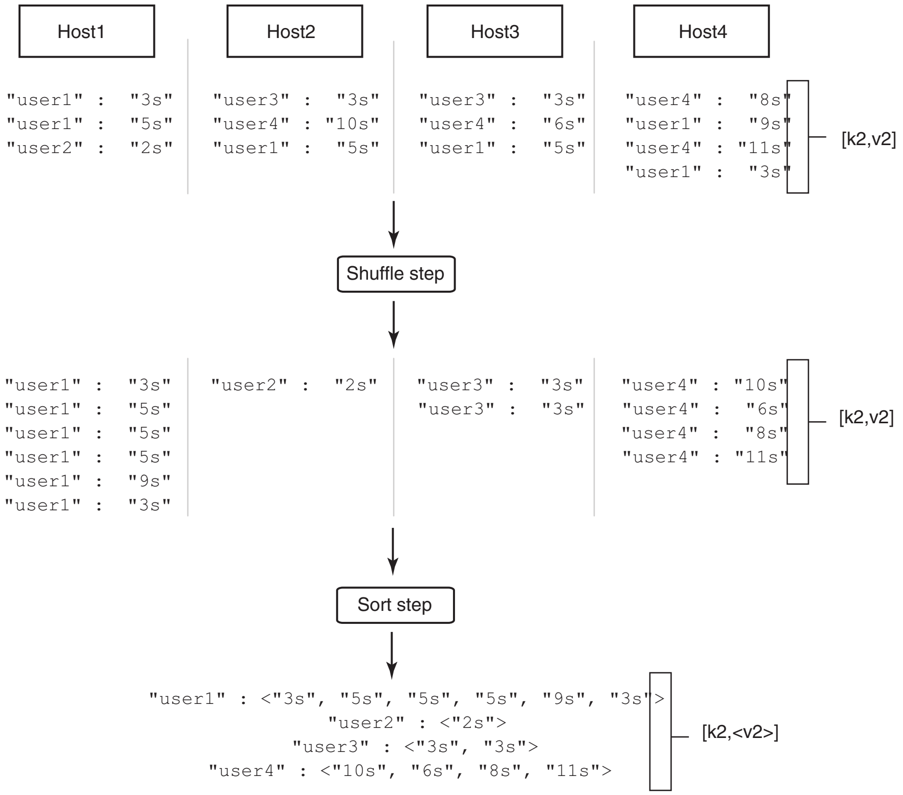


Figure 3.3 Hadoop performs the Shuffle and Sort Step automatically. It serves to prepare the output from the Map Step for aggregation in the Reduce Step. No values are changed by the process; it serves only to reorganize data.

Before Hadoop can pass the values of `[k2,v2]` on to the Reduce Step, a little bookkeeping is necessary. Remember that bit about grouping by `UserID`? The Reduce Step expects to operate over all `TimeSpent` by a given `UserID`. For this to happen correctly, that grouping work happens now. Hadoop calls these the *Shuffle and Sort Steps*. Figure 3.3 illustrates these steps.

MapReduce takes `[k2,v2]`, the output from all four Map Steps on all four servers, and assigns it to reducers. Each reducer is assigned a set of values of `UserID` and it copies those `[k2,v2]` pairs from the mapper nodes. This is called the Shuffle Step. A `reduce` task expects to process all values of `k2` at the same time, so a sort on key is necessary. The output of that Sort Step is `[k2,<v2>]`, a list of `Times` for each `UserID`. With the grouping complete, the `reduce` tasks run. The aggregate work function looks like this:

```
def reduce(user, times):
  for time in times:
    sum += time
  emit(user, sum)
```

Aggregate work

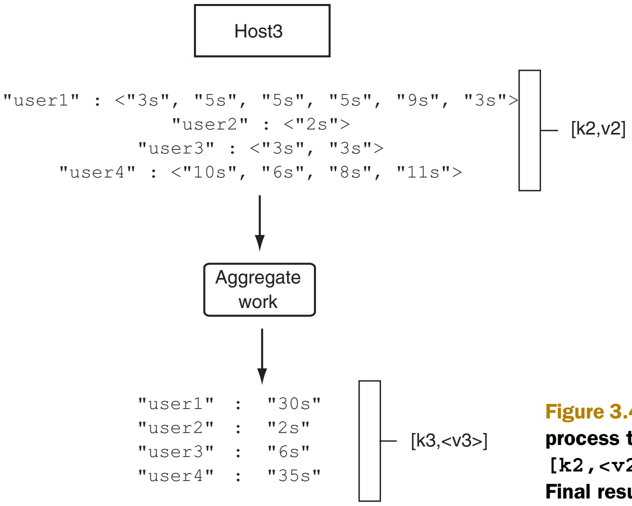


Figure 3.4 Aggregate work. Available servers process the groups of `UserID` to `Times`, `[k2,<v2>]`, in this case, summing the values. Final results are emitted back to Hadoop.

The Reduce Step processes the `[k2,<v2>]` input and produces aggregated work as pairs of `[UserID:TotalTime]`. These sums are collected by Hadoop and written to the output destination. Figure 3.4 illustrates this final step.

You can run this application if you'd like; the source is bundled with the TwitBase code. To do so, compile the application JAR and launch the job like this:

```
$ mvn clean package
[INFO] Scanning for projects...
[INFO]
[INFO] ---------------------------------------------------------------
[INFO] Building TwitBase 1.0.0
[INFO] ---------------------------------------------------------------
...
[INFO] ---------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ---------------------------------------------------------------
$ java -cp target/twitbase-1.0.0.jar \
  HBaseIA.TwitBase.mapreduce.TimeSpent \
  src/test/resource/listing\ 3.3.txt ./out
...
22:53:15 INFO mapred.JobClient: Running job: job_local_0001
22:53:15 INFO mapred.Task:  Using ResourceCalculatorPlugin : null
22:53:15 INFO mapred.MapTask: io.sort.mb = 100
22:53:15 INFO mapred.MapTask: data buffer = 79691776/99614720
22:53:15 INFO mapred.MapTask: record buffer = 262144/327680
22:53:15 INFO mapred.MapTask: Starting flush of map output
22:53:15 INFO mapred.MapTask: Finished spill 0
22:53:15 INFO mapred.Task: Task:attempt_local_0001_m_000000_0 is done. And is
     in the process of commiting
22:53:16 INFO mapred.JobClient:  map 0% reduce 0%
...
```

```
22:53:21 INFO mapred.Task: Task 'attempt_local_0001_r_000000_0' done.
22:53:22 INFO mapred.JobClient:  map 100% reduce 100%
22:53:22 INFO mapred.JobClient: Job complete: job_local_0001
$ cat out/part-r-00000
user1    30
user2    2
user3    6
user4    35
```

That's MapReduce as the data flows. Every MapReduce application performs this sequence of steps, or most of them. If you can follow these basic steps, you've successfully grasped this new paradigm.

3.2.2 MapReduce under the hood

Building a system for general-purpose, distributed, parallel computation is nontrivial. That's precisely why we leave that problem up to Hadoop! All the same, it can be useful to understand how things are implemented, particularly when you're tracking down a bug. As you know, Hadoop MapReduce is a distributed system. Several independent components form the framework. Let's walk through them and see what makes MapReduce tick.

A process called the JobTracker acts as an overseer application. It's responsible for managing the MapReduce applications that run on your cluster. Jobs are submitted to the JobTracker for execution and it manages distributing the workload. It also keeps tabs on all portions of the job, ensuring that failed tasks are retried. A single Hadoop cluster can run multiple MapReduce applications simultaneously. It falls to the JobTracker to oversee resource utilization, and job scheduling as well.

The work defined by the Map Step and Reduce Step is executed by another process called the TaskTracker. Figure 3.5 illustrates the relationship between a JobTracker and its TaskTrackers. These are the actual worker processes. An individual TaskTracker isn't specialized in any way. Any TaskTracker can run any task, be it a `map` or `reduce`, from any job. Hadoop is smart and doesn't randomly spray work across the

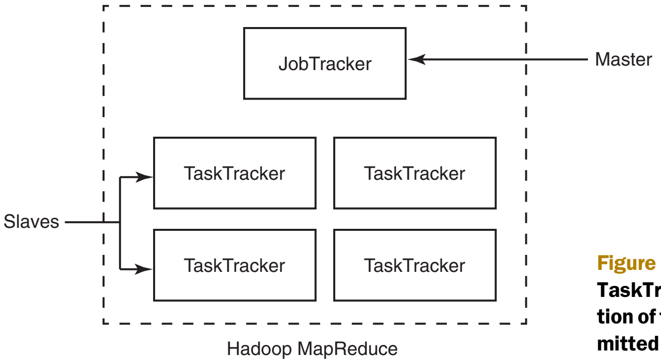


Figure 3.5 The JobTracker and its TaskTrackers are responsible for execution of the MapReduce applications submitted to the cluster.

nodes. As we mentioned, Hadoop is optimized for minimal network I/O. It achieves this by bringing computation as close as possible to the data. In a typical Hadoop, HDFS DataNodes and MapReduce TaskTrackers are collocated with each other. This allows the `map` and `reduce` tasks to run on the same physical node where the data is located. By doing so, Hadoop can avoid transferring the data over the network. When it isn't possible to run the tasks on the same physical node, running the task in the same rack is a better choice than running it on a different rack. When HBase comes into the picture, the same concepts apply, but in general HBase deployments look different from standard Hadoop deployments. You'll learn about deployment strategies in chapter 10.

3.3 HBase in distributed mode

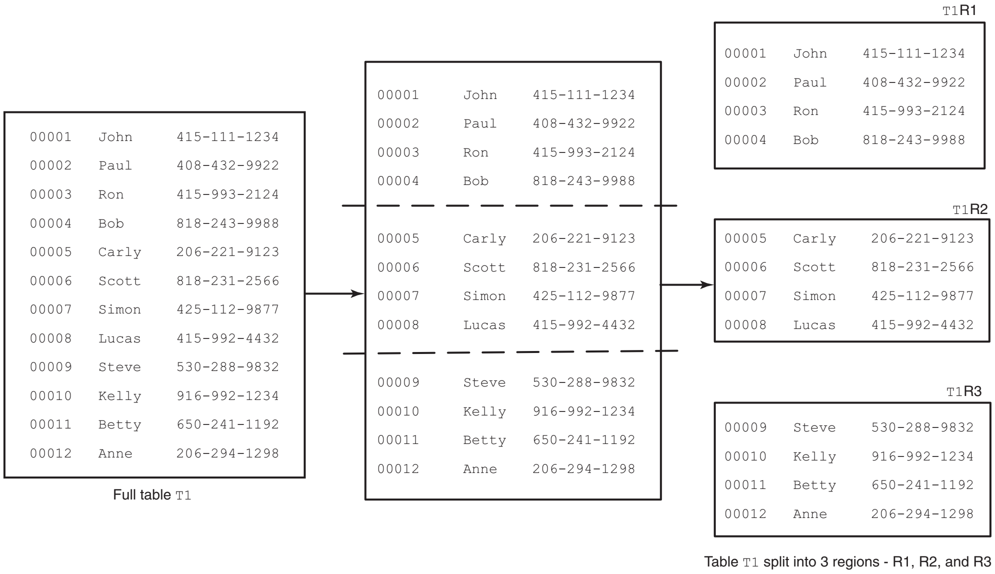
By now you know that HBase is essentially a database built on top of HDFS. It's also sometimes referred to as the *Hadoop Database*, and that's where it got its name. Theoretically, HBase can work on top of any distributed file system. It's just that it's tightly integrated with HDFS, and a lot more development effort has gone into making it work well with HDFS as compared to other distributed file systems. Having said that, from a theoretical standpoint, there is no reason that other file systems can't support HBase. One of the key factors that makes HBase scalable (and fault tolerant) is that it persists its data onto a distributed file system that provides it a single namespace. This is one of the key factors that allows HBase to be a fully consistent data store.

HDFS is inherently a scalable store, but that's not enough to scale HBase as a low-latency data store. There are other factors at play that you'll learn about in this section. Having a good understanding of these is important in order to design your application optimally. This knowledge will enable you to make smart choices about how you want to access HBase, what your keys should look like, and, to some degree, how HBase should be configured. Configuration isn't something you as an application developer should be worried about, but it's likely that you'll have some role to play when bringing HBase into your stack initially.

3.3.1 Splitting and distributing big tables

Just as in any other database, tables in HBase comprise rows and columns, albeit with a different kind of schema. Tables in HBase can scale to billions of rows and millions of columns. The size of each table can run into terabytes and sometimes even petabytes. It's clear at that scale that the entire table can't be hosted on a single machine. Instead, tables are split into smaller chunks that are distributed across multiple servers. These smaller chunks are called *regions* (figure 3.6). Servers that host regions are called *RegionServers*.

