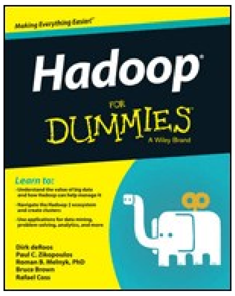


Hadoop for Dummies

by Dirk deRoos et al.

John Wiley & Sons (US). (c) 2014. Copying Prohibited.

Reprinted for Venkata Kiran Polineni, Verizon

venkata.polineni@one.verizon.com

Reprinted with permission as a subscription benefit of **Skillport**,
http://skillport.books24x7.com/

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Chapter 12: Extremely Big Tables—Storing Data in HBase

In This Chapter

- Introducing HBase
- Storing data in HBase
- Looking at the nuts and bolts of HBase
- Taking HBase for a spin
- Interfacing with HBase
- Comparing HBase to relational databases
- Going with a real HBase deployment

Do you remember your first surfing experience on the World Wide Web? You just knew that it was an incredible innovation for the IT industry. Having this vast ocean of knowledge at your fingertips was transformational. Times change, though, and now the Internet is truly just another part of everyday life that many people take for granted. You open your favorite browser and visit a search engine, and — in a matter of seconds — you're learning something new.

In this chapter, we ask you to take a step back and ponder the immensity of the web and, more specifically, how exactly an entity such as Google stores all those references and web pages for your use? If the picture in your mind includes the concept of a database, you're right, but what kind of database? Every database administrator has thought about limits at one time or another. Storing gigabytes (or even terabytes) of data using your database of choice is common, but you may be faced with petabytes of data as Google was when it sought to index the web. The company's strategy was to use BigTable — Google researchers even published an important paper outlining their vision of BigTable in 2006.

You may wonder what all this has to do with the history of HBase. Well, HBase is an implementation of Google's BigTable distributed data storage system (DDSS, for short). After Google's release of the BigTable paper, Powerset, a company focused on building a natural language processing (NLP) search engine for the Internet, became interested in creating its own implementation of BigTable. So when the University of Michigan's Mike Cafarella made his first code drop of HBase to the Apache Open Source community in early 2007, Powerset engineers decided to carry the work forward. By 2008 HBase had become a sub-project of Hadoop and in 2010 HBase became an Apache top-level project. HBase, which has an affinity to Hadoop, is referred to as "the Hadoop database" on its Apache web page. (Don't believe us? Check out the Welcome Apache HBase page at http://hbase.apache.org.)

After you know a bit of the history of HBase, you're on better footing to start understanding what HBase actually does. In subsequent sections of this chapter, you can see how HBase works and why it's vital in the age of big data.

Say Hello to HBase

HBase is a Java implementation of Google's BigTable. Google defines BigTable as a "sparse, distributed, persistent multidimensional sorted map." We're sure that you'll agree that it's quite a concise definition, but that you'll also agree that it's a bit on the complex side. To break down BigTable's complexity a bit, we discuss each attribute in this section.

Sparse

As you might have guessed, the BigTable distributed data storage system was designed to meet the demands of big data. Now, big data applications store lots of data but big data content is also often variable. Imagine a traditional table in a company database storing customer contact information, as shown in Table 12-1.

Table 12-1: Traditional Customer Contact Information Table

Customer ID	*Last Name*	*First Name*	*Middle Name*	*E-mail Address*	*Street Address*
00001	Smith	John	Timothy	`John.Smith@xyz.com`	1 Hadoop Lane, NY 11111
00002	Doe	Jane	NULL	NULL	7 HBase Ave, CA 22222

A company or individual may require a complete data record for each of its customers or constituents. A good example is your doctor, who needs all your contact information in order to provide you with proper care. Other companies or individuals may require only partial contact information or may need to learn that information over time. For example, a customer service company may process phone calls or e-mail messages for service requests. Clients may or may not choose to give service companies all their contact information. However, with each interaction over time, companies may learn more about their clients that will enable them to provide better service — by issuing proactive service alerts, for example.

REMEMBER In this context, *sparse* means that fields in rows can be empty or NULL but that doesn't bring HBase to a screeching halt. HBase can handle the fact that we don't (yet) know Jane Doe's middle name and e-mail address, for example.

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

Here's another example: a database for storing satellite images. It turns out that Google uses BigTable technology to store satellite imagery of the earth. In almost every case, whenever imagery is stored, metadata is also stored with it. The metadata may include the street address of the image or only the latitude and longitude if the image is captured from the wilderness. The metadata is variable in content so some fields will be NULL — and that's OK.

In both examples, the data sets that are collected can be extremely large — especially in the second example. Imagery databases are almost always measured in terabytes or sometimes in petabytes. We've already mentioned that HBase is designed for storing big data, but it's also designed for storing sparse data records at no cost. This concern is crucial when you're using big data applications! Storing a few NULL records over a million rows is wasteful, but try to imagine the waste over a quadrillion rows! Thankfully, this was a key consideration for Google designers and the HBase community. Sparse data is supported with no waste of costly storage space.

And it doesn't stop there. Consider the power of a schema-less data store. Table 12-1 shows you a classic customer contact table. When companies design these tables, they know up front what they want to store. In other words the schema is *fixed*; it's defined even before the first byte of information is stored in the table. Now what if, over time, a new field is needed for a customer? How about a Twitter handle or a new mobile phone number? You're seemingly stuck with a schema that no longer works for you. Well, HBase solves this challenge as well — you can not only skip fields at no cost when you don't have the data, but also dynamically add fields (or *columns* in the HBase vernacular — more on this later) over time without having to redesign the schema or disrupt operations. So you can think of HBase as a schema-less data store; that is, it's fluid — you can add to, subtract from or modify the schema as you go along.

It's Distributed and Persistent

BigTable is a distributed and persistent data store. *Persistent* simply means that the data you store in BigTable (and HBase, for that matter) will persist or remain after your program or session ends. That's pretty straightforward — persistent means that it persists — but you should spend a little more time thinking about *how* the data is persisted. In its BigTable paper, Google described the distributed file system known as Google File System or GFS. It turns out that, just as HBase is an open source implementation of BigTable, HDFS is an open source implementation of GFS. By default, HBase leverages HDFS to persist its data to disk storage. (For more on the mechanics of HDFS, see Chapter 3.) Though other distributed data stores can be used with HBase, the vast majority of HBase installations leverage HDFS. This makes perfect sense given that HBase is the "Hadoop Database" — hey, it's built into the name, for goodness sake.

REMEMBER HDFS is a key enabling technology not only for Hadoop but also for HBase. By storing data in HDFS, HBase offers reliability, availability, seamless scalability, high performance and much more — all on cost effective distributed servers!

It Has a Multidimensional Sorted Map

Starting from the basics, a *map* (also known as an *associative array*) is an abstract collection of key-value pairs, where the key is unique. This definition is crucial to your understanding of HBase because the HBase data model is often described in different ways — often incompletely as a column-oriented store. HBase is, at bottom, a key-value data store where each key is unique — meaning it appears at most once in the HBase data store. Additionally, the map is sorted and multidimensional. The keys are stored in HBase and sorted in byte-lexicographical order. Each value can have multiple versions, which makes the data model multidimensional. By default, data versions are implemented with a timestamp.

Understanding the HBase Data Model

HBase data stores consist of one or more tables, which are indexed by row keys. Data is stored in rows with columns, and rows can have multiple versions. By default, data versioning for rows is implemented with time stamps. Columns are grouped into *column families*, which must be defined up front during table creation. Column families are stored together on disk, which is why HBase is referred to as a column-oriented data store. To show you a practical example, we've altered Table 12-1 to make it conform to an HBase data model — behold the logical view of information in Table 12-2.

Because the data model is critical to understanding HBase, we discuss Table 12-2 in detail in the following five sections.

Table 12-2: Logical View of Customer Contact Information in HBase

Row Key	*Column Family: {Column Qualifier:Version:Value}*
00001	CustomerName: {‘FN’: 1383859182496:‘John’, ‘LN’: 1383859182858:‘Smith’, ‘MN’: 1383859183001:‘Timothy’, ‘MN’: 1383859182915:‘T’} ContactInfo: {‘EA’: 1383859183030:‘John.Smith@xyz.com’, ‘SA’: 1383859183073:‘1 Hadoop Lane, NY 11111’}

00002	CustomerName: {‘FN’: 1383859183103:‘Jane’,
	‘LN’: 1383859183163:‘Doe’,
	ContactInfo: {
	‘SA’: 1383859185577:‘7 HBase Ave, CA 22222’}

Row Keys

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

For the sake of illustration, Table 12-2 has two simple row keys: 00001 and 00002. Row keys are implemented as byte arrays, and are sorted in byte-lexicographical order, which simply means that the row keys are sorted, byte by byte, from left to right. If you think in terms of numeric values when designing row keys, then sorting is simple. Given two keys, if the byte at Index 1 in Key 1 is less than the byte at Index 1 in Key 2, Row Key 1 will always be stored before Row Key 2, no matter what's next in the sequence of bytes. However, it's common to use printable (ASCII) characters rather than numeric values for row keys in HBase and if you do, you need to understand that the Java language represents characters using the Unicode Standard. The following example illustrates this design consideration for Basic Latin (ASCII).

```
"RowA" precedes "RowA"
"Row-1" precedes "Row11"
"Row1" precedes "RowA"
```

TECHNICAL STUFF If you're not sure of the order for ASCII characters, you can view an ordered table at www.unicode.org/.

You may wonder why you would bother with this fine detail with respect to row keys. The reason for this special attention is that proper row key design is crucial to achieving good performance in HBase — not doing so means you won't realize the full value of your HBase cluster. Our detailed discussion of Row key design at the end of this chapter can help you grasp the importance of the sorting scheme. For now, keep in mind that sorted row keys can help you access your data faster.

Column Families

Table 12-2 shows two column families: CustomerName and ContactInfo. When creating a table in HBase, the developer or administrator is required to define one or more column families using printable characters. (See the earlier section "Row keys" for more on printable characters.) Generally, column families remain fixed throughout the lifetime of an HBase table but new column families can be added by using administrative commands. At the time this book was written, the official recommendation for the number of column families per table was three or less. (We have that number on good authority — see the Apache HBase online documentation at http://hbase.apache.org/book/number.of.cfs.html.) In addition, you should store data with similar access patterns in the same column family — you wouldn't want a customer's middle name stored in a separate column family from the first or last name because you generally access all name data at the same time.

REMEMBER Column families are grouped together on disk, so grouping data with similar access patterns reduces overall disk access and increases performance.

Column Qualifiers

Column qualifiers are specific names assigned to your data values in order to make sure you're able to accurately identify them. Unlike column families, column qualifiers can be virtually unlimited in content, length and number. If you omit the column qualifier, the HBase system will assign one for you. Printable characters are not required, so any type and number of bytes can be used to create a column qualifier. Because the number of column qualifiers is variable, new data can be added to column families on the fly, making HBase flexible and highly scalable. But there's a cost to consider: HBase stores the column qualifier with your value (it's actually part of the key), and since HBase doesn't limit the number of column qualifiers you can have, creating long column qualifiers can be quite costly in terms of storage. That's why we decided to abbreviate the column qualifiers in Table 12-2 (for example, "LN:" was used instead of "LastName"). Notice in our logical representation of the customer contact information in HBase that the system is taking advantage of sparse data support in the case of Jane Doe (again, see Table 12-2). Assuming this table represents customer contact information from a service company, the company isn't too worried about Jane's middle name (abbreviated 'MN') and e-mail addresses (abbreviated 'EA') now, but hopes to (progressively) gather that information over time.