RegionServers are typically collocated with HDFS DataNodes (figure 3.7) on the same physical hardware, although that's not a requirement. The only requirement is that RegionServers should be able to access HDFS. They're essentially clients and



00001	John	415-111-1234
00002	Paul	408-432-9922
00003	Ron	415-993-2124
00004	Bob	818-243-9988
00005	Carly	206-221-9123
00006	Scott	818-231-2566
00007	Simon	425-112-9877
00008	Lucas	415-992-4432
00009	Steve	530-288-9832
00010	Kelly	916-992-1234
00011	Betty	650-241-1192
00012	Anne	206-294-1298

Full table T1

00001	John	415-111-1234
00002	Paul	408-432-9922
00003	Ron	415-993-2124
00004	Bob	818-243-9988
00005	Carly	206-221-9123
00006	Scott	818-231-2566
00007	Simon	425-112-9877
00008	Lucas	415-992-4432
00009	Steve	530-288-9832
00010	Kelly	916-992-1234
00011	Betty	650-241-1192
00012	Anne	206-294-1298

T1R1

00001	John	415-111-1234
00002	Paul	408-432-9922
00003	Ron	415-993-2124
00004	Bob	818-243-9988

T1R2

00005	Carly	206-221-9123
00006	Scott	818-231-2566
00007	Simon	425-112-9877
00008	Lucas	415-992-4432

T1R3

00009	Steve	530-288-9832
00010	Kelly	916-992-1234
00011	Betty	650-241-1192
00012	Anne	206-294-1298

Table T1 split into 3 regions - R1, R2, and R3

Figure 3.6 A table consists of multiple smaller chunks called regions.

store/access data on HDFS. The *master* process does the distribution of regions among RegionServers, and each RegionServer typically hosts multiple regions.

Given that the underlying data is stored in HDFS, which is available to all clients as a single namespace, all RegionServers have access to the same persisted files in the file system and can therefore host any region (figure 3.8). By physically collocating DataNodes and RegionServers, you can use the data locality property; that is, RegionServers can theoretically read and write to the local DataNode as the primary DataNode.

You may wonder where the TaskTrackers are in this scheme of things. In some HBase deployments, the MapReduce framework isn't deployed at all if the workload is primarily random reads and writes. In other deployments, where the MapReduce processing is also a part of the workloads, TaskTrackers, DataNodes, and HBase RegionServers can run together.

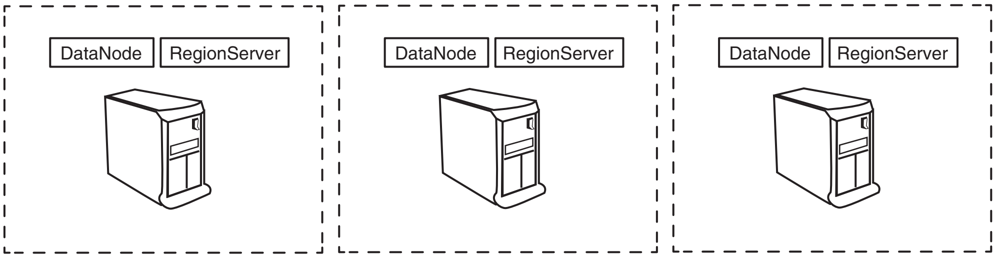
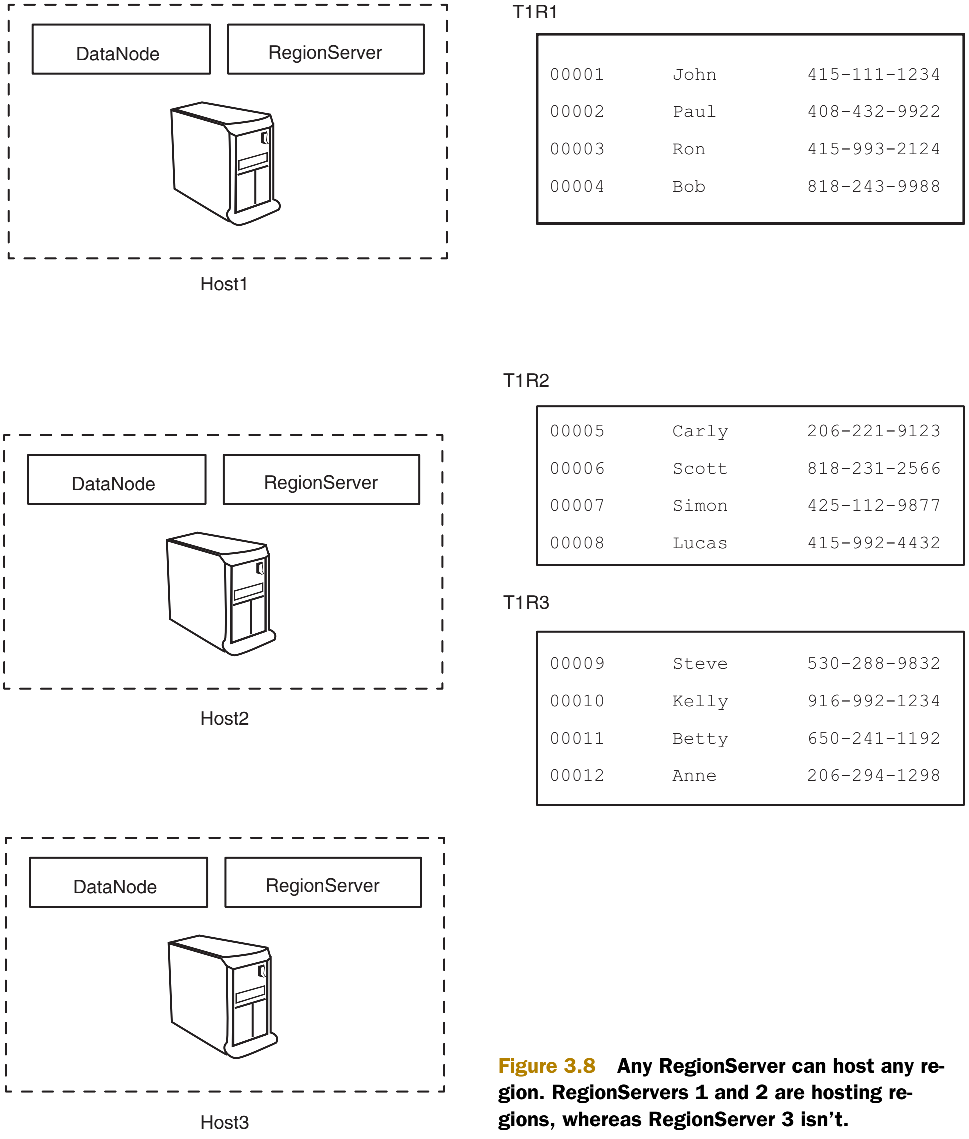


Figure 3.7 HBase RegionServer and HDFS DataNode processes are typically collocated on the same host.



T1R1

00001	John	415-111-1234
00002	Paul	408-432-9922
00003	Ron	415-993-2124
00004	Bob	818-243-9988

T1R2

00005	Carly	206-221-9123
00006	Scott	818-231-2566
00007	Simon	425-112-9877
00008	Lucas	415-992-4432

T1R3

00009	Steve	530-288-9832
00010	Kelly	916-992-1234
00011	Betty	650-241-1192
00012	Anne	206-294-1298

Figure 3.8 Any RegionServer can host any region. RegionServers 1 and 2 are hosting regions, whereas RegionServer 3 isn't.

The size of individual regions is governed by the configuration parameter `hbase.hregion.max.filesize`, which can be configured in the hbase-site.xml file of your deployment. When a region becomes bigger than that size (as you write more data into it), it gets split into two regions.

3.3.2 How do I find my region?

You've learned that tables are split into regions and regions are assigned to RegionServers without any predefined assignment rules. In case you're wondering, regions don't keep moving around in a running system! Region assignment happens when regions split (as they grow in size), when RegionServers die, or when new RegionServers are added to the deployment. An important question to ask here is, "When a region is assigned to a RegionServer, how does my client application (the one doing reads and writes) know its location?"

Two special tables in HBase, `-ROOT-` and `.META.`, help find where regions for various tables are hosted. Like all tables in HBase, `-ROOT-` and `.META.` are also split into regions. `-ROOT-` and `.META.` are both special tables, but `-ROOT-` is more special than `.META.`; `-ROOT-` never splits into more than one region. `.META.` behaves like all other tables and can split into as many regions as required.

When a client application wants to access a particular row, it goes to the `-ROOT-` table and asks it where it can find the region responsible for that particular row. `-ROOT-` points it to the region of the `.META.` table that contains the answer to that question. The `.META.` table consists of entries that the client application uses to determine which RegionServer is hosting the region in question. Think of this like a distributed B+Tree of height 3 (see figure 3.9). The `-ROOT-` table is the `-ROOT-` node of the B+Tree. The `.META.` regions are the children of the `-ROOT-` node (`-ROOT-` region)n and the regions of the user tables (leaf nodes of the B+Tree) are the children of the `.META.` regions.

Let's put `-ROOT-` and `.META.` into the example; see figure 3.10. Note that the region assignments shown here are arbitrary and don't represent how they will happen when such a system is deployed.

3.3.3 How do I find the -ROOT- table?

You just learned how the `-ROOT-` and `.META.` tables help you find out other regions in the system. At this point, you might be left with a question: "Where is the `-ROOT-` table?" Let's figure that out now.

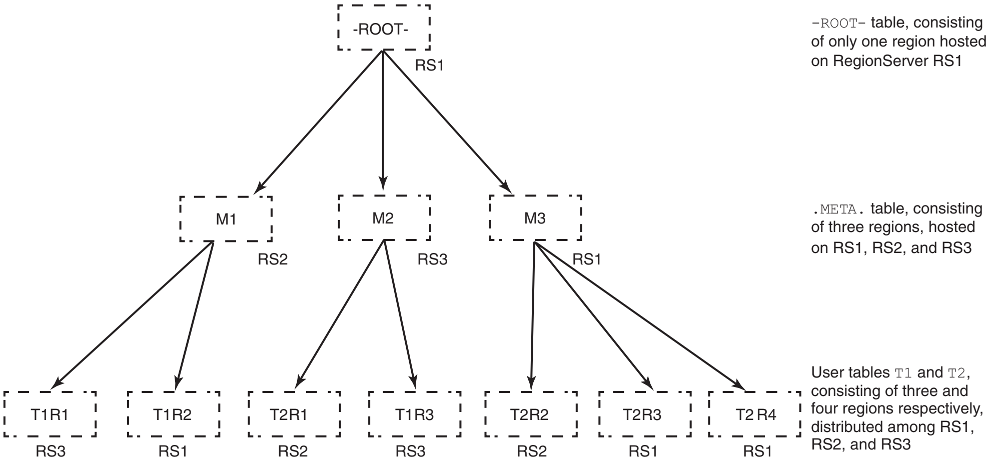
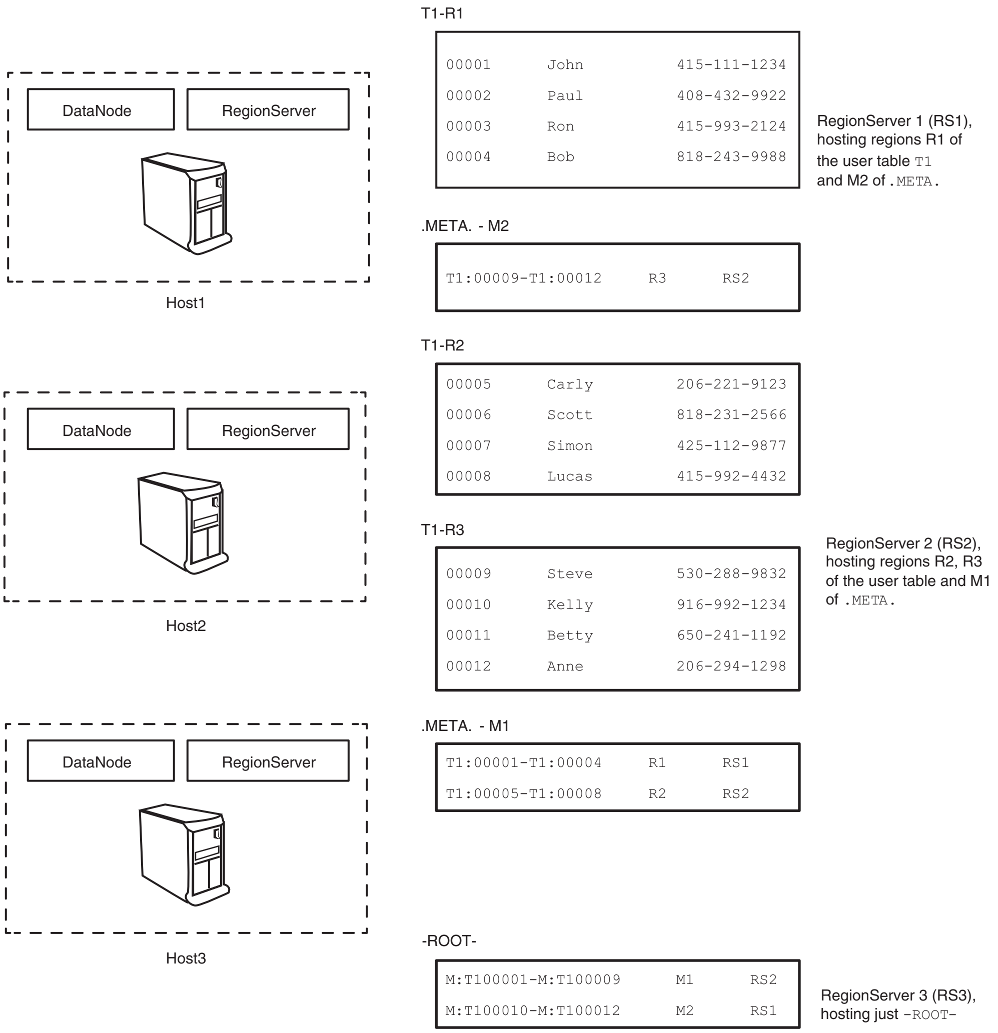

-ROOT-
RS1
M1
RS2
M2
RS3
M3
RS1
T1R1
RS3
T1R2
RS1
T2R1
RS2
T1R3
RS3
T2R2
RS2
T2R3
RS1
T2 R4
RS1
-ROOT- table, consisting of only one region hosted on RegionServer RS1
.META. table, consisting of three regions, hosted on RS1, RS2, and RS3
User tables T1 and T2, consisting of three and four regions respectively, distributed among RS1, RS2, and RS3

Figure 3.9 **`-ROOT-`, `.META.`, and user tables viewed as a B+Tree**


DataNode
RegionServer
Host1
DataNode
RegionServer
Host2
DataNode
RegionServer
Host3
T1-R1
.META. - M2
T1-R2
T1-R3
.META. - M1
-ROOT-
RegionServer 1 (RS1), hosting regions R1 of the user table T1 and M2 of .META.
RegionServer 2 (RS2), hosting regions R2, R3 of the user table and M1 of .META.
RegionServer 3 (RS3), hosting just -ROOT-

T1-R1

```
00001    John     415-111-1234
00002    Paul     408-432-9922
00003    Ron      415-993-2124
00004    Bob      818-243-9988
```

.META. - M2

```
T1:00009-T1:00012    R3    RS2
```

T1-R2

```
00005    Carly    206-221-9123
00006    Scott    818-231-2566
00007    Simon    425-112-9877
00008    Lucas    415-992-4432
```

T1-R3

```
00009    Steve    530-288-9832
00010    Kelly    916-992-1234
00011    Betty    650-241-1192
00012    Anne     206-294-1298
```

.META. - M1

```
T1:00001-T1:00004    R1    RS1
T1:00005-T1:00008    R2    RS2
```

-ROOT-

```
M:T100001-M:T100009    M1    RS2
M:T100010-M:T100012    M2    RS1
```

Figure 3.10 User table `T1` in HBase, along with `-ROOT-` and `.META.`, distributed across the various RegionServers

The entry point for an HBase system is provided by another system called ZooKeeper (http://zookeeper.apache.org/). As stated on ZooKeeper's website, ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. It's a highly available, reliable distributed configuration service. Just as HBase is modeled after Google's BigTable, ZooKeeper is modeled after Google's Chubby.[1]

The client interaction with the system happens in steps, where ZooKeeper is the point of entry, as mentioned earlier. These steps are highlighted in figure 3.11.

[1] Mike Burrow, "The Chubby Lock Service for Loosely-Coupled Distributed Systems," Google Research Publications, http://research.google.com/archive/chubby.html.

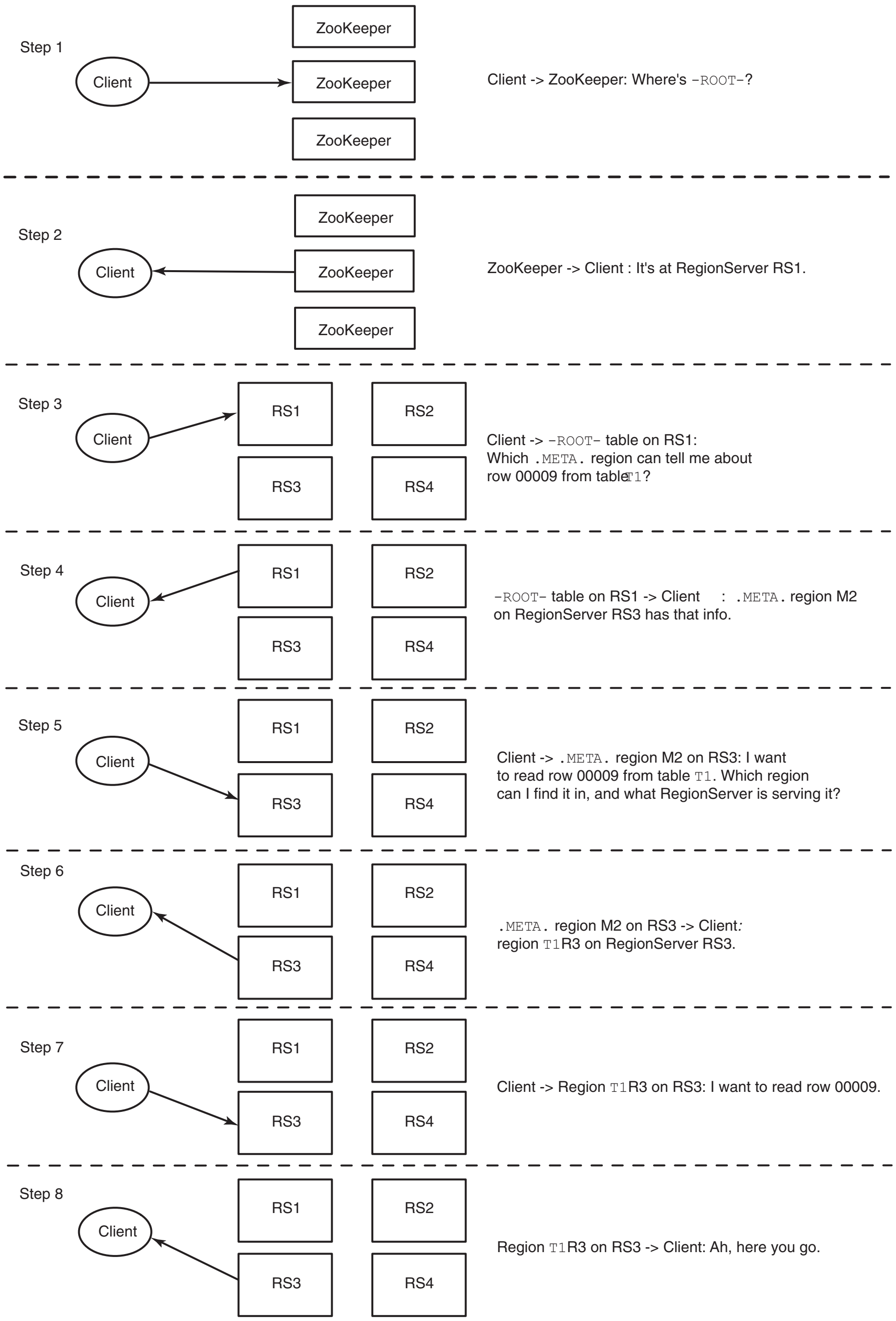

Step 1
Client
ZooKeeper
ZooKeeper
ZooKeeper
Client -> ZooKeeper: Where's -ROOT-?
Step 2
Client
ZooKeeper
ZooKeeper
ZooKeeper
ZooKeeper -> Client : It's at RegionServer RS1.
Step 3
Client
RS1
RS2
RS3
RS4
Client -> -ROOT- table on RS1:
Which .META. region can tell me about
row 00009 from table T1?
Step 4
Client
RS1
RS2
RS3
RS4
-ROOT- table on RS1 -> Client : .META. region M2
on RegionServer RS3 has that info.
Step 5
Client
RS1
RS2
RS3
RS4
Client -> .META. region M2 on RS3: I want
to read row 00009 from table T1. Which region
can I find it in, and what RegionServer is serving it?
Step 6
Client
RS1
RS2
RS3
RS4
.META. region M2 on RS3 -> Client:
region T1R3 on RegionServer RS3.
Step 7
Client
RS1
RS2
RS3
RS4
Client -> Region T1R3 on RS3: I want to read row 00009.
Step 8
Client
RS1
RS2
RS3
RS4
Region T1R3 on RS3 -> Client: Ah, here you go.

Figure 3.11 Steps that take place when a client interacts with an HBase system. The interaction starts with ZooKeeper and goes to the RegionServer serving the region with which the client needs to interact. The interaction with the RegionServer could be for reads or writes. The information about `-ROOT-` and `.META.` is cached by the client for future interactions and is refreshed if the regions it's expecting to interact with based on that information don't exist on the node it thinks they should be on.

This section gave you an overview of the implementation of HBase's distributed architecture. You can see all these details for yourself on your own cluster. We show you exactly how to explore ZooKeeper, `-ROOT-`, and `.META.` in appendix A.

3.4 HBase and MapReduce

Now that you have an understanding of MapReduce and HBase in distributed mode, let's look at them together. There are three different ways of interacting with HBase from a MapReduce application. HBase can be used as a *data source* at the beginning of a job, as a *data sink* at the end of a job, or as a *shared resource* for your tasks. None of these modes of interaction are particularly mysterious. The third, however, has some interesting use cases we'll address shortly.

All the code snippets used in this section are examples of using the Hadoop MapReduce API. There are no HBase client `HTable` or `HTablePool` instances involved. Those are embedded in the special input and output formats you'll use here. You will, however, use the `Put`, `Delete`, and `Scan` objects with which you're already familiar. Creating and configuring the Hadoop Job and Configuration instances can be messy work. These snippets emphasize the HBase portion of that work. You'll see a full working example in section 3.5.

3.4.1 HBase as a source

In the example MapReduce application, you read lines from log files sitting in the HDFS. Those files, specifically the directory in HDFS containing those files, act as the data source for the MapReduce job. The schema of that data source describes `[k1,v1]` tuples as `[line number:line]`. The `TextInputFormat` class configured as part of the job defines this schema. The relevant code from the `TimeSpent` example looks like this:

```
Configuration conf = new Configuration();
Job job = new Job(conf, "TimeSpent");
...
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
```

`TextInputFormat` defines the `[k1,v1]` type for `line number` and `line` as the types `LongWritable` and `Text`, respectively. `LongWritable` and `Text` are serializable Hadoop wrapper types over Java's `Long` and `String`. The associated `map` task definition is typed for consuming these input pairs:

```
public void map(LongWritable key, Text value,
Context context) {
  ...
}
```

HBase provides similar classes for consuming data out of a table. When mapping over data in HBase, you use the same `Scan` class you used before. Under the hood, the row-range defined by the `Scan` is broken into pieces and distributed to all the workers (figure 3.12).

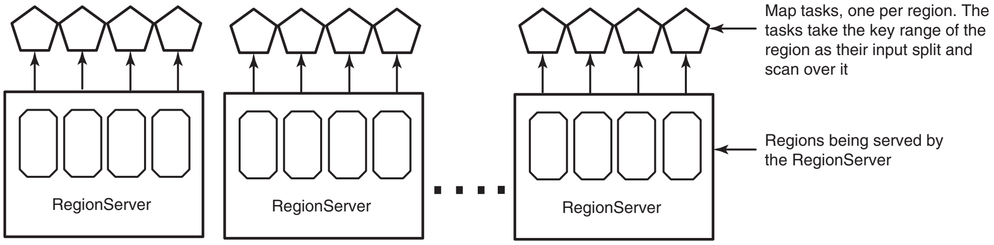

Map tasks, one per region. The tasks take the key range of the region as their input split and scan over it
Regions being served by the RegionServer
RegionServer
RegionServer
RegionServer

Figure 3.12 MapReduce job with mappers taking regions from HBase as their input source. By default, one mapper is created per region.