Versions

Looking back at Table 12-2, you can see a number between the column qualifier and value ('FN': 1383859182496:'John,' for example). That number is the *version* number for each value in the table. Values stored in HBase are time stamped by default, which means you have a way to identify different versions of your data right out of the box. It's possible to create a custom versioning scheme, but users typically go with a time stamp created using the current Unix time. (The Unix time or Unix *epoch* represents the number of milliseconds since midnight January 1, 1970 UTC.) The versioned data is stored in decreasing order, so that the most recent value is returned by default unless a query specifies a particular timestamp. You can see in Table 12-2 that our fictional service company at first only had an initial for John Smith's middle name but then later on they learned that the "T" stood for "Timothy." The most recent value for the 'MN' column is stored first in the table.

TIP You can set a limit on the amount of time that data can stay in HBase with a variable called time to live (TTL). You can also set a variable which controls the number of versions per value. This can be done per column family. (You'll be learning more about these variables and how to set them later in the chapter.)

Key Value Pairs

If you're reading this chapter from start to finish, you should be developing a feel for the logical HBase data model. It's simple yet elegant, and it provides a natural data storage mechanism for all kinds of data — especially unstructured big data sets. A little later in this chapter, we cap our discussion of the data model by walking you through a hands-on example to create your first HBase table. First, though, we spend a little time explaining how all these parts of the data model converge into a key-value pair.

First off, in a world where you can think of the row key as the primary key for data stored in HBase, how do you end up leveraging the rest of the data model components? Well, it all depends on how much data you want returned in queries and how long you're willing to wait. Specifying only the row key can potentially return a ton of data, because an individual row can have millions of columns. Also, with only the row key to work from, HBase can return every column qualifier, version, and value related to the row key. What if you want only a particular column or version of your data? From the example shown in Table 12-2, can you see what happens if you want only the last name of a particular

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

customer? The solution is to build a more complex key to specify exactly what you need. A key-value pair can look like this:

```
RowKey:(Column Family:Column Qualifier:Version) => Value
```

After you specify the key, the rest is optional. The more specific you make the query, however (moving from left to right), the more granular the results. Your performance will worsen, because the system has to spend more time locating the exact value or values you need, but less data is returned when the query is finished. So keys are more complex than you might imagine from studying Table 12-2. For example, if you want the most recent middle name (or the only middle name so far) of the customer in row ‘00001’, the resulting key-value pair would look like this:

```
'00001:CustomerName:MN' => 'Timothy'
```

Remember that versions are implemented using time stamps by default and are sorted in decreasing order so that you automatically get the most recent value if you don't specify a version. If you want a prior middle initial for your customer (refer to Table 12-2), your resulting key-value pair would look like this:

```
'00001:CustomerName:MN:1383859182915' => 'T'
```

We hope that our various descriptions of HBase are starting to take shape in your mind. Specifically HBase is both a column family oriented data store and a key-value-pair data store. Referring to HBase as simply a "column oriented" data store leaves a lot to the imagination.

REMEMBER In case you were curious, there are no data types in HBase — values in HBase are just one or more bytes. Again, simple but powerful because you can store anything!

Understanding the HBase Architecture

The reason that folks such as chief financial officers are excited by the thought of using Hadoop is that it lets you store massive amounts of data across a cluster of low cost commodity servers — that's music to the ears of financially minded people. Well, HBase offers the same economic bang for the buck — it's a distributed data store, which leverages a network attached cluster of low-cost commodity servers to store and persist data.

REMEMBER HBase persists data by storing it in HDFS, but alternate storage arrangements are possible. For example, HBase can be deployed in standalone mode in the cloud (typically for educational purposes) or on expensive servers if the use case warrants it.

In most cases, though, HBase implementations look pretty much like the one shown in Figure 12-1.

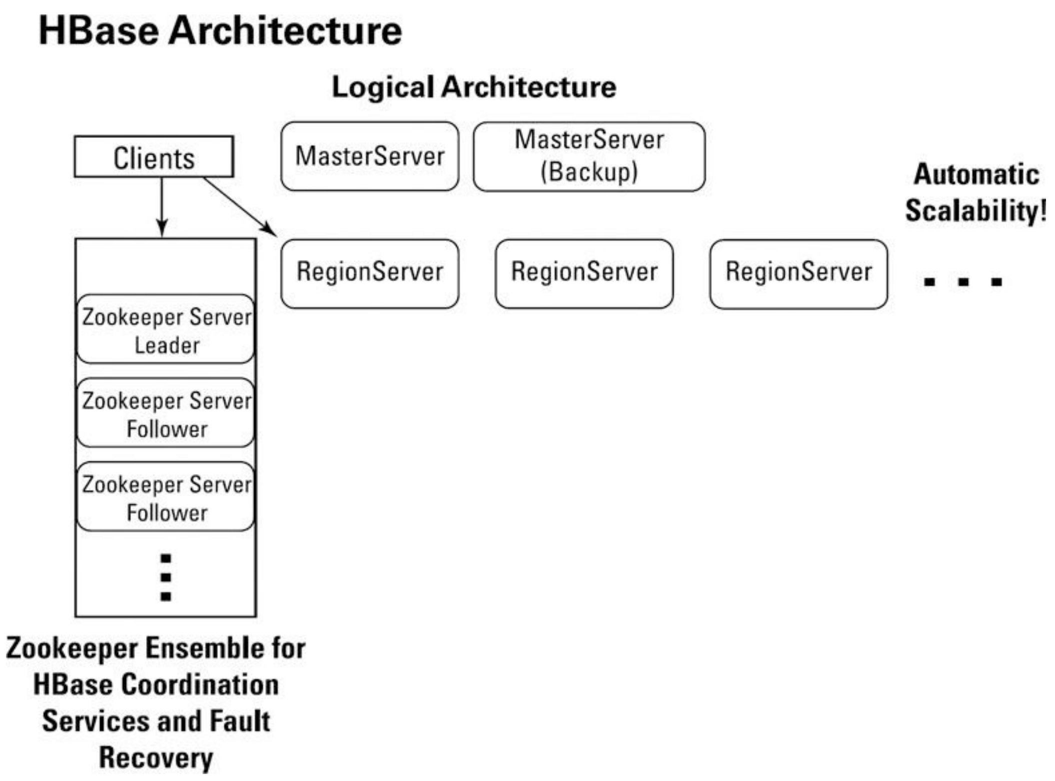


Figure 12-1: The HBase architecture

As with the data model, understanding the components of the architecture is critical for successful HBase cluster deployment. In the next few sections we discuss the key components.

RegionServers

RegionServers are the software processes (often called daemons) you activate to store and retrieve data in HBase. In production

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

environments, each RegionServer is deployed on its own dedicated compute node. When you start using HBase, you create a table and then begin storing and retrieving your data. However, at some point — and perhaps quite quickly in big data use cases — the table grows beyond a configurable limit. At this point, the HBase system automatically splits the table and distributes the load to another RegionServer.

TECHNICAL STUFF In this process, often referred to as *auto-sharding*, HBase automatically scales as you add data to the system — a huge benefit compared to most database management systems, which require manual intervention to scale the overall system beyond a single server. With HBase, as long as you have in the rack another spare server that's configured, scaling is automatic!

Why set a limit on tables and then split them? After all, HDFS is the underlying storage mechanism, so all available disks in the HDFS cluster are available for storing your tables. (Not counting the replication factor, of course; see Chapter 3 for that wrinkle.) If you have an entire cluster at your disposal, why limit yourself to one RegionServer to manage your tables?

Simple. You may have any number of tables large or small and you'll want HBase to leverage all available RegionServers when managing your data. You want to take full advantage of the cluster's compute performance. Furthermore, with many clients accessing your HBase system, you'll want to use many RegionServers to meet the demand. HBase addresses all of these concerns for you and scales automatically in terms of storage capacity and compute power.

Regions

RegionServers are one thing, but you also have to take a look at how individual regions work. In HBase, a table is both spread across a number of RegionServers as well as being made up of individual regions. As tables are split, the splits become regions. Regions store a range of key-value pairs, and each RegionServer manages a configurable number of regions. But what do the individual regions look like? HBase is a column-family-oriented data store, so how do the individual regions store key-value pairs based on the column families they belong to? Figure 12-2 begins to answer these questions and helps you digest more vital information about the architecture of HBase.

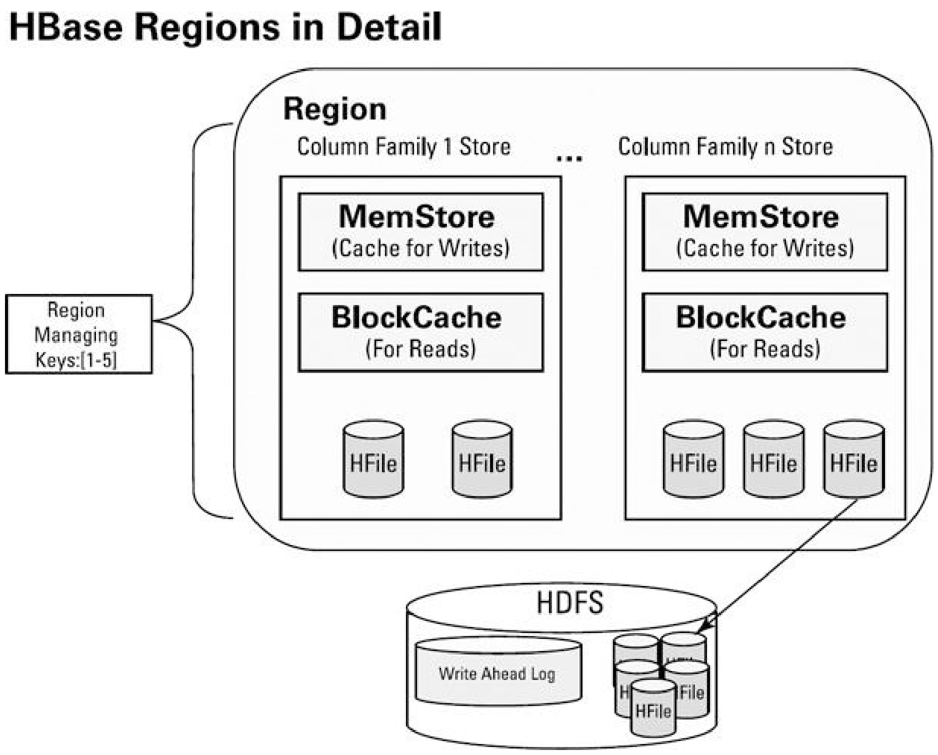


Figure 12-2: HBase regions in detail

REMEMBER HBase is written in Java — like the vast majority of Hadoop technologies. Java is an object oriented programming language and an elegant technology for distributed computing. So, as you continue to find out more about HBase, remember that all of the components in the architecture are ultimately Java objects.

First off, Figure 12-2 gives a pretty good idea of what region objects actually look like, generally speaking. Figure 12-2 also makes it clear that regions separate data into column families and store the data in the HDFS using HFile objects. When clients put key-value pairs into the system, the keys are processed so that data is stored based on the column family the pair belongs to. As shown in the figure, each column family store object has a read cache called the BlockCache and a write cache called the MemStore. The BlockCache helps with random read performance. Data is read in blocks from the HDFS and stored in the BlockCache. Subsequent reads for the data — or data stored in close proximity — will be read from RAM instead of disk, improving overall performance. The Write Ahead Log (WAL, for short) ensures that your HBase writes are reliable. There is one WAL per RegionServer.

REMEMBER Always heed the Iron Law of Distributed Computing: A failure isn't the exception — it's the norm, especially when clustering hundreds or even thousands of servers. Google followed the Iron Law in designing BigTable and HBase followed suit. If you're reading the entire chapter, you'll find out more about how node failures are handled in HBase and how the WAL is a key part of this overall strategy. When

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

you write or modify data in HBase, the data is first persisted to the WAL, which is stored in the HDFS, and then the data is written to the MemStore cache. At configurable intervals, key-value pairs stored in the MemStore are written to HFiles in the HDFS and afterwards WAL entries are erased. If a failure occurs *after* the initial WAL write but *before* the final MemStore write to disk, the WAL can be replayed to avoid any data loss.