This is identical to the splitting you saw in figure 3.1. Creating a `Scan` instance for scanning over all rows in a table from MapReduce looks like this:

```
Scan scan = new Scan();
scan.addColumn(Bytes.toBytes("twits"), Bytes.toBytes("twit"));
```

In this case, you're asking the scanner to return only the text from the `twits` table.

Just like consuming text lines, consuming HBase rows requires a schema. All jobs reading from an HBase table accept their `[k1,v1]` pairs in the form of `[rowkey:scan result]`. That's the same scanner result as when you consume the regular HBase API. They're presented using the types `ImmutableBytesWritable` and `Result`. The provided `TableMapper` wraps up these details for you, so you'll want to use it as the base class for your Map Step implementation:

```
protected void map(
    ImmutableBytesWritable rowkey,
    Result result,
    Context context) {
  ...
}
```

Define types for input [kl,vl] the map task receives, in this case from scanner

The next step is to take your `Scan` instance and wire it into MapReduce. HBase provides the handy `TableMapReduceUtil` class to help you initialize the `Job` instance:

```
TableMapReduceUtil.initTableMapperJob(
  "twits",
  scan,
  Map.class,
  ImmutableBytesWritable.class,
  Result.class,
  job);
```

This takes your job-configuration object and sets up the HBase-specific input format (`TableInputFormat`). It then configures MapReduce to read from the table using your `Scan` instance. It also wires in your `Map` and `Reduce` class implementations. From here, you write and run the MapReduce application as normal.

When you run a MapReduce job as described here, one `map` task is launched for every region in the HBase table. In other words, the `map` tasks are partitioned such that each `map` task reads from a region independently. The JobTracker tries to schedule `map` tasks as close to the regions as possibly and take advantage of data locality.

3.4.2 HBase as a sink

Writing to an HBase table from MapReduce (figure 3.13) as a data sink is similar to reading from a table in terms of implementation.

HBase provides similar tooling to simplify the configuration. Let's first make an example of sink configuration in a standard MapReduce application.

In `TimeSpent`, the values of `[k3,v3]` generated by the aggregators are `[UserID:TotalTime]`. In the MapReduce application, they're of the Hadoop serializable types `Text` and `LongWritable`, respectively. Configuring output types is similar to configuring input types, with the exception that the `[k3,v3]` output types can't be inferred by the `OutputFormat`:

```
Configuration conf = new Configuration();
Job job = new Job(conf, "TimeSpent");
job.setOutputKeyClass(Text.class);
job.setOutputValueClass(LongWritable.class);
...
job.setInputFormatClass(TextInputFormat.class);
job.setOutputFormatClass(TextOutputFormat.class);
```

In this case, no line numbers are specified. Instead, the `TextOuputFormat` schema creates a tab-separated output file containing first the `UserID` and then the `TotalTime`. What's written to disk is the `String` representation of both types.

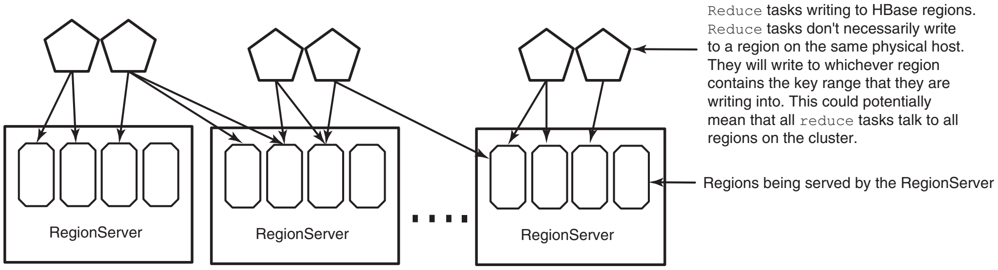
The `Context` object contains the type information. The `reduce` function is defined as

```
public void reduce(Text key, Iterable<LongWritable> values,
Context context) {
  ...
}
```

When writing to HBase from MapReduce, you're again using the regular HBase API. The types of `[k3,v3]` are assumed to be a rowkey and an object for manipulating HBase. That means the values of `v3` must be either `Puts` or `Deletes`. Because both of these object types include the relevant rowkey, the value of `k3` is ignored. Just as the `TableMapper` wraps up these details for you, so does the `TableReducer`:

```
protected void reduce(
    ImmutableBytesWritable rowkey,
    Iterable<Put> values,
    Context context) {
  ...
}
```

Define input types for reducer [k2,{v2}]. These are intermediate key-value pairs that the map tasks are outputting.



`Reduce` tasks writing to HBase regions. `Reduce` tasks don't necessarily write to a region on the same physical host. They will write to whichever region contains the key range that they are writing into. This could potentially mean that all `reduce` tasks talk to all regions on the cluster.

Regions being served by the RegionServer

Figure 3.13 **HBase as a sink for a MapReduce job. In this case, the `reduce` tasks are writing to HBase.**

The last step is wiring your reducer into the job configuration. You need to specify the destination table along with all the appropriate types. Once again, it's `TableMapReduceUtil` to the rescue; it sets up the `TableOutputFormat` for you! You use `IdentityTableReducer`, a provided class, because you don't need to perform any computation in the Reduce Step:

```
TableMapReduceUtil.initTableReducerJob(
  "users",
  IdentityTableReducer.class,
  job);
```

Now your job is completely wired up, and you can proceed as normal. Unlike the case where `map` tasks are reading from HBase, tasks don't necessarily write to a single region. The writes go to the region that is responsible for the rowkey that is being written by the `reduce` task. The default partitioner that assigns the intermediate keys to the `reduce` tasks doesn't have knowledge of the regions and the nodes that are hosting them and therefore can't intelligently assign work to the reducers such that they write to the local regions. Moreover, depending on the logic you write in the `reduce` task, which doesn't have to be the identity reducer, you might end up writing all over the table.

3.4.3 HBase as a shared resource

Reading from and writing to HBase using MapReduce is handy. It gives us a harness for batch processing over data in HBase. A few predefined MapReduce jobs ship with HBase; you can explore their source for more examples of using HBase from MapReduce. But what else can you do with HBase?

One common use of HBase is to support a large map-side join. In this scenario, you're reading from HBase as an indexed resource accessible from all `map` tasks. What is a map-side join, you ask? How does HBase support it? Excellent questions!

Let's back up a little. A *join* is common practice in data manipulation. Joining two tables is a fundamental concept in relational databases. The idea behind a join is to combine records from the two different sets based on like values in a common attribute. That attribute is often called the *join key*.

For example, think back to the `TimeSpent` MapReduce job. It produces a dataset containing a `UserID` and the `TotalTime` they spent on the TwitBase site:

UserID	TimeSpent
Yvonn66	30s
Mario23	2s
Rober4	6s
Masan46	35s

You also have the user information in the TwitBase table that looks like this:

UserID	Name	Email	TwitCount
Yvonn66	Yvonne Marc	Yvonn66@unmercantile.com	48
Masan46	Masanobu Olof	Masan46@acetylic.com	47
Mario23	Marion Scott	Mario23@Wahima.com	56
Rober4	Roberto Jacques	Rober4@slidage.com	2

You'd like to know the ratio of how much time a user spends on the site to their total twit count. Although this is an easy question to answer, right now the relevant data is split between two different datasets. You'd like to join this data such that all the information about a user is in a single row. These two datasets share a common attribute: `UserID`. This will be the join key. The result of performing the join and dropping unused fields looks like this:

```
UserID   TwitCount  TimeSpent

Yvonn66  48         30s
Mario23  56          2s
Rober4    2          6s
Masan46  47         35s
```

Joins in the relational world are a lot easier than in MapReduce. Relational engines enjoy many years of research and tuning around performing joins. Features like indexing help optimize join operations. Moreover, the data typically resides on a single physical server. Joining across multiple relational servers is far more complicated and isn't common in practice. A join in MapReduce means joining on data spread across multiple servers. But the semantics of the MapReduce framework make it easier than trying to do a join across different relational database systems. There are a couple of different variations of each type, but a join implementation is either *map-side* or *reduce-side*. They're referred as map- or reduce-side because that's the task where records from the two sets are linked. Reduce-side joins are more common because they're easier to implement. We'll describe those first.

REDUCE-SIDE JOIN

A reduce-side join takes advantage of the intermediate Shuffle Step to collocate relevant records from the two sets. The idea is to map over both sets and emit tuples keyed on the join key. Once together, the reducer can produce all combinations of values. Let's build out the algorithm.

Given the sample data, the pseudo-code of the `map` task for consuming the `TimeSpent` data looks like this:

```
map_timespent(line_num, line):
  userid, timespent = split(line)
  record = {"TimeSpent" : timespent,
            "type" : "TimeSpent"}
  emit(userid, record)
```

Producing compound records as v2 is common in MapReduce jobs

This `map` task splits the `k1` input line into the `UserID` and `TimeSpent` values. It then constructs a dictionary with `type` and `TimeSpent` attributes. As `[k2,v2]` output, it produces `[UserID:dictionary]`.

A `map` task for consuming the `Users` data is similar. The only difference is that it drops a couple of unrelated fields:

```
map_users(line_num, line):
  userid, name, email, twitcount = split(line)
  record = {"TwitCount" : twitcount,
            "type" : "TwitCount"}
  emit(userid, record)
```

Name and email aren't carried along for the ride

Both map tasks use UserID as the value for k2. This allows Hadoop to group all records for the same user. The reduce task has everything it needs to complete the join:

```
reduce(userid, records):
  timespent_recs = []
  twitcount_recs = []

  for rec in records:
    if rec.type == "TimeSpent":
      rec.del("type")
      timespent_recs.push(rec)
    else:
      rec.del("type")
      twitcount_recs.push(rec)

  for timespent in timespent_recs:
    for twitcount in twitcount_recs:
      emit(userid, merge(timespent, twitcount))
```

Group by type

Once grouped, type attribute is no longer needed

Produce all possible combinations of twitcount and timespent for user; for this example, should be a single value

The reduce task groups records of identical type and produces all possible combinations of the two types as k3. For this specific example, you know there will be only one record of each type, so you can simplify the logic. You also can fold in the work of producing the ratio you want to calculate:

```
reduce(userid, records):
  for rec in records:
    rec.del("type")
  merge(records)
  emit(userid, ratio(rec["TimeSpent"], rec["TwitCount"]))
```

This new and improved reduce task produces the new, joined dataset:

```
UserID   ratio

Yvonn66  30s:48
Mario23   2s:56
Rober4    6s:2
Masan46  35s:47
```

There you have it: the reduce-side join in its most basic glory. One big problem with the reduce-side join is that it requires all [k2,v2] tuples to be shuffled and sorted. For our toy example, that's no big deal. But if the datasets are very, very large, with millions of pairs per value of k2, the overhead of that step can be huge.

Reduce-side joins require shuffling and sorting data between map and reduce tasks. This incurs I/O costs, specifically network, which happens to be the slowest aspect of any distributed system. Minimizing this network I/O will improve join performance. This is where the map-side join can help.

MAP-SIDE JOIN

The map-side join is a technique that isn't as general-purpose as the reduce-side join. It assumes the map tasks can look up random values from one dataset while they iterate over the other. If you happen to want to join two datasets where at least one of them can fit in memory of the map task, the problem is solved: load the smaller dataset into a hash-table so the map tasks can access it while iterating over the other dataset. In these cases, you can skip the Shuffle and Reduce Steps entirely and emit your final output from the Map Step. Let's go back to the same example. This time you'll put the Users dataset into memory. The new map_timespent task looks like this:

```
map_timespent(line_num, line):
  users_recs = read_timespent("/path/to/users.csv")
  userid, timespent = split(line)
  record = {"TimeSpent" : timespent}
  record = merge(record, users_recs[userid])
  emit(userid, ratio(record["TimeSpent"], record["TwitCount"]))
```

Compared to the last version, this looks like cheating! Remember, though, you can only get away with this approach when you can fit one of the datasets entirely into memory. In this case, your join will be much faster.

There are of course implications to doing joins like this. For instance, each map task is processing a single split, which is equal to one HDFS block (typically 64–128 MB), but the join dataset that it loads into memory is 1 GB. Now, 1 GB can certainly fit in memory, but the cost involved in creating a hash-table for a 1 GB dataset for every 128 MB of data being joined makes it not such a good idea.