Figure 12-2 shows three HFile objects in one column family and two in the other. The design of HBase is to flush column family data stored in the MemStore to one HFile per flush. Then at configurable intervals HFiles are combined into larger HFiles. This strategy queues up the critical compaction operation in HBase, as described in the next section

Compactions Major and Minor

Compaction, the process by which HBase cleans up after itself, comes in two flavors: major and minor. Major compactions can be a big deal so we'll discuss managing them in detail in a bit, but first you need to understand minor compactions.

Minor compactions combine a configurable number of smaller HFiles into one larger HFile. You can tune the number of HFiles to compact and the frequency of a minor compaction. Minor compactions are important because without them, reading a particular row can require many disk reads and cause slow overall performance. Figure 12-3, which illustrates how this concept works, can help you visualize how Table 12-2 can be persisted on the HDFS.

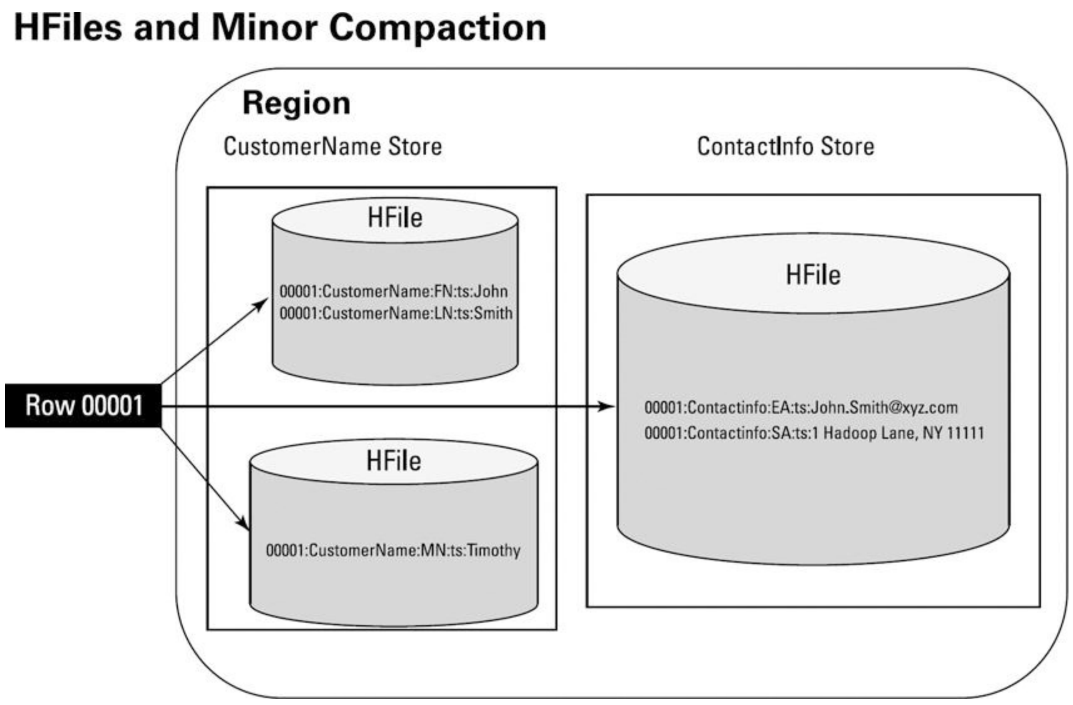

HFiles and Minor Compaction
Region
CustomerName Store
ContactInfo Store
HFile
00001:CustomerName:FN:ts:John
00001:CustomerName:LN:ts:Smith
HFile
00001:CustomerName:MN:ts:Timothy
HFile
00001:Contactinfo:EA:ts:John.Smith@xyz.com
00001:Contactinfo:SA:ts:1 Hadoop Lane, NY 11111
Row 00001

Figure 12-3: HFiles and minor compaction

Looking at Figure 12-3, notice how the CustomerName column family was written to the HDFS with two MemStore flushes and how the data in the ContactInfo column family was persisted to disk with only one MemStore flush. This example is hypothetical, but it's a likely scenario depending on the timing of the writes. Picture a service company that's gaining more and more customer contact information over time. The service company may know its client's first and last name but not learn about its middle name until hours or weeks later in subsequent service requests. This scenario would result in parts of Row 00001 being persisted to the HDFS in different HFiles. Until the HBase system performs a minor compaction, reading from Row 00001 would require three disk reads to retrieve the relevant HFile content! Minor compactions seek to minimize system overhead while keeping the number of HFiles under control. HBase designers took special care to give the HBase administrator as much tuning control as possible to make any system impact "minor."

As its name implies, a major compaction is different from the perspective of a system impact. However, the compaction is quite important to the overall functionality of the HBase system. A major compaction seeks to combine *all* HFiles into one large HFile. In addition, a major compaction does the cleanup work after a user deletes a record. When a user issues a Delete call, the HBase system places a marker in the key-value pair so that it can be permanently removed during the next major compaction. Additionally, because major compactions combine all HFiles into one large HFile, the time is right for the system to review the versions of the data and compare them against the time to live (TTL) property. Values older than the TTL are purged.

REMEMBER *Time to live* refers to the variable in HBase you can set in order to define how long data with multiple versions will remain in HBase. For more information on versions in HBase see the "Understanding the HBase Data Model" section, earlier in this chapter.

TIP For a complete list of HBase tuning parameters see http://hbase.apache.org/book/config.files.html.

You may have guessed that a major compaction significantly affects the system response time. Users who are trying to add, retrieve or manipulate data in the system during a major compaction, they may see poor system response time. In addition, the HBase cluster may have

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

to split regions at the same time that a major compaction is taking place *and* balance the regions across all RegionServers. This scenario would result in a significant amount of network traffic between RegionServers. For these reasons, your HBase administrator needs to have a major compaction strategy for your deployment. We discuss a solution to the major compaction challenge at the end of this chapter, but for now we continue your tour of the basic HBase architecture.

MasterServer

Starting our discussion of architecture by describing RegionServers instead of the MasterServer may surprise you. The term *RegionServer* would seem to imply that it depends on (and is secondary to) the MasterServer and that we should therefore describe the MasterServer first. As the old song goes, though, "it ain't necessarily so." The RegionServers do depend on the MasterServer for certain functions, but not in the sense of a master-slave relationship for data storage and retrieval. In the upper-left corner of Figure 12-1, notice that the clients do not point to the MasterServer, but point instead to the Zookeeper cluster and RegionServers.

The MasterServer isn't in the path for data storage and access — that's the job of the Zookeeper cluster and the RegionServers. We'll cover Zookeeper in the following section and describe client interaction later in this chapter; for now, take a look at the primary functions of the MasterServer, which is also a software process (or daemon) like the RegionServers. The MasterServer is there to

- **Monitor the RegionServers in the HBase cluster**: The MasterServer maintains a list of active RegionServers in the HBase cluster.
- **Handle metadata operations**: When a table is created or its attributes are altered (compression setting, cache settings, versioning, and more) the MasterServer handles the operation and stores the required metadata.
- **Assign regions**: The MasterServer assigns regions to RegionServers.
- **Manage RegionServer failover**: As with any distributed cluster, you hope that node failures don't occur and you plan for them anyway. When region servers fail, Zookeeper notifies the MasterServer so that failover and restore operations can be initiated. We discuss this topic in greater detail in the later section "Zookeeper and HBase reliability."
- **Oversee load balancing of regions across all available RegionServers**: You may recall that tables are comprised of regions which are evenly distributed across all available RegionServers. This is the work of the balancer thread (or *chore*, if you prefer) which the MasterServer periodically activates.
- **Manage (and clean) catalog tables**: Two key catalog tables — labeled `ROOT-` and `.META` — are used by the HBase system to help a client find a particular key value pair in the system.
  - The `-ROOT-` table keeps track of the `.META` table's location in the cluster.
  - The `.META` table keeps track of where each region is located in the cluster.

  The MasterServer provides management of these critical tables on behalf of the overall HBase system.
- **Clear the WAL**: The MasterServer interacts with the WAL during RegionServer failover and periodically cleans the logs.
- **Provide a coprocessor framework for observing master operations**: Here's another new term for your growing HBase glossary. *Coprocessors* run in the context of the MasterServer or RegionServers. For example, a MasterServer observer coprocessor allows you to change or extend the normal functionality of the server when operations such as table creation or table deletion take place. Often coprocessors are used to manage table indexes for advanced HBase applications.

TIP A coprocessor, which runs in the context of the MasterServer and or RegionServer (or both) can be used to enhance security, create secondary indexes, and more. You can find more information about coprocessors at this HBase community blog:

REMEMBER As with all open source Hadoop technologies, MasterServer operations will likely change over time as the community of engineers work on innovations designed to enhance HBase. As of this writing, however, you now have a fairly thorough list that serves as a high-level reference for the MasterServer

Finally, we have one more important point to make about the HBase MasterServer. There can and should be a backup MasterServer in any HBase cluster. (Refer to Figure 12-1.) There needs to be only one active MasterServer at any given time, so the backup MasterServer is for failover purposes. You may recall that the MasterServer isn't in the data access path for HBase clients. However, you may also recall (from the list of functions in this section) that the MasterServer is responsible for actions such as RegionServer failover and load balancing. The good news is that clients can continue to query the HBase cluster if the master goes down but for normal cluster operations, the master should not remain down for any length of time.

Zookeeper and HBase Reliability

Zookeeper is a distributed cluster of servers that collectively provides reliable coordination and synchronization services for clustered applications. Admittedly, the name "Zookeeper" may seem at first to be an odd choice, but when you understand what it does for an HBase cluster, you can see the logic behind it. When you're building and debugging distributed applications "it's a zoo out there," so you should put Zookeeper on your team. (If you're like us, you love it when a technology is appropriately named.)

HBase clusters can be huge and coordinating the operations of the MasterServers, RegionServers, and clients can be a daunting task, but that's where Zookeeper enters the picture. As in HBase, Zookeeper clusters typically run on low-cost commodity x86 servers. Each individual

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

x86 server runs a single Zookeeper software process (hereafter referred to as a Zookeeper server), with one Zookeeper server elected by the ensemble as the leader and the rest of the servers are followers. Zookeeper ensembles are governed by the principle of a majority quorum. Configurations with one Zookeeper server are supported for test and development purposes, but if you want a reliable cluster that can tolerate server failure, you need to deploy at least three Zookeeper servers to achieve a majority quorum.

TIP So, how many Zookeeper servers will you need? Five is the minimum recommended for production use, but you really don't want to go with the bare minimum. When you decide to plan your Zookeeper ensemble, follow this simple formula: 2F + 1 = N where F is the number of failures you can accept in your Zookeeper cluster and N is the total number of Zookeeper servers you must deploy. Five is recommended because one server can be shut down for maintenance but the Zookeeper cluster can still tolerate one server failure.

Zookeeper provides coordination and synchronization with what it calls *znodes*, which are presented as a directory tree and resemble the file path names you'd see in a Unix file system. Znodes *do* store data but not much to speak of — currently less than 1 MB by default. The idea here is that Zookeeper stores znodes in memory and that these memory-based znodes provide fast client access for coordination, status, and other vital functions required by distributed applications like HBase. Zookeeper replicates znodes across the ensemble so if servers fail, the znode data is still available as long as a majority quorum of servers is still up and running.

Another primary Zookeeper concept concerns how znode reads (versus writes) are handled. Any Zookeeper server can handle reads from a client, including the leader, but only the leader issues *atomic* znode writes — writes that either completely succeed or completely fail. When a znode write request arrives at the leader node, the leader broadcasts the write request to the follower nodes and then waits for a majority of followers to acknowledge znode write complete. After the acknowledgement, the leader issues the znode write itself and then reports the successful completion status to the client.