MAP-SIDE JOIN WITH HBASE

Where does HBase come in? We originally described HBase as a giant hash-table, remember? Look again at the map-side join implementation. Replace users_recs with the Users table in TwitBase. Now you can join over the massive Users table and massive TimeSpent data set in record time! The map-side join using HBase looks like this:

```
map_timespent(line_num, line):
  users_table = HBase.connect("Users")
  userid, timespent = split(line)
  record = {"TimeSpent" : timespent}
  record = merge(record, users_table.get(userid, "info:twitcount"))
  emit(userid, ratio(record["TimeSpent"], record["info:twitcount"]))
```

Think of this as an external hash-table that each map task has access to. You don't need to create that hash-table object for every task. You also avoid all the network I/O involved in the Shuffle Step necessary for a reduce-side join. Conceptually, this looks like figure 3.14.

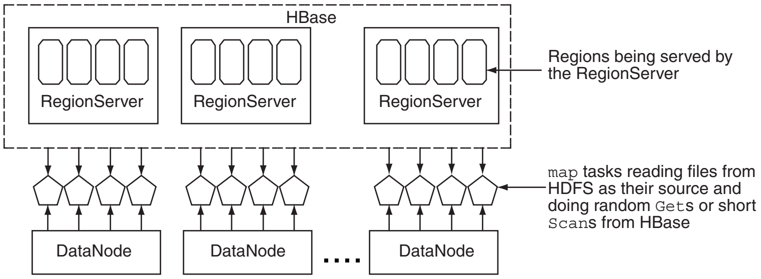

HBase
RegionServer
RegionServer
RegionServer
Regions being served by the RegionServer
map tasks reading files from HDFS as their source and doing random Gets or short Scans from HBase
DataNode
DataNode
....
DataNode

Figure 3.14 Using HBase as a lookup store for the map tasks to do a map-side join

There's a lot more to distributed joins than we've covered in this section. They're so common that Hadoop ships with a contrib JAR called `hadoop-datajoin` to make things easier. You should now have enough context to make good use of it and also take advantage of HBase for other MapReduce optimizations.

3.5 Putting it all together

Now you see the full power of Hadoop MapReduce. The JobTracker distributes work across all the TaskTrackers in the cluster according to optimal resource utilization. If any of those nodes fails, another machine is staged and ready to pick up the computation and ensure job success.

Idempotent operations

Hadoop MapReduce assumes your `map` and `reduce` tasks are idempotent. This means the `map` and `reduce` tasks can be run any number of times with the same input and produce the same output state. This allows MapReduce to provide fault tolerance in job execution and also take maximum advantage of cluster processing power. You must take care, then, when performing stateful operations. HBase's `Increment` command is an example of such a stateful operation.

For example, suppose you implement a row-counting MapReduce job that maps over every key in the table and increments a cell value. When the job is run, the JobTracker spawns 100 mappers, each responsible for 1,000 rows. While the job is running, a disk drive fails on one of the TaskTracker nodes. This causes the `map` task to fail, and Hadoop assigns that task to another node. Before failure, 750 of the keys were counted and incremented. When the new instance takes up that task, it starts again at the beginning of the key range. Those 750 rows are counted twice.

Instead of incrementing the counter in the mapper, a better approach is to emit `["count",1]` pairs from each mapper. Failed tasks are recovered, and their output isn't double-counted. Sum the pairs in a reducer, and write out a single value from there. This also avoids an unduly high burden being applied to the single machine hosting the incremented cell.

Another thing to note is a feature called *speculative execution*. When certain tasks are running more slowly than others and resources are available on the cluster, Hadoop schedules extra copies of the task and lets them compete. The moment any one of the copies finishes, it kills the remaining ones. This feature can be enabled/disabled through the Hadoop configuration and should be disabled if the MapReduce jobs are designed to interact with HBase.

This section provides a complete example of consuming HBase from a MapReduce application. Please keep in mind that running MapReduce jobs on an HBase cluster creates a significant burden on the cluster. You don't want to run MapReduce jobs on the same cluster that serves your low-latency queries, at least not when you expect to maintain OLTP-style service-level agreements (SLAs)! Your online access will suffer while the MapReduce jobs run. As food for thought, consider this: don't even run a JobTracker or TaskTrackers on your HBase cluster. Unless you absolutely must, leave the resources consumed by those processes for HBase.

3.5.1 Writing a MapReduce application

HBase is running on top of Hadoop, specifically the HDFS. Data in HBase is partitioned and replicated like any other data in the HDFS. That means running a MapReduce program over data stored in HBase has all the same advantages as a regular MapReduce program. This is why your MapReduce calculation can execute the same HBase scan as the multithreaded example and attain far greater throughput. In the MapReduce application, the scan is executing simultaneously on multiple nodes. This removes the bottleneck of all data moving through a single machine. If you're running MapReduce on the same cluster that's running HBase, it's also taking advantage of any collocation that might be available. Putting it all together, the Shakespearean counting example looks like the following listing.

Listing 3.1 A Shakespearean twit counter

```
package HBaseIA.TwitBase.mapreduce;

//...

public class CountShakespeare {

  public static class Map
    extends TableMapper<Text, LongWritable> {

    public static enum Counters {ROWS, SHAKESPEAREAN};

    private boolean containsShakespeare(String msg) {
      //...
    }

    @Override
    protected void map(
        ImmutableBytesWritable rowkey,
        Result result,
        Context context) {
      byte[] b = result.getColumnLatest(
                          TwitsDAO.TWITS_FAM,
                          TwitsDAO.TWIT_COL).getValue();
      String msg = Bytes.toString(b);
      if (msg != null && !msg.isEmpty())
        context.getCounter(Counters.ROWS).increment(1);
      if (containsShakespeare(msg))
        context.getCounter(Counters.SHAKESPEAREAN).increment(1);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = HBaseConfiguration.create();
    Job job = new Job(conf, "TwitBase Shakespeare counter");
    job.setJarByClass(CountShakespeare.class);
```

Import details omitted

Natural language processing (NLP) magic happens here

Counters are a cheap way to collect metrics in Hadoop jobs

```
    Scan scan = new Scan();
    scan.addColumn(TwitsDAO.TWITS_FAM, TwitsDAO.TWIT_COL);
    TableMapReduceUtil.initTableMapperJob(
      Bytes.toString(TwitsDAO.TABLE_NAME),
      scan,
      Map.class,
      ImmutableBytesWritable.class,
      Result.class,
      job);

    job.setOutputFormatClass(NullOutputFormat.class);
    job.setNumReduceTasks(0);
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

Just like scan executed in multithreaded example

CountShakespeare is pretty simple; it packages a Mapper implementation and a main method. It also takes advantage of the HBase-specific MapReduce helper class TableMapper and the TableMapReduceUtil utility class that we talked about earlier in the chapter. Also notice the lack of a reducer. This example doesn't need to perform additional computation in the reduce phase. Instead, map output is collected via job counters.

3.5.2 Running a MapReduce application

Would you like to see what it looks like to run a MapReduce job? We thought so. Start by populating TwitBase with a little data. These two commands load 100 users and then load 100 twits for each of those users:

```
$ java -cp target/twitbase-1.0.0.jar \
  HBaseIA.TwitBase.LoadUsers 100
$ java -cp target/twitbase-1.0.0.jar \
  HBaseIA.TwitBase.LoadTwits 100
```

Now that you have some data, you can run the CountShakespeare application over it:

```
$ java -cp target/twitbase-1.0.0.jar \
  HBaseIA.TwitBase.mapreduce.CountShakespeare
...
19:56:42 INFO mapred.JobClient: Running job: job_local_0001
19:56:43 INFO mapred.JobClient:  map 0% reduce 0%
...
19:56:46 INFO mapred.JobClient:  map 100% reduce 0%
19:56:46 INFO mapred.JobClient: Job complete: job_local_0001
19:56:46 INFO mapred.JobClient: Counters: 11
19:56:46 INFO mapred.JobClient: CountShakespeare$Map$Counters
19:56:46 INFO mapred.JobClient:     ROWS=9695
19:56:46 INFO mapred.JobClient:     SHAKESPEAREAN=4743
...
```

According to our proprietary algorithm for Shakespearean reference analysis, just under 50% of the data alludes to Shakespeare!

Counters are fun and all, but what about writing back to HBase? We've developed a similar algorithm specifically for detecting references to *Hamlet*. The mapper is similar to the Shakespearean example, except that its `[k2,v2]` output types are `[ImmutableBytesWritable,Put]`—basically, HBase rowkey and an instance of the `Put` command you learned in the previous chapter. Here's the reducer code:

```
public static class Reduce
  extends TableReducer<
          ImmutableBytesWritable,
          Put,
          ImmutableBytesWritable> {