TECHNICAL STUFF Znodes provide some very powerful guarantees. When a Zookeeper client (such as an HBase RegionServer) writes or reads a znode, the operation is *atomic*. It either completely succeeds or completely fails — there are no partial reads or writes. No other competing client can cause the read or write operation to fail. In addition, a znode has an access control lists (ACL) associated with it for security, and it supports versions, timestamps and notification to clients when it changes.

REMEMBER Zookeeper replicates znodes across the ensemble so if servers fail, the znode data is still available as long as a majority quorum of servers is still up and running. This means that writes to any znode from any Zookeeper server must be propagated across the ensemble. The Zookeeper leader manages this operation.

TECHNICAL STUFF This znode write approach can cause followers to fall behind the leader for short periods. Zookeeper solves this potential problem by providing a synchronization command. Clients that cannot tolerate this temporary lack of synchronization within the Zookeeper cluster may decide to issue a sync command before reading znodes.

In a znode world, you're going to come across what looks like the Unix-style pathnames. (Typically they begin with `/hbase`.) These pathnames, which are a subset of the znodes in the Zookeeper system created by HBase, are described in this list:

- `master`: Holds the name of the primary MasterServer,
- `hbaseid`: Holds the cluster's ID,
- `root-region-server`: Points to the RegionServer holding the `-ROOT-` table),
- Something called `/hbase/rs`.

So now you may wonder what's up with this rather vaguely defined `/hbase/rs`. In the previous section, we describe the various operations of the MasterServer and mention that Zookeeper notifies the MasterServer whenever a RegionServer fails. Now we help you take a closer look at how the process actually works in HBase — and you'd be right to assume that it has something to do with `/hbase/rs`. Zookeeper uses its *watches* mechanism to notify clients whenever a znode is created, accessed, or changed in some way. The MasterServers are Zookeeper clients as well as the RegionServers and can leverage these znode watches. When a RegionServer comes online in the HBase system, it connects to the Zookeeper ensemble and creates its own unique ephemeral znode under the znode pathname `/hbase/rs`. At the same time, the Zookeeper system establishes a session with the RegionServer and monitors the session for events. If the RegionServer breaks the session for whatever reason (by failing to send a heartbeat ping, for example), the ephemeral znode that it created is deleted. The action of deleting the RegionServer's child node under `/hbase/rs` will cause the MasterServer to be notified so that it can initiate RegionServer failover. This notification is accomplished by way of a watch that the MasterServer sets up on the `/hbase/rs znode`.

REMEMBER HBase provides a high degree of reliability. When configured with the proper redundancy (a backup MasterServer, proper Zookeeper configuration, and sufficient RegionServers), HBase is sometimes considered *fault tolerant*, meaning that HBase can tolerate any failure and still function properly. This is not exactly true, of course, since (for example) a cascading failure could cause the cluster to fail if the Zookeeper ensemble and or the MasterServers all failed at once. When thinking about HBase and fault tolerance, remember that HBase is a distributed system and that failure modes are quite different in distributed systems versus the traditional high-end scalable database server in a high availability (HA) configuration. To understand HBase fault tolerance and availability in more detail you need to consider the CAP theorem which we introduce in Chapter 11. No discussion of HBase and fault tolerance would be complete without at least mentioning the CAP theorem. CAP stands for "Consistency" (in the data stored), "Availability" (ready for use) and "Partition Tolerance" (tolerant of network failures). Remember, HBase provides "Consistency" and "Partition Tolerance" but is not always "Available." For example, you may have a RegionServer failure and when you do, the availability of your data may be delayed if the failed RegionServer was managing the key (or keys) you were querying at the time of failure. The good news is that the system, if configured properly, will recover (thanks to Zookeeper and the MasterServer) and your data will become available again without manual intervention. So HBase is consistent and tolerant of network failures but not highly available like traditional HA database systems.

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

Taking HBase for a Test Run

In this section, you find out how to download and deploy HBase in standalone mode. We think you'll agree that it's amazingly simple to install HBase and start using the technology. Just keep in mind that HBase is typically deployed on a cluster of commodity servers, though you can also easily deploy HBase in a standalone configuration instead, for learning or demonstration purposes.

TIP For more information on the hardware requirements for HBase, check out the section "Deploying and Tuning HBase," later in this chapter. For Apache's official Quick Start Guide to HBase, check out http://hbase.apache.org/book/quickstart.html.

Like Apache Hadoop, HBase supports Linux primarily but you *can* use Windows in non-production environments if you first download Cygwin. Cygwin gives Microsoft Windows users a Unix shell with all its commands and utilities. So if you follow the Quick Start Guide — which we recommend you do — you'll want to download the latest HBase release (HBase 0.94.7 at the time of this writing).

You get to choose where to install HBase. We decided to install it on a nice little laptop that's currently running a 64-bit Linux kernel. You get to choose where you want to install your HBase. It turns out, though, that if you want things to run in standalone mode, you'll need to edit a couple of files before you can actually start HBase. Look for these files in the `$INSTALL DIR/hbase-0.94.7/conf` directory in the HBase release. The first file is the `hbase-site.xml` file shown in Listing 12-1. The changes you'll want to make are bolded to make them stand out:

Listing 12-1: The *hbase-site.xml* File

```
<configuration>
 <property>
  <name>hbase.rootdir</name></line>
<line>  <value>file:///home/biadmin/my-local-hbase/hbase-data</value></line>
 </property>
 <property>
  <name>hbase.cluster.distributed</name>
  <value>true</value>
 </property>
 <property>
  <name>hbase.zookeeper.property.clientPort</name>
  <value>2222</value>
  <description>Property from ZooKeeper's config zoo.cfg.
   The port at which the clients will connect.
  </description>
 </property>
 <property>
  <name>hbase.zookeeper.property.dataDir</name>
  <value>/home/biadmin/my-local-hbase/zookeeper</value>
 </property>
  <property>
   <name>hbase.zookeeper.quorum</name>
   <value>bivm</value>
  </property>
</configuration>
```

Using the `hbase.rootdir` property, you specify a directory in the local file system to store the HBase data. In production environments, this property would point to the HDFS for the data store. You also set the `hbase.cluster.distributed` property to `true` which causes HBase to start up in a pseudo-distributed mode. If you would choose not to set this property to `true`, HBase would run all of the necessary processes in a single Java Virtual Machine (JVM). However, for the sake of illustration, pseudo-distributed mode will cause HBase to start a RegionServer instance, a MasterServer instance, and a Zookeeper process. Additionally, you need to specify the `hbase.zookeeper.property.clientPort`, the directory where Zookeeper will store its data (`hbase.zookeeper.property.dataDir`) and a list of servers on which Zookeeper will run to form a quorum (`hbase.zookeeper.quorum`). For standalone, you specify only the single Zookeeper server `bivm`.

REMEMBER Getting started with HBase in standalone mode is very straightforward in part because HBase manages Zookeeper for you. You can download a separate Zookeeper release and point HBase to it, but for standalone installs, you'll find it much easier to let HBase manage Zookeeper for you.

To crystallize the decision to let HBase manage Zookeeper for you, we show you how to set an environment variable in yet another HBase file: the `hbase-env.sh` file, to be precise. Listing 12-2 shows what needs to be added:

Listing 12-2: The *hbase-env.sh* File

```
# Tell HBase whether it should manage its own instance of Zookeeper or not.
export HBASE_MANAGES_ZK=true

# The java implementation to use. Java 1.6 required.
```

```
export JAVA_HOME=/opt/ibm/biginsights/jdk
```

In the listing, we've also set the `JAVA_HOME` environment variable to point to the IBM JDK we have on our system. You'll have to make sure you set `JAVA_HOME` to point to your chosen JDK. Finally, you need to specify the name of your Linux system in yet another file — the `regionservers` file. (In a fully distributed production environment, the `regionservers` file would have a line by line list of all servers on which HBase can start the RegionServer process on.)

With the `hbase-site.xml` file and the `hbase-env.sh` file configured, you can now start up HBase and test your install. To start HBase, use the `start-hbase.sh` script as spelled out in Listing 12-3. (We show you how to test the install below.)

Listing 12-3: Starting HBase

```
$ cd $INSTALL_DIR/hbase-0.94.7/bin
$ ./start-hbase.sh
bivm: starting zookeeper, logging to /home/biadmin/my-local-hbase/hbase-0.94.7/
               bin/../logs/hbase-biadmin-zookeeper-bivm.out
starting master, logging to /home/biadmin/my-local-hbase/hbase-0.94.7/bin/..
               /logs/hbase-biadmin-master-bivm.out
localhost: starting regionserver, logging to /home/biadmin/my-local-hbase/hbase-
               0.94.7/bin/../logs/hbase-biadmin-regionserver-bivm.out
```

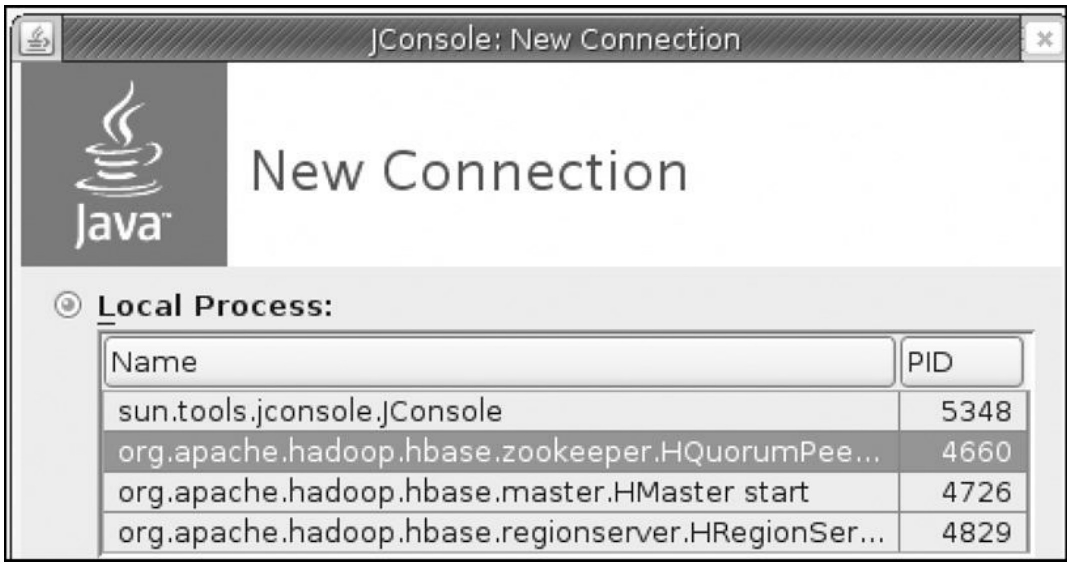
Note that the first line has a `cd` (change directory) command that moves you to an environment variable called `$INSTALL_DIR`. You have to set that variable to your actual install directory for HBase or type out the full path.

REMEMBER In Listing 12-1 we set the `hbase.cluster.distributed` property to `true` which causes HBase to start up in a pseudo-distributed mode. We explained that this would cause HBase to start three processes: a RegionServer instance, a MasterServer instance, and a Zookeeper process. This is exactly what we see in Listing 12-3.

Next we use the JConsole tool, which comes bundled with Java, to perform a quick check on what processes are running after the `start-hbase.sh` script finishes. You can start the JConsole tool by typing the following command:

```
$JAVA_HOME/bin/jconsole
```

In Figure 12-4, JConsole reveals that the three processes that the `start-hbase.sh` script claimed to start are indeed running — the zookeeper, the master and the RegionServer processes.