  @Override
  protected void reduce(
      ImmutableBytesWritable rowkey,
      Iterable<Put> values,
      Context context) {
    Iterator<Put> i = values.iterator();
    if (i.hasNext()) {
      context.write(rowkey, i.next());
    }
  }
}
```

There's not much to it. The reducer implementation accepts `[k2,{v2}]`, the rowkey and a list of `Put`s as input. In this case, each `Put` is setting the `info:hamlet_tag` column to `true`. A `Put` need only be executed once for each user, so only the first is emitted to the output context object. `[k3,v3]` tuples produced are also of type `[ImmutableBytesWritable,Put]`. You let the Hadoop machinery handle execution of the `Put`s to keep the `reduce` implementation idempotent.

3.6 Availability and reliability at scale

You'll often hear the terms *scalable*, *available*, and *reliable* in the context of distributed systems. In our opinion, these aren't absolute, definite qualities of any system, but a set of parameters that can have varied values. In other words, different systems scale to different sizes and are available and reliable in certain scenarios but not others. These properties are a function of the architectural choices that the systems make. This takes us into the domain of the CAP theorem,[2] which always makes for an interesting discussion and a fascinating read.[3] Different people have their own views about it,[4] and we'd prefer not to go into a lot of detail and get academic about what the CAP theorem means for various database systems. Let's instead jump into understanding what *availability* and *reliability* mean in the context of HBase and how it achieves them. These properties are useful to understand from the point of view of building your application so that you as an application developer can understand what you can expect from HBase as a back-end data store and how that affects your SLAs.

[2] CAP theorem: http://en.wikipedia.org/wiki/CAP_theorem.

[3] Read more on the CAP theorem in Henry Robinson's "CAP Confusion: Problems with 'partition tolerance,'" Cloudera, http://mng.bz/4673.

[4] Learn how the CAP theorem is incomplete in Daniel Abadi's "Problems with CAP, and Yahoo's little known NoSQL system," DBMS Musings, http://mng.bz/j01r.

AVAILABILITY

Availability in the context of HBase can be defined as the ability of the system to handle failures. The most common failures cause one or more nodes in the HBase cluster to fall off the cluster and stop serving requests. This could be because of hardware on the node failing or the software acting up for some reason. Any such failure can be considered a network partition between that node and the rest of the cluster.

When a RegionServer becomes unreachable for some reason, the data it was serving needs to instead be served by some other RegionServer. HBase can do that and keep its availability high. But if there is a network partition and the HBase masters are separated from the cluster or the ZooKeepers are separated from the cluster, the slaves can't do much on their own. This goes back to what we said earlier: availability is best defined by the kind of failures a system can handle and the kind it can't. It isn't a binary property, but instead one with various degrees.

Higher availability can be achieved through defensive deployment schemes. For instance, if you have multiple masters, keep them in different racks. Deployment is covered in detail in chapter 10.

RELIABILITY AND DURABILITY

Reliability is a general term used in the context of a database system and can be thought of as a combination of data durability and performance guarantees in most cases. For the purpose of this section, let's examine the data durability aspect of HBase. Data durability, as you can imagine, is important when you're building applications atop a database. `/dev/null` has the fastest write performance, but you can't do much with the data once you've written it to `/dev/null`. HBase, on the other hand, has certain guarantees in terms of data durability by virtue of the system architecture.

3.6.1 HDFS as the underlying storage

HBase assumes two properties of the underlying storage that help it achieve the availability and reliability it offers to its clients.

SINGLE NAMESPACE

HBase stores its data on a single file system. It assumes all the RegionServers have access to that file system across the entire cluster. The file system exposes a single namespace to all the RegionServers in the cluster. The data visible to and written by one RegionServer is available to all other RegionServers. This allows HBase to make availability guarantees. If a RegionServer goes down, any other RegionServer can read the data from the underlying file system and start serving the regions that the first RegionServer was serving (figure 3.15).

At this point, you may be thinking that you could have a network-attached storage (NAS) that was mounted on all the servers and store the data on that. That's theoretically doable, but there are implications to every design and implementation choice. Having a NAS that all the servers read/write to means your disk I/O will be bottlenecked by the interlink between the cluster and the NAS. You can have fat interlinks, but they will still limit the amount you can scale to. HBase made a design choice to use

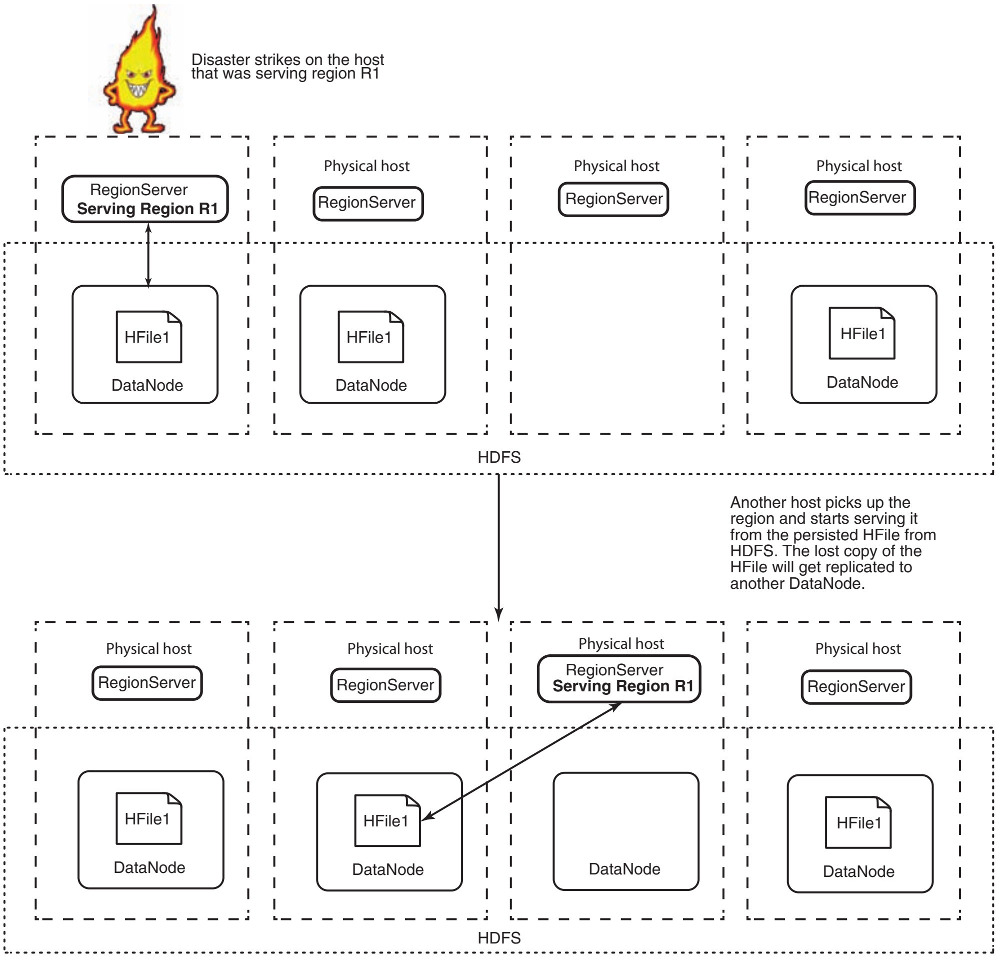


Figure 3.15 **If a RegionServer fails for some reason (such as a Java process dying or the entire physical node catching fire), a different RegionServer picks up the regions the first one was serving and begins serving them. This is enabled by the fact that HDFS provides a single namespace to all the RegionServers, and any of them can access the persisted files from any other.**

distributed file systems instead and was built tightly coupled with HDFS. HDFS provides HBase with a single namespace, and the DataNodes and RegionServers are collocated in most clusters. Collocating these two processes helps in that RegionServers can read and write to the local DataNode, thereby saving network I/O whenever possible. There is still network I/O, but this optimization reduces the costs.

You're currently using a standalone HBase instance for the TwitBase application. Standalone HBase isn't backed by HDFS. It's writing all data onto the local file system. Chapter 9 goes into details of deploying HBase in a fully distributed manner, backed by HDFS. When you do that, you'll configure HBase to write to HDFS in a prespecified location, which is configured by the parameter `hbase.rootdir`. In standalone mode, this is pointing to the default value, `file:///tmp/hbase-${user.name}/hbase`.

RELIABILITY AND FAILURE RESISTANCE

HBase assumes that the data it persists on the underlying storage system will be accessible even in the face of failures. If a server running the RegionServer goes down, other RegionServers should be able to take up the regions that were assigned to that server and begin serving requests. The assumption is that the server going down won't cause data loss on the underlying storage. A distributed file system like HDFS achieves this property by replicating the data and keeping multiple copies of it. At the same time, the performance of the underlying storage should not be impacted greatly by the loss of a small percentage of its member servers.

Theoretically, HBase could run on top of any file system that provides these properties. But HBase is tightly coupled with HDFS and has been during the course of its development. Apart from being able to withstand failures, HDFS provides certain write semantics that HBase uses to provide durability guarantees for every byte you write to it.

3.7 Summary

We covered quite a bit of ground this chapter, much of if at an elementary level. There's a lot more going on in Hadoop than we can cover with a single chapter. You should now have a basic understanding of Hadoop and how HBase uses it. In practice, this relationship with Hadoop provides an HBase deployment with many advantages. Here's an overview of what we discussed.

HBase is a database built on Hadoop. It depends on Hadoop for both *data access* and *data reliability*. Whereas HBase is an *online* system driven by *low latency*, Hadoop is an *offline* system optimized for *throughput*. These complementary concerns make for a powerful, flexible data platform for building horizontally scalable data applications.

Hadoop MapReduce is a distributed computation framework providing data access. It's a *fault-tolerant*, *batch-oriented* computing model. MapReduce programs are written by composing `map` and `reduce` operations into *Jobs*. Individual *tasks* are assumed to be *idempotent*. MapReduce takes advantage of the HDFS by assigning tasks to blocks on the file system and *distributing the computation to the data*. This allows for *highly parallel* programs with minimal distribution overhead.

HBase is designed for MapReduce interaction; it provides a `TableMapper` and a `TableReducer` to ease implementation of MapReduce applications. The `TableMapper` allows your MapReduce application to easily read data directly out of HBase. The `TableReducer` makes it easy to write data back to HBase from MapReduce. It's also possible to interact with the HBase key-value API from within the Map and Reduce Steps. This is helpful for situations where all your tasks need random access to the same data. It's commonly used for implementing distributed map-side joins.

If you're curious to learn more about how Hadoop works or investigate additional techniques for MapReduce, *Hadoop: The Definitive Guide* by Tom White (O'Reilly, 2009) and *Hadoop in Action* by Chuck Lam (Manning, 2010) are both great references.

data. This code is *run in parallel* across all the RegionServers. This transforms an HBase cluster from horizontally scalable storage to a highly capable, distributed, data-storage and *-processing* system.

> **WARNING** Coprocessors are a brand-new feature in HBase and are *untested* in production deployments. Their integration with HBase internals is *extremely invasive*. Think of them as akin to Linux kernel modules or RDBMS stored procedures implemented in C. Writing an observer coprocessor is tricky to get right, and such a coprocessor can be extremely difficult to debug when running at scale. Unlike client-side bugs, a buggy coprocessor *will* take down your cluster. The HBase community is still working out exactly how to use coprocessors effectively.[1] Caution is advised.

In this chapter, we'll introduce you to the two types of coprocessors and show examples of how to use each one. We hope this will open your mind to the possibilities so you'll be able to use coprocessors in your own applications. You never know: maybe you can be the one to write the blog post describing the canonical coprocessor example! Please make it more interesting than WordCount.

> **More inspiration from Google**
>
> As with much of the rest of the Hadoop ecosystem, coprocessors come to the open source community by way of Google. The idea for HBase coprocessors came from two slides[2] in a talk presented in 2009. Coprocessors are cited as crucial for a number of horizontally scalable, low-latency operations. These operations include machine translation, full-text queries, and scalable metadata management.

5.1 The two kinds of coprocessors

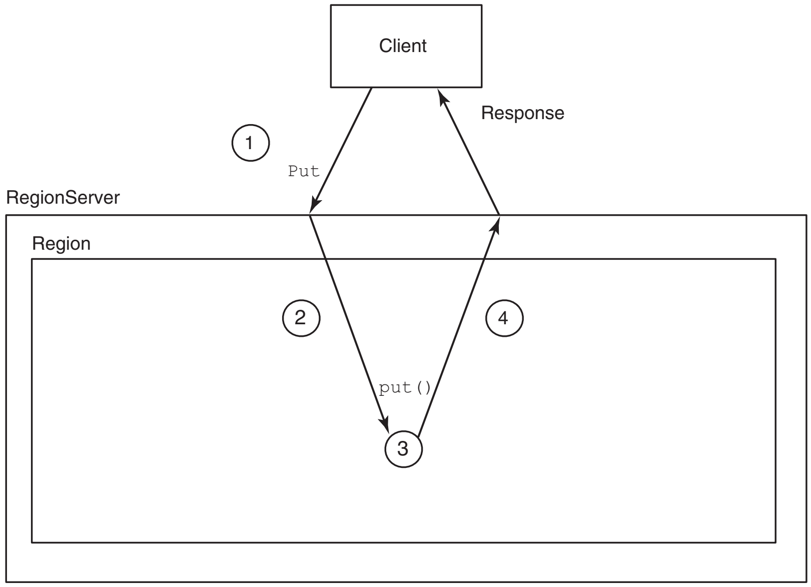
Coprocessors come in two flavors: *observers* and *endpoints*. Each serves a different purpose and is implemented according to its own API. Observers allow the cluster to behave differently during normal client operations. Endpoints allow you to extend the cluster's capabilities, exposing new operations to client applications.

5.1.1 Observer coprocessors

To understand observer coprocessors, it helps to understand the lifecycle of a request. A request starts with the client, creating a request object and invoking the appropriate method on the `HTableInterface` implementation. For example, a `Put` instance is created and the `put()` method called. The HBase client resolves the RegionServer that should receive the `Put` based on the rowkey and makes the RPC call. The RegionServer receives the `Put` and delegates it to the appropriate region. The region handles the

[1] The HBase blog has an excellent overview of coprocessors that is appended periodically with new details of current and future work: Mingjie Lai, Eugene Koontz, and Andrew Purtell, "Coprocessor Introduction," http://mng.bz/TzuY.

[2] Jeff Dean, "Designs, Lessons and Advice from Building Large Distributed Systems," LADIS '09, http://mng.bz/U2DB, pages 66-67.



1 Client sends `Put` request.
2 Request is dispatched to appropriate RegionServer and region.
3 The region receives the `put()`, processes it, and constructs a response.
4 The final result is returned to the client.

Figure 5.1 The lifecycle of a request. A `Put` request dispatched from the client results directly in a `put()` call on the region.

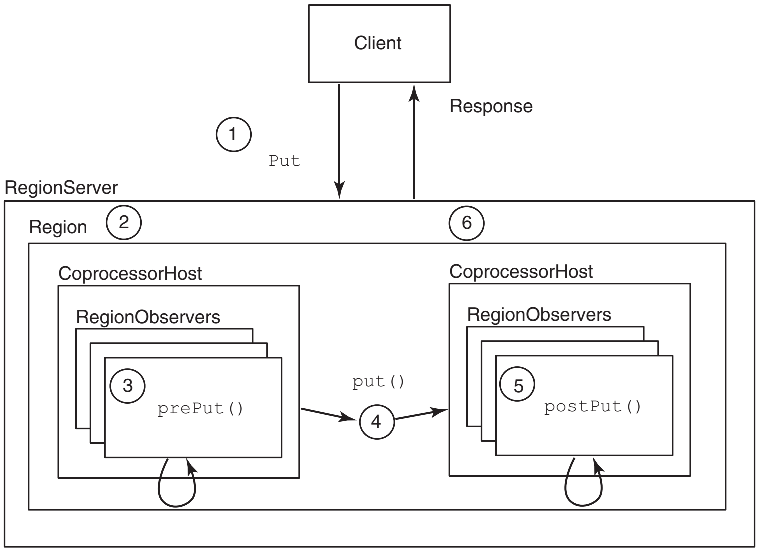
request and constructs a response, which is passed back to the client. Figure 5.1 illustrates this path.

Observers sit between the client and HBase, modifying data access as it happens. You can run an observer after every `Get` command, modifying the result returned to the client. Or you can run an observer after a `Put` command, performing manipulation on the data that a client writes to HBase before it's persisted. You can think of observer coprocessors as analogous to *triggers* from a relational database or to *advice* from aspect-oriented programming (AOP). Multiple observers can be registered simultaneously; they're executed in priority order. The `CoprocessorHost` class manages observer registration and execution on behalf of the region. Figure 5.2 illustrates a RegionObserver intercepting a `Put` command.

A word of caution

Bear in mind that coprocessors are executed in the same process space as the RegionServer. This means code in a coprocessor has full rights and privileges of the HBase user process on the server. It also means a buggy coprocessor can potentially crash the process. No isolation guarantees are in place at this point. You can follow along with the efforts to resolve this potential issue by tracking the JIRA ticket.[3]

[3] "[Coprocessors] Generic external process host," Apache Software Foundation, http://mng.bz/9uOy.



1 Client sends `Put` request.

2 Request is dispatched to appropriate RegionServer and region.

3 CoprocessorHost intercepts the request and invoices `prePut()` on each RegionObserver registered on the table.

4 Unless interrupted by a `prePut()`, the request continues to region and is processed normally.

5 The result produced by the region is once again intercepted by the CoprocessorHost. This time `postPut()` is called on each registered RegionObserver.

6 Assuming no `postPut()` interrupts the response, the final result is returned to the client.

Figure 5.2 A RegionObserver in the wild. Instead of calling `put()` directly, the region calls `prePut()` and `postPut()` on all registered RegionObservers, one after the next. Each has a chance to modify or interrupt the operation before a response is returned to the client.

As of HBase version 0.92, three kinds of observers are available:

- RegionObserver—This observer hooks into the stages of data access and manipulation. All of the standard data-manipulation commands can be intercepted with both pre- and post-hooks. It also exposes pre- and post-hooks for internal region operations such as flushing the MemStore and splitting the region. The RegionObserver runs on the region; thus there can be multiple RegionObservers running on the same RegionServer. Register RegionObservers through either schema updates or the `hbase.coprocessor.region.classes` configuration property.
- WALObserver—The write-ahead log (WAL) also supports an observer coprocessor. The only available hooks are pre- and post-WAL write events. Unlike the RegionObserver, WALObservers run in the context of a RegionServer. Register

WALObservers through either schema updates or the `hbase.coprocessor.wal.classes` configuration property.

- MasterObserver—For hooking into DDL events, such as table creation or schema modifications, HBase provides the MasterObserver. For example, you can use the `postDeleteTable()` hook to also delete secondary indexes when the primary table is deleted. This observer runs on the Master node. Register MasterObservers through the `hbase.coprocessor.master.classes` configuration property.

5.1.2 Endpoint Coprocessors

Endpoints are a generic extension to HBase. When an endpoint is installed on your cluster, it extends the HBase RPC protocol, exposing new methods to client applications. Just like observers, endpoints execute on the RegionServers, right next to your data.

Endpoint coprocessors are similar to stored procedures in other database engines. From the client's perspective, invoking an endpoint coprocessor is similar to invoking any other HBase command, except that the functionality is based on the custom code that defines the coprocessor. The request object is created, it's passed to the `HTableInterface` to execute on the cluster, and the results are collected. This arbitrary code can do anything for which you can write code in Java.

At their most basic, endpoints can be used to implement *scatter-gather algorithms*. HBase ships with an Aggregate example: an endpoint that computes simple aggregates like sum and average. `AggregateImplementation` instances calculate partial results on the nodes hosting data, and the `AggregationClient` computes the final result in the client process. Figure 5.3 illustrates the Aggregation example in action.

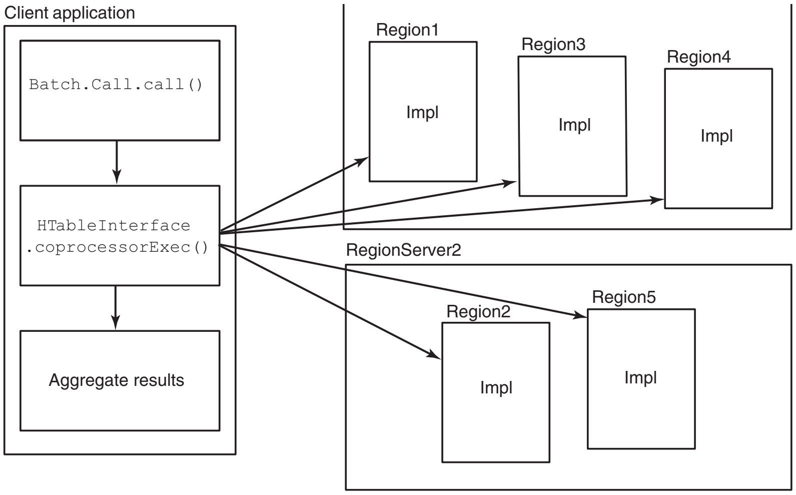


Figure 5.3 An endpoint coprocessor at work. The regions deploy an implementation of the interface consumed by the client. An instance of `Batch.Call` encapsulates method invocation, and the `coprocessorExec()` method handles distributed invocation. After each request completes, results are returned to the client and aggregated.

appendix B
More about the workings of HDFS

Hadoop Distributed File System (HDFS) is the underlying distributed file system that is the most common choice for running HBase. Many HBase features depend on the semantics of the HDFS to function properly. For this reason, it's important to understand a little about how the HDFS works. In order to understand the inner working of HDFS, you first need to understand what a distributed file system is. Ordinarily, the concepts at play in the inner workings of a distributed file system can consume an entire semester's work for a graduate class. But in the context of this appendix, we'll briefly introduce the concept and then discuss the details you need to know about HDFS.

B.1 Distributed file systems

Traditionally, an individual computer could handle the amount of data that people wanted to store and process in the context of a given application. The computer might have multiple disks, and that sufficed for the most part—until the recent explosion of data. With more data to store and process than a single computer could handle, we somehow needed to combine the power of multiple computers to solve these new storage and compute problems. Such systems in which a network of computers (also sometimes referred to as a *cluster*) work together as a single system to solve a certain problem are called *distributed systems*. As the name suggests, the work is distributed across the participating computers.

Distributed file systems are a subset of distributed systems. The problem they solve is data storage. In other words, they're storage systems that span multiple computers.

TIP The data stored in these file systems is spread across the different nodes automatically: you don't have to worry about manually deciding which node to put what data on. If you're curious to know more about the placement strategy of HDFS, the best way to learn is to dive into the HDFS code.

Distributed file systems provide the scale required to store and process the vast amount of data that is being generated on the web and elsewhere. Providing such scale is challenging because the number of moving parts causes more susceptibility to failure. In large distributed systems, failure is a norm, and that must be taken into account while designing the systems.

In the sections that follow, we'll examine the challenges of designing a distributed file system and how HDFS addresses them. Specifically, you'll learn how HDFS achieves scale by separating metadata and the contents of files. Then, we'll explain the consistency model of HDFS by going into details of the HDFS read and write paths, followed by a discussion of how HDFS handles various failure scenarios. We'll then wrap up by examining how files are split and stored across multiple nodes that make up HDFS.

Let's get started with HDFS's primary components—NameNode and DataNode—and learn how scalability is achieved by separating metadata and data.

B.2 Separating metadata and data: NameNode and DataNode

Every file that you want to store on a file system has metadata attached to it. For instance, the logs coming in from your web servers are all individual files. The metadata includes things like filename, inode number, block location, and so on; the data is the actual content of the file.

In traditional file systems, metadata and data are stored together on the same machine because the file systems never span beyond that. When a client wants to perform any operation on the file and it needs the metadata, it interacts with that single machine and gives it the instructions to perform the operation. Everything happens at a single point. For instance, suppose you have a file Web01.log stored on the disk mounted at /mydisk on your *nix system. To access this file, the client application only needs to talk to the particular disk (through the operating system, of course) to get the metadata as well as the contents of the file. The only way this model can work with data that's more than a single system can handle is to make the client aware of how you distribute the data among the different disks, which makes the client stateful and difficult to maintain.

Stateful clients are even more complicated to manage as the number grows, because they have to share the state information with each other. For instance, one client may write a file on one machine. The other client needs the file location in order to access the file later, and has to get the information from the first client. As you can see, this can quickly become unwieldy in large systems and is hard to scale.

In order to build a distributed file system where the clients remain simple and ignorant of each other's activities, the metadata needs to be maintained outside of the clients. The easiest way to do this is to have the file system itself manage the metadata. But storing both metadata and data together at a single location doesn't work either, as we discussed earlier. One way to solve this is to dedicate one or more machines to hold the metadata and have the rest of the machines store the file contents. HDFS's design is based on this concept. It has two components: NameNode and DataNode. The metadata is stored in the NameNode. The data, on the other hand, is stored on a cluster of DataNodes. The NameNode not only manages the metadata for the content stored in HDFS but also keeps account of things like which nodes are part of the cluster and how many copies of a particular file exist. It also decides what needs to be done when nodes fall out of the cluster and replicas are lost.

This is the first time we've mentioned the word *replica*. We'll talk about it in detail later—for now, all you need to know is that every piece of data stored in HDFS has multiple copies residing on different servers. The NameNode essentially is the HDFS Master, and the DataNodes are the Slaves.

B.3 HDFS write path

Let's go back to the example with Web01.log stored on the disk mounted at /mydisk. Suppose you have a large distributed file system at your disposal, and you want to store that file on it. It's hard to justify the cost of all those machines otherwise, isn't it? To store data in HDFS, you have various options. The underlying operations that happen when you write data are the same regardless of what interface you use to write the data (Java API, Hadoop command-line client, and so on).

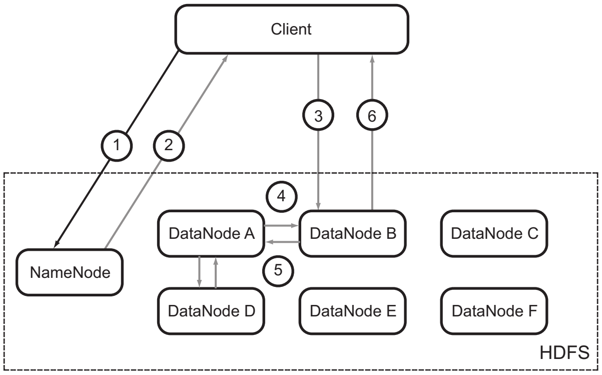
> NOTE If you're like us and want to play with the system as you learn about the concepts, we encourage you to do so. But it's not required for this section, and you don't need to do so to understand the concepts.

Let's say you're using the Hadoop command-line client and you want to copy a file Web01.log to HDFS. You write the following command:

```
$ hadoop fs -copyFromLocal /home/me/Web01.log /MyFirstDirectory/
```

It's important that you understand what happens when you write this command. We'll go over the write process step by step while referring to figures B.1–B.4. Just to remind you, the client is simple and doesn't know anything about the internals of HDFS and how the data is distributed.

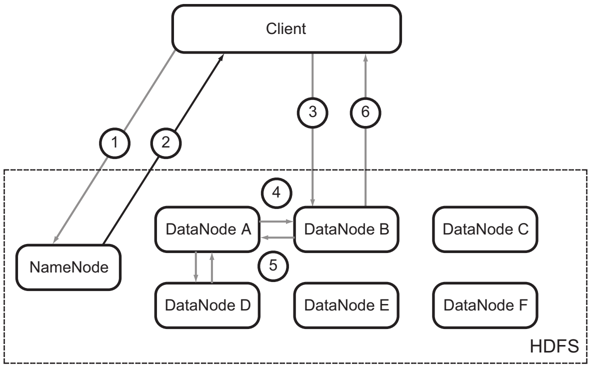
The client does, however, know from the configuration files which node is the NameNode. It sends a request to the NameNode saying that it wants to write a file Web01.log to HDFS (figure B.1). As you know, the NameNode is responsible for managing the metadata about everything stored in HDFS. The NameNode acknowledges the client's request and internally makes a note about the filename and a set of DataNodes that will store the file. It stores this information in a file-allocation table in its memory.



Client -> NameNode: "I want to write file Web01.log."

Figure B.1 Write operation: client's communication with the NameNode

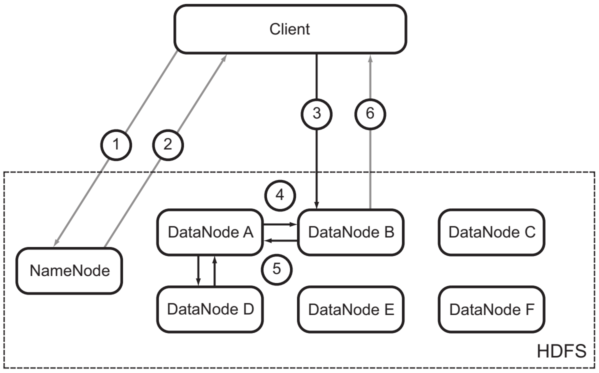
It then sends this information back to the client (figure B.2). The client is now aware of the DataNodes to which it has to send the contents of Web01.log.



NameNode -> Client: "Absolutely, let's get you started. You can write to DataNodes A, B, and D. B will be the DataNode you should contact directly."

Figure B.2 Write operation: NameNode acknowledges the write operation and sends back a DataNode list

The next step for it is to send the contents over to the DataNodes (figure B.3). The primary DataNode streams the contents of the file synchronously to other DataNodes that will hold the replicas of this particular file. Once all the DataNodes that have to



Client -> DataNode B: "Here is the content of Web01.log. Please keep a copy and send a copy each to DataNode A and D."

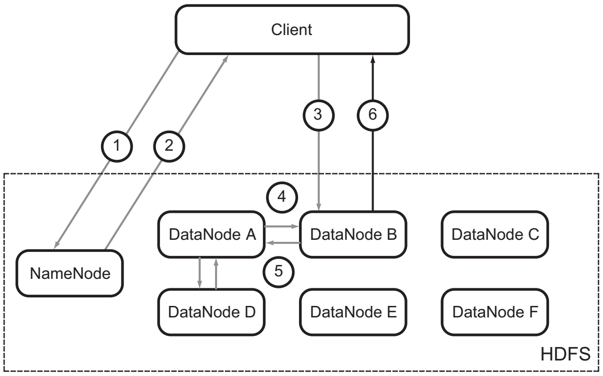
DataNode B -> DataNode A: "Here's file Web01.log. Please keep a copy and send a copy to DataNode D."

DataNode A -> DataNode D: "Here's file Web01.log. Please keep a copy of it with you."

DataNode D -> DataNode A: "Ack."

DataNode A -> DataNode B: "Ack."

Figure B.3 Write operation: client sends the file contents to the DataNodes



DataNode B -> Client: "Ack. Your write is now complete."

Figure B.4 **Write operation: DataNode acknowledges completion of the write operation**

hold replicas of the file have the contents in memory, they send an acknowledgement to the DataNode that the client connected to. The data is persisted onto disk asynchronously later. We'll cover the topic of replication in detail later in the appendix.

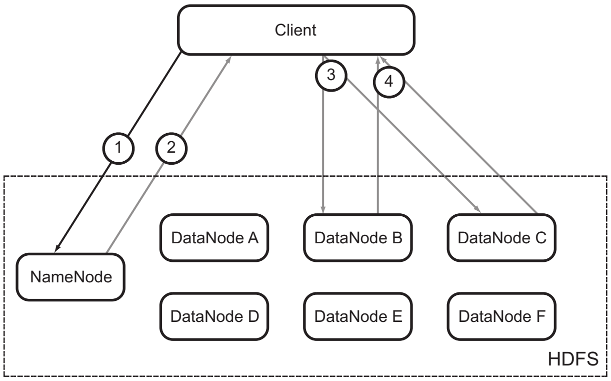
The primary DataNode sends a confirmation to the client that the file has been written to HDFS (figure B.4). At the end of this process, the file is considered written to HDFS and the write operation is complete.

Note that the file is still in the DataNodes' main memory and hasn't yet been persisted to disk. This is done for performance reasons: committing all replicas to disk would increase the time taken to complete a write operation. Once the data goes into the main memory, the DataNode persists it to disk as soon as it can. The write operation doesn't block on it.

In distributed file systems, one of the challenges is *consistency*. In other words, how do you make sure the view of the data residing on the system is consistent across all nodes? Because all nodes store data independently and don't typically communicate with each other, there has to be a way to make sure all nodes contain the same data. For example, when a client wants to read file Web01.log, it should be able to read exactly the same data from all the DataNodes. Looking back at the write path, notice that the data isn't considered written unless all DataNodes that will hold that data have acknowledged that they have a copy of it. This means all the DataNodes that are supposed to hold replicas of a given set of data have exactly the same contents before the write completes—in other words, consistency is accomplished during the write phase. A client attempting to read the data will get the same bytes back from whichever DataNode it chooses to read from.

B.4 HDFS read path

Now you know how a file is written into HDFS. It would be strange to have a system in which you could write all you want but not read anything back. Fortunately, HDFS isn't one of those. Reading a file from HDFS is as easy as writing a file. Let's see how the file you wrote earlier is read back when you want to see its contents.



Client -> NameNode: "I want to read file Web01.log."

Figure B.5 Read operation: client's communication with the NameNode

Once again, the underlying process that takes place while reading a file is independent of the interface used. If you're using the command-line client, you write the following command to copy the file to your local file system so you can use your favorite editor to open it:

```
$ hadoop fs -copyToLocal /MyFirstDirectory/Web01.log /home/me/
```

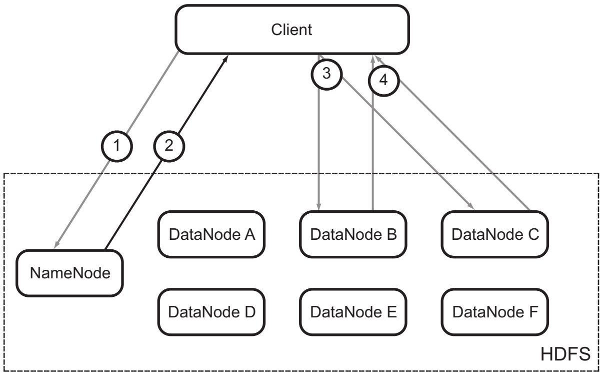
Let's look at what happens when you run this command. The client asks the NameNode where it should read the file from (figure B.5).

The NameNode sends back block information to the client (figure B.6).

The block information contains the IP addresses of the DataNodes that have copies of the file and the block ID that the DataNode needs to find the block on its local storage. These IDs are unique for all blocks, and this is the only information the DataNodes need in order to identify the block on their local storage. The client examines this information, approaches the relevant DataNodes, and asks for the blocks (figure B.7).

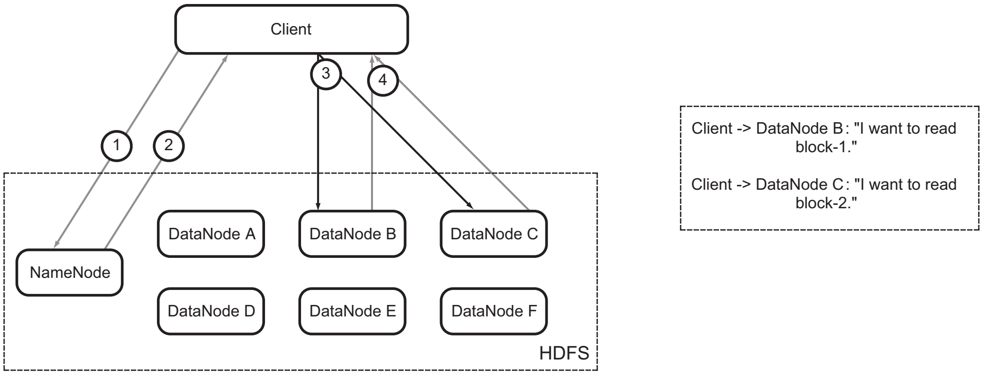
The DataNodes serve the contents back to the client (figure B.8), and the connection is closed. That completes the read step.

This is the first time we've mentioned the word *block* in the context of files in HDFS. In order to understand the read process, consider that a file is made up of blocks that are stored on the DataNodes. We'll dig deeper into this concept later in the appendix.



NameNode -> Client: "Absolutely, let's get you started. You can read block-1 from DataNode B and block-2 from DataNode C."

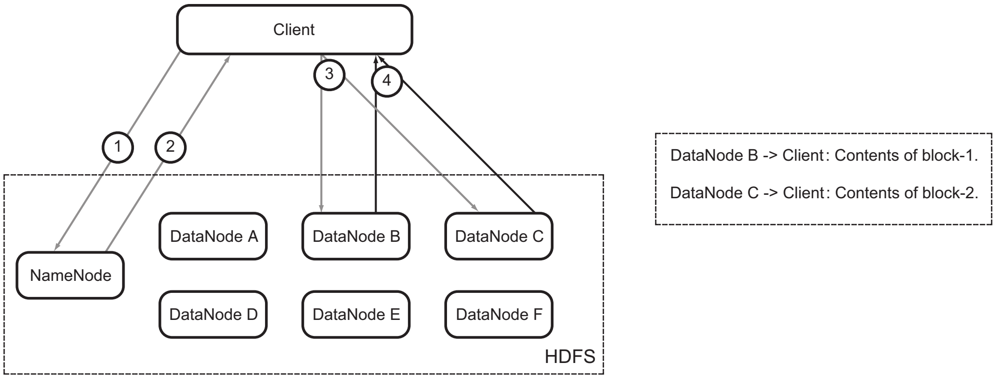
Figure B.6 Read operation: NameNode acknowledges the read and sends back block information to the client



Client -> DataNode B: "I want to read block-1."

Client -> DataNode C: "I want to read block-2."

Figure B.7 Read operation: client contacts the relevant DataNodes and asks for the contents of the blocks



DataNode B -> Client: Contents of block-1.

DataNode C -> Client: Contents of block-2.

Figure B.8 Read operation: DataNodes serve the block contents to the client. This completes the read step.

Note that the client gets separate blocks for a file in parallel from different DataNodes. The client joins these blocks to make the full file. The logic for this is in the client library, and the user who's writing the code doesn't have to do anything manually. It's all handled under the hood. See how easy it is?

B.5 Resilience to hardware failures via replication

In large distributed systems, disk and network failures are commonplace. In case of a failure, the system is expected to function normally without losing data. Let's examine the importance of replication and how HDFS handles failure scenarios. (You may be thinking that we should have covered this earlier—hang on for a bit and it will make sense.)

When everything is working fine, DataNodes periodically send heartbeat messages to the NameNode (by default, every three seconds). If the NameNode doesn't receive a heartbeat for a predefined time period (by default, 10 minutes), it considers the DataNode to have failed, removes it from the cluster, and starts a process to recover from the failure. DataNodes can fall out of the cluster for various reasons—disk failures, motherboard failures, power outages, and network failures. The way HDFS recovers from each of these is the same.

For HDFS, losing a DataNode means losing replicas of the blocks stored on that disk. Given that there is always more than one replica at any point in time (the default is three), failures don't lead to data loss. When a disk fails, HDFS detects that the blocks that were stored on that disk are under-replicated and proactively creates the number of replicas required to reach a fully replicated state.

There could be a situation in which multiple disks failed together and all replicas of a block were lost, in which case HDFS would lose data. For instance, it's theoretically possible to lose all the nodes that are holding replicas of a given file because there is a network partition. It's also possible for a power outage to take down entire racks. But such situations are rare; and when systems are designed that have to store mission-critical data that absolutely can't be lost, measures are taken to safeguard against such failures—for instances, multiple clusters in different data centers for backups.

From the context of HBase as a system, this means HBase doesn't have to worry about replicating the data that is written to it. This is an important factor and affects the consistency that HBase offers to its clients.

B.6 Splitting files across multiple DataNodes

Earlier, we examined the HDFS write path. We said that files are replicated three ways before a write is committed, but there's a little more to that. Files in HDFS are broken into blocks, typically 64–128 MB each, and each block is written onto the file system. Different blocks belonging to the same file don't necessarily reside on the same DataNode. In fact, it's beneficial for different blocks to reside on different DataNodes. Why is that?

When you have a distributed file system that can store large amounts of data, you may want to put large files on it—for example, outputs of large simulations of subatomic particles, like those done by research labs. Sometimes, such files can be larger than the size of a single hard drive. Storing them on a distributed system by breaking them into blocks and spreading them across multiple nodes solves that problem.

There are other benefits of distributing blocks to multiple DataNodes. When you perform computations on these files, you can benefit from reading and processing different parts of the files in parallel.

You may wonder how the files are split into blocks and who determines which DataNodes the various blocks should go to. When the client is writing to HDFS and talks to the NameNode about where it should write the files, the NameNode tells it the DataNodes where it should write the blocks. After every few blocks, the client goes back to the NameNode to get a fresh list of DataNodes to which the next few blocks should be written.

Believe it or not, at this point, you know enough about HDFS to understand how it functions. You may not win an architectural argument with a distributed systems expert, but you'll be able to understand what they're talking about. Winning the argument wasn't the intention of the appendix anyway!