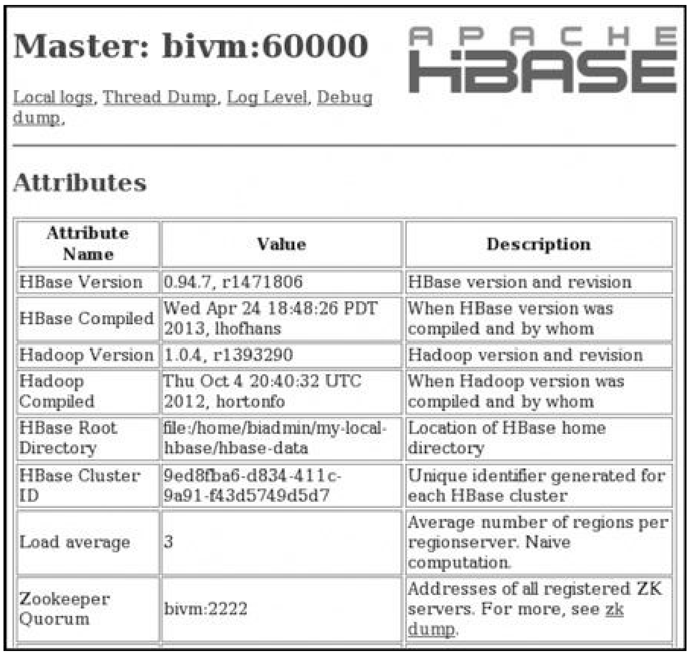


Name	PID
sun.tools.jconsole.JConsole	5348
org.apache.hadoop.hbase.zookeeper.HQuorumPee...	4660
org.apache.hadoop.hbase.master.HMaster start	4726
org.apache.hadoop.hbase.regionserver.HRegionSer...	4829

Figure 12-4: HBase Java processes running In pseudo-distributed mode

To put HBase through its paces, you interact with all three HBase processes, starting with the MasterServer. By default, the MasterServer reports on the system status by way of a browser user interface on port number 60010. In the example, our server name is `bivm` so you can confirm that the MasterServer is running correctly by entering the following URL in a web browser: `http://bivm:60010/`. Doing so brings up the information you see in Figure 12-5.

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited



Master: bivm:60000

Local logs, Thread Dump, Log Level, Debug dump.

Attributes

Attribute Name	Value	Description
HBase Version	0.94.7, r1471806	HBase version and revision
HBase Compiled	Wed Apr 24 18:48:26 PDT 2013, lhofhans	When HBase version was compiled and by whom
Hadoop Version	1.0.4, r1393290	Hadoop version and revision
Hadoop Compiled	Thu Oct 4 20:40:32 UTC 2012, hortonfo	When Hadoop version was compiled and by whom
HBase Root Directory	file:/home/biadmin/my-local-hbase/hbase-data	Location of HBase home directory
HBase Cluster ID	9ed8fba6-d834-411c-9a91-f43d5749d5d7	Unique identifier generated for each HBase cluster
Load average	3	Average number of regions per regionserver. Naive computation.
Zookeeper Quorum	bivm:2222	Addresses of all registered ZK servers. For more, see zk dump.

Figure 12-5: Master-Server user interface screenshot

To keep the figure simple, we've captured only a portion of the MasterServer metrics, but you can see the HBase Root Directory we set in the `hbase-site.xml` file along with the Zookeeper Quorum port number. The RegionServers also report their status and provide critical metrics via a browser user interface on port 60030 by default. We tell you how to interact with the Zookeeper process shortly but first we want to show you how to leverage the RegionServer process and enter some data.

There are a growing number of approaches for clients to access HBase. In the next section entitled "Getting things done with HBase" you'll learn more about the various client options for interacting with HBase. In this section, we introduce you to the HBase shell. You can think of the HBase shell as a client program for interacting with HBase. To activate the HBase shell, first use the `cd` command to change to the `$INSTALL-DIR/bin` directory and then type this command:

```
./hbase shell
```

You should see output like the following example, depending on which version of HBase you've managed to download:

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 0.94.7, r1471806, Wed Apr 24 18:48:26 PDT 2013
```

Creating a Table

And now the real work begins. The HBase shell provides you with a simple set of commands for creating, reading, writing or updating, and deleting tables. Commands to manage and configure tables are also provided. In this section you'll be learning about the `create, put, get, scan` and `describe` commands. (These HBase shell commands are implemented by a Java class called `HTable` that you'll get to try out in the section entitled "Working with an HBase Java API client example".) Start by building the Customer Contact Information table, using the information from Table 12-2.

```
hbase(main):002:0> create 'CustomerContactInfo', 'CustomerName', 'ContactInfo'
0 row(s) in 1.2080 seconds
```

This command creates two column families — ‘CustomerName’ and ‘ContactInfo’ — in a table called ‘CustomerContactInfo’.

Now enter the records from Table 12-2 into the new table, using Listing 12-4 as a model:

Listing 12-4: Entering Records

```
hbase(main):008:0> put 'CustomerContactInfo', '00001', 'CustomerName:FN', 'John'
0 row(s) in 0.2870 seconds

hbase(main):009:0> put 'CustomerContactInfo', '00001', 'CustomerName:LN', 'Smith'
0 row(s) in 0.0170 seconds

hbase(main):010:0> put 'CustomerContactInfo', '00001', 'CustomerName:MN', 'T'
0 row(s) in 0.0070 seconds

hbase(main):011:0> put 'CustomerContactInfo', '00001', 'CustomerName:MN', 'Timothy'
0 row(s) in 0.0050 seconds
```

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

```
hbase(main):012:0> put 'CustomerContactInfo', '00001', 'ContactInfo:EA', 'John.Smith@xyz.com'
0 row(s) in 0.0170 seconds

hbase(main):013:0> put 'CustomerContactInfo', '00001', 'ContactInfo:SA', '1 Hadoop Lane, NY 11111'
0 row(s) in 0.0030 seconds

hbase(main):014:0> put 'CustomerContactInfo', '00002', 'CustomerName:FN', 'Jane'
0 row(s) in 0.0290 seconds

hbase(main):015:0> put 'CustomerContactInfo', '00002', 'CustomerName:LN', 'Doe'
0 row(s) in 0.0090 seconds

hbase(main):016:0> put 'CustomerContactInfo', '00002', 'ContactInfo:SA', '7 HBase Ave, CA 22222'
0 row(s) in 0.0240 seconds
```

After you enter all the data from Table 12-2, you can retrieve the contents of the new table by using the HBase `scan` command. The result should look like Listing 12-5:

Listing 12-5: Scan Results

```
hbase(main):020:0> scan 'CustomerContactInfo', {VERSIONS => 2}
ROW                      COLUMN+CELL
 00001                    column=ContactInfo:EA, timestamp=1383859183030, value=John.Smith@xyz.com
 00001                    column=ContactInfo:SA, timestamp=1383859183073, value=1 Hadoop Lane, NY 11111
 00001                    column=CustomerName:FN, timestamp=1383859182496, value=John
 00001                    column=CustomerName:LN, timestamp=1383859182858, value=Smith
 00001                    column=CustomerName:MN, timestamp=1383859183001, value=Timothy
 00001                    column=CustomerName:MN, timestamp=1383859182915, value=T
 00002                    column=ContactInfo:SA, timestamp=1383859185577, value=7 HBase Ave, CA 22222
 00002                    column=CustomerName:FN, timestamp=1383859183103, value=Jane
 00002                    column=CustomerName:LN, timestamp=1383859183163, value=Doe
2 row(s) in 0.0520 seconds
```

Notice that we specified that HBase should return *two* versions of our values if they exist in the table. This allows us to see the original middle initial of John Smith as well as the latest full middle name.

Now we want to show you how to retrieve individual key-value pairs from our Customer Contact Information table instead of retrieving the whole table with the `scan` command. This will also further illustrate the versioning in HBase. To retrieve data, you'll need to build a key using the shell's `get` command. As you saw earlier in the chapter, keys look like this:

```
RowKey:(Column Family:Column Qualifier:Version)
```

Now, if you just specify the row key (0001 or 0002, for example) then you get *all* the data associated with a row key — the row keys you're using are just not that granular. However, the more specific you get, the less data you get back. Listing 12-6 illustrates this principle in HBase.

Listing 12-6: Using the *get* Command to Retrieve Entire Rows and Individual Values

```
(1) hbase(main):037:0> get 'CustomerContactInfo', '00001'
COLUMN                   CELL
 ContactInfo:EA               timestamp=1383859183030, value=John.Smith@xyz.com
 ContactInfo:SA               timestamp=1383859183073, value=1 Hadoop Lane, NY 11111
 CustomerName:FN              timestamp=1383859182496, value=John
 CustomerName:LN              timestamp=1383859182858, value=Smith
 CustomerName:MN              timestamp=1383859183001, value=Timothy
5 row(s) in 0.0150 seconds

(2) hbase(main):038:0> get 'CustomerContactInfo', '00001',
              {COLUMN => 'CustomerName:MN'}
COLUMN                   CELL
 CustomerName:MN              timestamp=1383859183001, value=Timothy
1 row(s) in 0.0090 seconds

(3) hbase(main):039:0> get 'CustomerContactInfo', '00001',
              {COLUMN => 'CustomerName:MN',
              TIMESTAMP => 1383859182915}
COLUMN                   CELL
 CustomerName:MN              timestamp=1383859182915, value=T
```

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

```
1 row(s) in 0.0290 seconds
```

REMEMBER Note that, in Listing 12-6 above you can see how John Smith's full middle name (Timothy) is returned by default (lines 1 & 2) until we specify an exact timestamp to return the prior middle initial (T in line 3). Note as well that for the last `get` command (line 3), we constructed a full key to retrieve a specific value — in this case the prior middle initial for John Smith. We included the column family name (CustomerContactInfo), column qualifier (MN) and time stamp (1383859182915).

You may be wondering how many versions you can store in the Customer Contact Information table. To answer this question, you'd need to use the `describe` shell command to look at the table descriptors per column family. The first line of Listing 12-7 shows the syntax of the `describe` command and the bolded lines in the same listing give you the answer you're looking for.

Listing 12-7: Using the *describe* Command

```
hbase(main):018:0> describe 'CustomerContactInfo'
DESCRIPTION                                 ENABLED
 'CustomerContactInfo', {NAME => 'ContactInfo', REPLICATION_SCOPE => '0', KE true
 EP_DELETED_CELLS => 'false', COMPRESSION => 'NONE', ENCODE_ON_DISK => 'true
 ',BLOCKCACHE => 'true', MIN_VERSIONS => '0', DATA_BLOCK_ENCODING => 'NONE'
 ,IN_MEMORY => 'false', BLOOMFILTER => 'NONE', TTL => '2147483647', VERSIONS => '3', BLOCKSIZE => '6
 => '0', KEEP_DELETED_CELLS => 'false', COMPRESSION => 'NONE', ENCODE_ON_DI
 SK => 'true', BLOCKCACHE => 'true', MIN_VERSIONS => '0', DATA_BLOCK_ENCODIN
 G => 'NONE', IN_MEMORY => 'false', BLOOMFILTER => 'NONE', TTL => '214748364
 7', VERSIONS => '3', BLOCKSIZE => '65536'}
1 row(s) in 0.0350 seconds

hbase(main):022:0> quit
```

Notice that the default value for `VERSIONS` in both of our column families is 3. This descriptor and others can be modified with the `alter` command by disabling the table (via the `disable` command), altering it, and then enabling the table again with the help of the `enable` command.

Working with Zookeeper

After you've created the table, you should ensure that the Zookeeper process is working smoothly. The way the Zookeeper ensemble works is that it maintains critical data for HBase in data registers it calls *znodes*. If everything has been working correctly, you should now have some meaningful znodes to retrieve. It's time to see whether that assumption is correct.

You've set your Zookeeper port to `2222` in the `hbase-site.xml` file back in Listing 12-1, so using that port number you can bring up a Zookeeper command line interface as shown in Listing 12-8 using the command shown in line 1.

Listing 12-3: Testing Zookeeper

```
(1) ./hbase zkcli -server bivm:2222
Connecting to bivm:2222
13/06/30 12:54:44 INFO zookeeper.ZooKeeper: Client environment:zookeeper.
               version=3.4.5-1392090, built on 09/30/2012 17:52 GMT
13/06/30 12:54:44 INFO zookeeper.ZooKeeper: Client environment:host.name=bivm
...
(2) [zk: bivm:2222(CONNECTED) 0] ls /
[hbase, zookeeper]
(3) [zk: bivm:2222(CONNECTED) 1] ls /hbase
[root-region-server, rs, master, hbaseid, shutdown, backup-masters, unassigned,
             table92, draining, splitlog, online-snapshot, table]
(4) [zk: bivm:2222(CONNECTED) 2] ls /hbase/table
[CustomerContactInfo, .META., -ROOT-]
(5) [zk: bivm:2222(CONNECTED) 5] quit
Quitting...
```

Using the `ls` command (lines 2 & 3), you can browse through the znodes as set up by the MasterServer and RegionServer (line 3). Notice the results of line 4 `ls /hbase/table`. As expected, you can see the Customer Contact Information table that you created using the hbase shell. (We bolded it for you.)

Getting Things Done with HBase

HBase is written in Java, an elegant language for building distributed technologies like HBase, but let's face it — not everyone who wants to take advantage of HBase innovations is a Java developer. That's why there's a rich HBase client ecosystem out there whose sole purpose is

John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

to do the heavy Java lifting for you and let you concentrate on making HBase work for you.

Rich is usually a good characteristic, but when that adjective crosses the line into *overwhelming*, you start having a problem. In case the rich HBase client ecosystem strikes you as overwhelming, we thought we should do some pruning and highlight only the most popular clients available. To make things even easier, we start by giving you an overview of the client ecosystem in diagram form, as shown in Figure 12-6. Note that the diagram is similar to the HBase architecture diagram in Figure 12-1, with an exploded view of the client box.

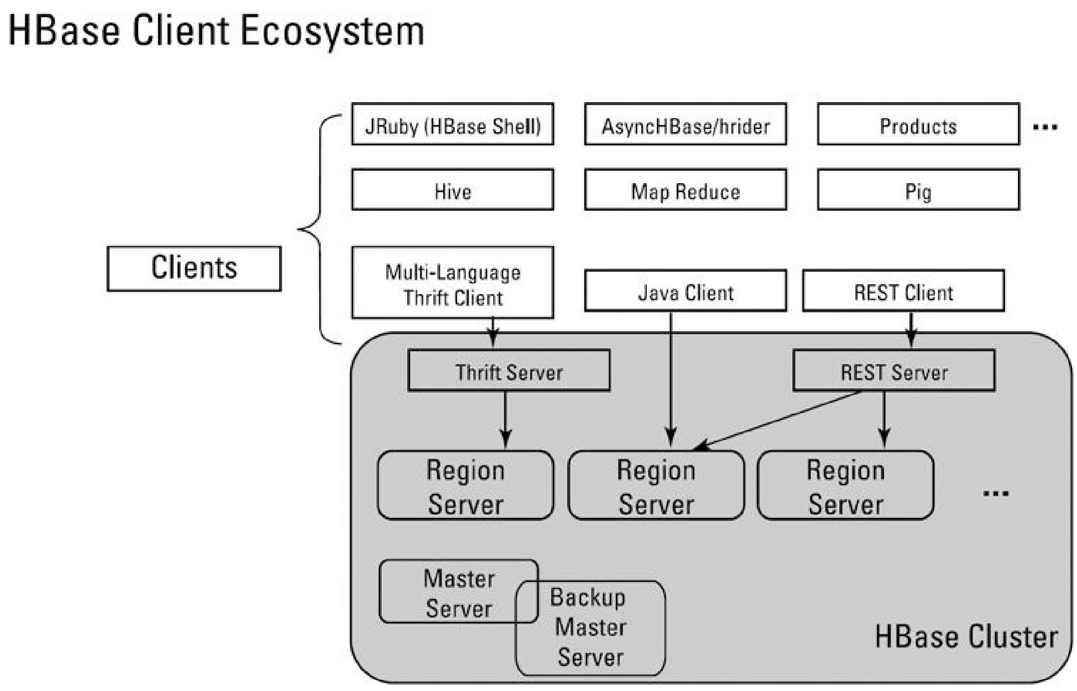


Figure 12-6: The HBase client ecosystem

The following lists summarize your options, starting with the Apache Hadoop clients, more specifically those HBase clients which are part of the Apache Hadoop ecosystem along with those technologies bundled with HBase that are designed to help you build HBase clients:

- **Hive**: Hive is another top level Apache project and it happens to have an entire chapter in this book devoted to it. (Chapter 13, if you're curious.) Hive provides its own take on data warehousing capabilities on top of Apache Hadoop. It comes with a storage handler for HBase, and also provides the HiveQL query language, which is quite similar to SQL. With Hive, you can do all the querying of HBase that you want using HiveQL and — here's the kicker — no Java coding is required when you're using HBase with Hive.

- **MapReduce**: MapReduce is part of the Apache Hadoop framework (and gets some nice coverage in Chapter 6 of this book). MapReduce's claim to fame is that it's a programming model for processing data in parallel on a distributed cluster. In the Hadoop universe, HBase is (as the name implies) the "Hadoop Database." HBase leverages the Hadoop Distributed File System (HDFS) and can also be leveraged by MapReduce jobs. HBase tables can be a source or sink to parallel processing MapReduce jobs. This is an exciting feature included with HBase and has many applications.

- **Pig**: Pig is another technology included with Apache Hadoop and, as with Hive, Pig can leverage HBase. Pig takes you up a level by giving you a higher level programming language called Pig Latin, which can do the heavy MapReduce lifting for you. The details are a bit complicated, but you'll find out more in Chapter 8.

- **Multi-Language Thrift System**: Thrift provides a language-neutral approach to building HBase clients. Developed by Facebook, Thrift's Interface Definition Language (IDL) allows you to define data types and service interfaces so that two different systems written in different languages can communicate with one another. After the IDL is written, Thrift generates the code necessary for communication.

  Here's the really cool part — HBase comes with the Thrift IDL already written for you! As long as Thrift supports your language — and there are 14 supported languages as of this writing — you're well on your way to writing your own custom HBase client. HBase also includes the Thrift Server that's necessary to act as a gateway for your custom client. (That's why the Thrift Server is depicted inside the HBase cluster in Figure 12-6.) It doesn't have to run on a cluster node; it ships with HBase and only needs network access to the cluster. The Thrift server provides a gateway between your client and the HBase Java Client APIs. (More on those in a bit.) You start the Thrift gateway server pretty much like you'd start the HBase shell client — by using the `$INSTALL_DIR/hbase-0.94.7/bin/hbase thrift start` command.

- **Java Client**: If you happen to be a Java developer — hey, we've got no problem with that! — and you understand the ins and outs of Java packages, then you'll want to check out the `org.apache.hadoop.hbase.client` package which comes bundled with the HBase distribution.

  A little later in this chapter we show you a sample Java client that leverages this package, but if you just want to poke around a bit, a good place to start is with the package's `HTable` class. There you'll find the `get`, `put`, `checkAndPut`, `checkAndDelete`, and `delete`

John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

primitives, some of which you tried with the HBase shell in the hands-on example from the "Taking HBase for a test run" section, earlier in this chapter.

These primitives form the data manipulation language of HBase. (Okay, we need to add `scan` here as well; it's also part of the client package but in a separate class.) When you've mastered the package's `HTable` class, you'll want to check out its `HBaseAdmin` class so that you can manage your tables and, while you're at it, take a look at `HTablePool` as well, because it's an efficient way to leverage the Java client APIs.

- **REST System**: Probably the fastest approach for accessing a HBase table is to leverage the REST interface. REST, which stands for *R*epresentational *S*tate *T*ransfer, is the technology that makes your web browser work. Most folks just take web browsers for granted these days, so what could be more natural for anyone than just using your favorite browser as the gateway to an HBase cluster? As with the Thrift approach, the REST gateway server ships with HBase and you need to start at least one in order to enable browser interaction with your tables. To do that, just pick a port number for your gateway server (we'll use 7777) and type the following command:

  ```
  $INSTALL_DIR/hbase-0.94.7/bin/hbase rest start _p 7777
  ```

  If you continue leveraging the example of the Customer Contact Information table from earlier in this chapter, you can type `http://bivm:7777/CustomerContactInfo/schema/` in your browser to have the table schema returned to you — in effect mirroring what the `describe` command would do in the HBase shell (note that 'bivm' is the system name here so you'll need to enter the actual name of your system for this to work).

  TECHNICAL STUFF You can perform HBase client commands like `get`, `put`, `scan`, `delete`, and others using the Unix `curl` command. The `curl` command is often written as *cURL* because it lets you create web browser URLs using the command line. However, you'll need to do a little more work to get human readable results after you start retrieving your data. On its own, the browser returns base64 encoded data, since HBase is just storing bytes.

- **JRuby (HBase Shell)**: The fastest way to roll up your sleeves and learn to use HBase is via the HBase shell. As you've probably already seen in the hands-on example of the HBase shell in the previous section, the shell is a powerful tool for interacting with HBase. The HBase shell is based on JRuby's Interactive Ruby Shell or IRB for short. (For more on JRuby, check out http://jruby.org.) Keep in mind, however, that you can also write scripts and execute them in batch mode. (You see a use case for shell scripts in the "Deploying and Tuning HBase" section, later in this chapter, when we discuss major compactions.)

With the Apache Hadoop clients out of the way, it's time to turn to other HBase clients. The following list describes HBase clients which have been created by other open source communities and commercial companies.

- **AsyncHBase & hrider**: We're seeing lots of open source HBase clients springing up, so we want to introduce you to a pair that are really cool! The first is AsyncHBase which, as the name implies, is an asynchronous client. The standard bundled HBase client found in the `org.apache.hadoop.hbase.client` package is synchronous, which means that when you write a program using the standard package and it accesses an HBase table in some way, your program has to stop and wait for HBase to finish the operation. AsyncHBase provides an alternative to this Stop and Wait approach by letting your program do other things while HBase fulfills your request in the background. The second client is hrider which is a really cool little graphical user interface (GUI) on top of HBase. You know how we used the HBase shell earlier in our hands-on example? Well, hrider lets you interact with HBase through a GUI with a point and a click instead of typing out all of your HBase commands. You can find both of these projects and more on http://github.com.
- **Other Products**: As you would expect, plenty of commercial companies are creating innovative products for HBase — IBM, Cloudera, Hortonworks, and Amazon to name a few. To take just two examples, IBM created Big SQL which allows you to execute SQL against HBase tables, and Cloudera created Impala which improves HiveQL performance when querying data stored in HBase tables. You'll want to check out Chapter 14 for more on Big SQL and Impala.

Working with an HBase Java API Client Example

Here's a simple Java Client example to help you get started if you have your heart set on writing your own client. To run this code on the standalone pseudo-distributed install you've set up, set the Java CLASSPATH environment variable as follows:

```
CLASSPATH=$YOUR_HOME/HBaseClientApp:$INSTALL_DIR/hbase-
          0.94.7/hbase-0.94.7.jar:$INSTALL_DIR/hbase-0.94.7/conf:$INSTALL_DIR/
          hbase-0.94.7/lib/*
```

Your application needs to not only find the HBase `jar` files, but also know where your configuration files reside. Setting the Java `CLASSPATH` environment variable as shown takes care of that task for you. (Without the HBase configuration files, the Java Client APIs cannot find Zookeeper, which is Step # 1 for accessing the installation.)

REMEMBER Before you start working your way through the following sample code, you should know that one of the more powerful features in HBase for making data retrieval more efficient is filters. A *filter* lets you leverage the RegionServer's processing power to separate out the data you need — and the sample Java client example takes advantage of one of these built-in filters. This approach makes your queries faster and reduces the load on your network. Now clients don't have to sort through potentially huge chunks of data to find the record they need!

Listing 12-9 shows a simple Java client example in all its splendor. Note that the code below has been documented with comments — lines starting with `//`, that is — to help you understand how the HBaseClientApp class works. We labeled the comments with bold numbers so you can keep them straight:

Listing 12-9: A Simple Java Client Example

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Get;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.client.ResultScanner;
import org.apache.hadoop.hbase.client.Scan;
import org.apache.hadoop.hbase.filter.*;
import org.apache.hadoop.hbase.util.Bytes;

public class HBaseClientApp {

  // Comment 1
  // HBase programming best practices call for declaring column
  // family names, column qualifiers and other frequently
  // used byte arrays once as constants instead of calling
  // Bytes.toBytes every time you need to create these byte
  // arrays. Bytes.toBytes can be very costly in terms of
  // CPU cycles and can slow down your code especially if you
  // call the method inside a loop.

  private static final byte[] FIRSTROWKEY = Bytes.toBytes("00001");
  private static final byte[] ROWKEY = Bytes.toBytes("91000");
  private static final byte[] CF_CustomerName = Bytes.toBytes("CustomerName");
  private static final byte[] CF_ContactInfo = Bytes.toBytes("ContactInfo");
  private static final byte[] CQ_FirstName = Bytes.toBytes("FN");
  private static final byte[] CQ_LastName = Bytes.toBytes("LN");
  private static final byte[] CQ_EmailAddr = Bytes.toBytes("EA");
  private static final byte[] CQ_StreetAddr = Bytes.toBytes("SA");

  public static void main(String[] args) throws IOException {

   // Comment 2
   // Find the hbase-site.xml configuration
   // file from your CLASSPATH

   Configuration myConfig = HBaseConfiguration.create();

   // Comment 3
   // Create an HTable object and connect it
   // to your Customer Contact Information table

   HTable myTable = new HTable(myConfig, "CustomerContactInfo");

   // Comment 4
   // Create a Put object to enter some new
   // customer information into your Customer Contact Information table.
   // This code assumes that you_re keeping track
   // of your ROWKEY.
   Put myPutObject = new Put(ROWKEY);

   myPutObject.add(CF_CustomerName, CQ_FirstName, Bytes.toBytes("Bruce"));
   myPutObject.add(CF_CustomerName, CQ_LastName,  Bytes.toBytes("Brown"));
   myPutObject.add(CF_ContactInfo, CQ_EmailAddr, Bytes.toBytes("brownb@client.com"));
   myPutObject.add(CF_ContactInfo, CQ_StreetAddr, Bytes.toBytes("HBase Author Lane, CA 33333"));

   // Comment 5
   // Commit our new record to the 'CustomerContactInfo'
   // table.
   myTable.put(myPutObject);

   // Comment 6
   // In the example below you are leveraging one of the many
   // built-in filters to query the Customer Contact
   // Information table for customers that have a
   // particular email address. Only client records that have
```

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

```
   // a particular domain name in their email address
   // are returned to our Java Client.

   Filter companyFilter = new ValueFilter(CompareFilter.CompareOp.EQUAL,
        new SubstringComparator("@client.com"));
   Scan myScanner = new Scan(FIRSTROWKEY,companyFilter);
   ResultScanner myResults = myTable.getScanner(myScanner);
   for (Result res : myResults) {
     System.out.println(Bytes.toString(res.value()));
   }
  }
}
```

Beginning with Comment 1, this block of Java code illustrates a best practice with HBase, namely that you want to define your byte arrays holding your HBase row keys, column family names, column qualifiers and data up front as constants. This saves valuable CPU cycles and makes your code run faster, especially if there are repeated loop constructs. If you were wondering how the main method that executes the HBase commands finds the Zookeeper ensemble and then RegionServers, Comment 2 explains this. However, the `HBaseConfiguration.create` method won't find your cluster if your `CLASSPATH` environment variable is not set correctly, so don't forget that task! Comment 3 explains how our `HBaseClientApp` class connects with the 'CustomerContactInfo' table and Comments 4 and 5 show you how you can place data in our 'CustomerContactInfo' table. Finally, Comment 6 explains how HBase filter technology can improve your table scans by allowing you to target specific data in the table. Without filters you would be pulling much more data out of the HBase cluster and across the network to your client where you would have to write code to sort through the results. Filters make HBase life a whole lot easier!

If you compile and run this example application, you'll have added a new customer name (Bruce Brown) and the customer's contact info (brownb@client.com, residing at HBase Author Lane, CA 33333) to the Customer Contact Information table and you'll have used a filter to track down and print the e-mail address.

HBase and the RDBMS world

We think it's best to state right up front that HBase and relational database technology (like Oracle, DB2, and MySQL to name just a few) really don't compare all that well. Despite the cliché, it's truly a case of comparing apples to oranges. HBase is a *NoSQL* technology — we explain the meaning of this catchy nomenclature in detail in Chapter 11 and we discuss the major differences between relational database management systems (RDBMSs) and HBase in Chapter 11 as well. If your background is in relational database technology and you are wondering how you might convert some of your databases to HBase — or even if that makes sense — then this section is just for you! We'll start with a brief description (or refresher if you read Chapter 11) of the differences and then we'll discuss some considerations and guidelines for making the move.

BigTable, HBase's Google forebear, was born out of a need to manage massive amounts of data in a seamless, scalable fashion. HBase is a direct implementation of BigTable providing the same scalability properties, reliability, fault recovery, a rich client ecosystem, and a simple yet powerful programming model. The relational data model and the database systems that followed were built with different goals in mind. The relational model and accompanying structured query language (SQL) is a mathematical approach that enforces data integrity, defines how data is to be manipulated, provides a basis for efficient data storage and prevents update anomalies by way of the normalization process. Though HBase and the RDBMS have some commonalities, the design goals were different.

You may wonder why the examples earlier in this chapter center on mapping a relational table — our Customer Contact Information table — to an HBase table. The reason is two-fold:

- The relational model is the most prevalent, so using *that* model for the sake of comparisons often helps professionals coming from the world of RDBMSs better grasp the HBase data model.
- The innovations provided by BigTable and HBase are making this new NoSQL technology an attractive alternative for certain applications that don't necessarily fit the RDBMS model. (The ability of HBase to scale automatically is alone a huge innovation for the world of database technology!)

Knowing When HBase Makes Sense for You?

So, when should you consider using HBase? Though the answer to this question isn't necessarily straightforward for everyone, for starters you clearly must have

- **A big data requirement**: We're talking terabytes to petabytes here — otherwise you'll have a lot of idle servers in your racks.
- **Sufficient hardware resources**: Five servers is a good starting point, as described in the "Hardware Architecture" row in Table 12-4.

When considering which route to take — HBase versus RDBMS — consider other requirements such as transaction support, rich data types, indexes, and query language support — though these factors are not as black and white as the preceding two bullets. Rich data types, indexes and query language support can be added via other technologies, such as Hive or commercial products, as described in Chapter 13. "What about transactions?" you ask. The answer to that question is in the following section.

ACID Properties in HBase

John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

Certain use cases for RDBMSs, like online transaction processing, depend on ACID-compliant transactions between the client and the RDBMS for the system to function properly. (We define the ACID acronym — **A**tomicity, **C**onsistency, **I**solation, and **D**urability — in Chapter 11.) When compared to an RDBMS, HBase isn't considered an ACID-compliant database as of this writing. HBase does not support ACID-compliant transactions over multiple rows or across tables. However, HBase does guarantee the following aspects:

- **Atomic**: All row level operations within a table are atomic. This guarantee is maintained even when there's more than one column family within a row. HBase provides, in addition to the `get`, `put`, `delete` and `scan` commands described earlier in this chapter, atomic `increment`, `checkAndPut` and `checkAndDelete` methods.
- **Consistency**: Scan operations return a consistent view of the data stored in HBase at some point in the past. Concurrent client interaction could update a row during a multi-row scan, but all rows returned by a scan operation will always contain valid data from some point in the past.
- **Durability**: Any data that can be retrieved from HBase has also been made *durable to disk* (persisted to HDFS, in other words).

TECHNICAL STUFF One of the exciting aspects of HBase and other open source Apache projects is that someone in the community is always innovating and trying to improve the technology. HBase does support multi-row transactions if the rows are on the same RegionServer. This feature, which requires additional coding, was introduced in HBase version 0.94.0 documented at (If you're curious, the additional coding focused on HBase's split policy.)

REMEMBER When ACID properties are required by HBase clients, design the HBase schema such that cross row or cross table data operations are not required. Keeping data within a row provides atomicity.

Transitioning from an RDBMS Model to HBase

If you're facing the design phase for your application and you believe that HBase would be a good fit, then designing your row keys and schema to fit the HBase data model and architecture is the right approach. However, sometimes it makes sense to move a database originally designed for an RDBMS to HBase. A common scenario where this approach makes sense is a MySQL database instance that has reached its limits of scalability. Techniques exist for horizontally scaling a MySQL instance (*sharding*, in other words) but this process is usually cumbersome and problematic because MySQL simply was not originally designed for sharding. If you're in this predicament yet you believe that the HBase differences are manageable, then read on. The tips in this section may save you some valuable time.

Transitioning from the relational model to the HBase model is a relatively new discipline. However, certain established patterns of thought are emerging and have coalesced into three key principles to follow when approaching a transition. These principles are *denormalization, duplication*, and *intelligent keys (DDI)*. The following list takes a closer look at each principle:

- **Denormalization**: The relational database model depends on a) a normalized database schema and b) joins between tables to respond to SQL operations. Database normalization is a technique which guards against data loss, redundancy, and other anomalies as data is updated and retrieved. There are a number of rules the experts follow to arrive at a normalized database schema (and database normalization is a whole study itself), but the process usually involves dividing larger tables into smaller tables and defining relationships between them. Database denormalization is the opposite of normalization, where smaller, more specific tables are joined into larger, more general tables. This is a common pattern when transitioning to HBase because joins are not provided across tables, and joins can be slow since they involve costly disk operations. Guarding against the update and retrieval anomalies is now the job of your HBase client application, since the protections afforded you by normalization are null and void.
- **Duplication**: As you denormalize your database schema, you will likely end up duplicating the data because it can help you avoid costly read operations across multiple tables. Don't be concerned about the extra storage (within reason of course); you can use the automatic scalability of HBase to your advantage. Be aware, though, that extra work will be required by your client application to duplicate the data and remember that natively HBase only provides row level atomic operations not cross row (with the exception described in the HBASE-5229 JIRA) or cross table.
- **Intelligent Keys**: Because the data stored in HBase is ordered by row key, and the row key is the only native index provided by the system, careful intelligent design of the row key can make a huge difference. For example, your row key could be a combination of a service order number and the customer's ID number that placed the service order. This row key design would allow you to look up data related to the service order or look up data related to the customer using the same row key in the same table. This technique will be faster for some queries and avoid costly table joins.

To clarify these particular patterns of thought, we expand on the example of the Customer Contact Information table by placing it within the context of a typical service order database. Figure 12-7 shows you what a normalized service order database schema might look like.

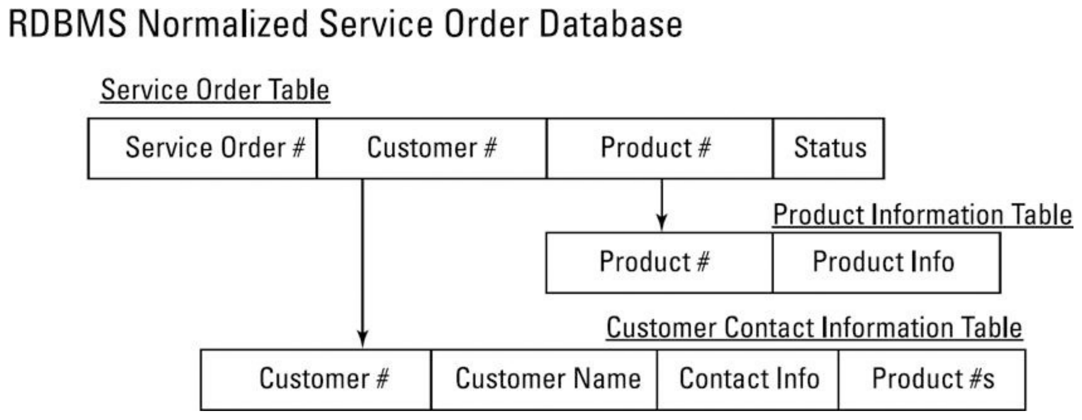


Figure 12-7: An RDBMS normalized service order database

Following the rules of RDBMS normalization, we set up the sample Customer Contact Information table so that it is separate from the service order table in order to avoid losing customer data when service orders are closed and possibly deleted. We took the same approach for the Products table, which means that new products can be added to the fictional company database independently of service orders. By relying on RDBMS join operations, this schema supports queries that reveal the number of service orders that are opened against a particular product along with the customer's location where the product is in use.

That's all fine and dandy, but it's a schema you'd use with RDBM. How do you transition this schema to an HBase schema? Figure 12-8 illustrates a possible HBase scheme — one that follows the DDI design pattern.

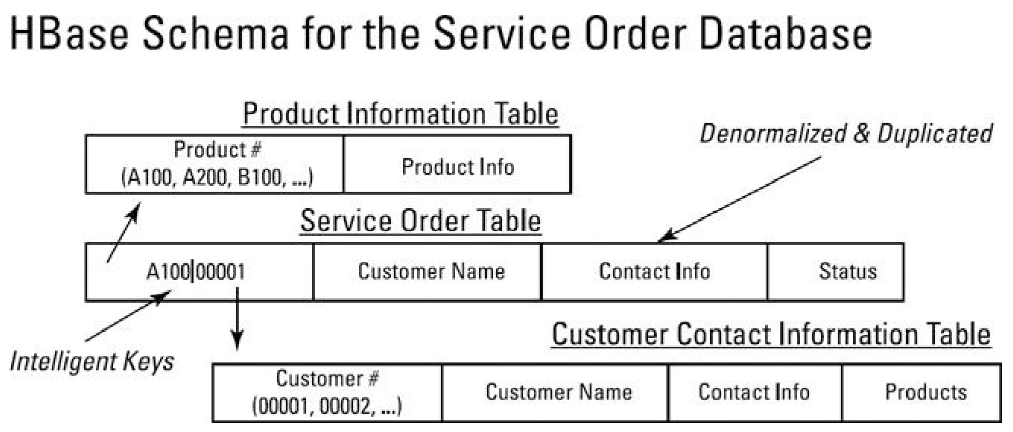

HBase Schema for the Service Order Database
Product Information Table
Product # (A100, A200, B100, ...)
Product Info
Denormalized & Duplicated
Service Order Table
A100|00001
Customer Name
Contact Info
Status
Intelligent Keys
Customer Contact Information Table
Customer # (00001, 00002, ...)
Customer Name
Contact Info
Products

Figure 12-8: An HBase schema for the service order database

To avoid costly search operations and additional HBase `get` and/or `scan` operations (or both), the Customer Contact Information table has been denormalized by including the customer name and contact info in place of the foreign keys used previously. (See Figure 12-7.) Also, the data is duplicated by keeping the Customer Contact Information table as is. Now joins across the Service Order table and Customer Contact Information table are not necessary. Additionally, an intelligent row key design has been employed that combines the product number with the customer number to form the service order number (A100|00001, for example). Using this intelligent key, the service order table can provide vital reports about product deficiencies and customers who are currently experiencing product issues. All these queries can all be supported by HBase in a row level atomic fashion for the application. We don't have to worry about the lack of ACID compliant joins across the two tables. Additionally, because you know that HBase orders row keys and sorts them in a lexicographical fashion, your application can make certain educated guesses about data locality when issuing scans for reporting. (All A* series product numbers will be stored together, for example.)

The service order database represented by the HBase schema (refer to Figure 12-8) is a relatively simple example, but it illustrates how HBase can, in certain cases, intersect with the RDBMS world and provide significant value. If the fictional company has terabytes or even petabytes of service call data to store, HBase would make a huge difference in terms of cost, reliability, performance, and scale. You can, of course, design your service order HBase schema in several different ways. Admittedly, the design all depends on the queries that must be supported, but you have the ability to transition some relational databases to very powerful HBase applications for production use as long as you work from a solid understanding of HBase architecture and the DDI design pattern.

REMEMBER This example has assumed that queries were performed by a Java application leveraging the HBase client APIs, or perhaps via another language using Apache Thrift. This application model may fit the requirements just fine and provide useful performance and customization options for the fictional service company. However, the downside is that the service order database schema is tied pretty tightly to the application layer that issues the queries and manages the database integrity. (We say "tied pretty tightly" because changes to the HBase schema would require changes to the application code.) You might ask these two questions: "could either HiveQL or other commercial offerings providing SQL support for HBase be used to make this process easier for the engineers creating this HBase application?" (changing HiveQL or SQL is certainly easier and less costly than changing application code) and "could joins be performed when appropriate using Hadoop MapReduce?" (That might be easier than following the DDI pattern if the limited ACID properties provided by

John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

HBase fit your application.) The answer to these questions is "yes," so you'll want to check out Chapters 13 and 15 to see how HBase can be combined with other Hadoop and commercial products to create some very powerful big data applications.

Deploying and Tuning HBase

HBase is a powerful and flexible technology, but accompanying this flexibility is the requirement for proper configuration and tuning. Now, installing and running HBase in a standalone mode for learning purposes is seamless and very straightforward, but standalone mode for testing purposes in no way, shape, or form represents the real world. Admittedly, some applications of HBase simply involve ingesting large amounts of data and then querying that data occasionally over time at a leisurely pace — no strict deadlines, in other words, which means that you don't have to worry too much about efficiencies. Other possibilities might include running a Map Reduce job against the data for business analytics or machine learning applications. These use cases are batch oriented in nature and if your use case fits into this category, you probably don't need to configure and tune HBase all that much. Your main concerns are the proper hardware configuration and correct configuration files. Reviewing the Apache HBase online Quick Start guide (http://hbase.apache.org/book/quickstart.html) and following the guidance in the later section "Hardware requirements" is likely all you need to soon be on your way.

Most HBase deployments, however, are going to include performance expectations or performance requirements (or both) along with the expectation that you take advantage of freebies such as auto sharding and automatic recovery after node failures. Often new use cases arise after an organization becomes accustomed to the new and shiny database toy on the network and so the original expectations or requirements can change. So for all these reasons and more, deploying HBase at scale in production environments typically requires careful thought and an understanding of how to tune HBase. The good news is that, in our experience, a little tuning goes a long way, so don't feel overwhelmed. We've personally seen HBase performance improve by several orders of magnitude by simply following the suggestions in this section and in the online page "Apache HBase performance tuning" (http://hbase.apache.org/book/performance.html).

Hardware Requirements

It's time for some general guidelines for configuring HBase clusters. Your "mileage" may vary, depending on specific compute requirements for your RegionServers (custom coprocessors, for example) and other applications you may choose to co-locate on your cluster.

RegionServers

The first temptation to resist when configuring your RegionServers is plunking down lots of cash for some high end enterprise systems. Don't do it! HBase is typically deployed on plain vanilla commodity x86 servers. Now, don't take that statement as license to deploy the cheapest, low quality servers. Yes, HBase is designed to recover from node failures but your availability suffers during recovery periods so hardware quality and redundancy *do* matter. Redundant power supplies as well as redundant network interface cards are a good idea for production deployments. Typically, organizations choose two socket machines with four to six cores each.

The second temptation to resist is configuring your server with the maximum storage and memory capacity. A common configuration would include from 6 to 12 terabytes (TB) of disk space and from 48 to 96 gigabytes (GB) of RAM. RAID controllers for the disks are unnecessary because HDFS provides data protection when disks fail.

TECHNICAL STUFF HBase requires a read and write cache that's allocated from the Java heap. Keep this statement in mind as you read about the HBase configuration variables because you'll see that a direct relationship exists between a RegionServer's disk capacity and a RegionServer's Java heap. You can find an excellent discussion on HBase RegionServer memory sizing at

```
http://hadoop-hbase.blogspot.com/2013/01/hbase-region-
          server-memory-sizing.html
```

The article points out that you can estimate the ratio of raw disk space to Java heap by following this formula:

RegionSize *divided by* Memstoresize *multiplied by* HDFS Replication Factor *multiplied by* HeapFractionForMemstores

Using the default HBase configuration variables from http://hbase.apache.org/book/config.files.html provides this ratio:

10GB / 128MB * 3 * 0.4 = Ratio of 96MB disk space: 1 MB Java heap space.

The preceding line equates to 3TB of raw disk capacity per RegionServer with 32GB of RAM allocated to the Java heap.

What you end up with, then, is 1 terabyte of usable space per RegionServer since the default HDFS replication factor is 3. This number is still impressive in terms of database storage per node but not so impressive given that commodity servers can typically accommodate eight or more drives with a capacity of 2 to 4 terabyte a piece. The overarching problem as of this writing is the fact that current Java Virtual Machines (JVMs) struggle to provide efficient memory management (garbage collection, to be precise) with large heap spaces (spaces greater than 32GB, for example).

Yes, there are garbage collection tuning parameters you can use, and you should check with your JVM vendor to insure you have the latest options, but you won't be able to get very far using them at this time. The memory management issue will eventually be solved but for now be aware that you may encounter a problem if your HBase storage requirements are in the range of hundreds of terabytes to more than a petabyte. You can easily increase the `hbase.hregion.max.filesize` to 20GB to reach 6TB raw and 2TB usable. You can make other tweaks (reducing MemStore size for read heavy workloads, for example) but you won't make orders of magnitude leaps in the useable space until we have a JVM that efficiently handles garbage collection with massive heaps.

TECHNICAL STUFF You can find ways around the JVM garbage collection issue for RegionServers but the solutions are new and are not

John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

yet part of the main HBase distribution as of this writing. If your HBase data store requirements are massive, check out the "bucket cache" article at before you buy too many RegionServers.

Master Servers

The MasterServer does not consume system resources like the RegionServers do. However, you should provide for hardware redundancy, including RAID to prevent system failure. For good measure, also configure a backup MasterServer into the cluster. A common configuration is 4 CPU cores, between 8GB and 16GB of RAM and 1 Gigabit Ethernet is a common configuration. If you co-locate MasterServers and Zookeeper nodes, 16GB of RAM is advisable.

Zookeeper

Like the MasterServer, Zookeeper doesn't require a large hardware configuration, but Zookeeper must not block (or be required to compete for) system resources. Zookeeper, which is the coordination service for an HBase cluster, sits in the data path for clients. If Zookeeper cannot do its job, time-outs will occur — and the results can be catastrophic. Zookeeper hardware requirements are the same as for the MasterServer except that a dedicated disk should be provided for the process. For small clusters you can co-locate Zookeeper with the master server but remember that Zookeeper needs sufficient system resources to run when ready.

Deployment Considerations

Now that you have a solid understanding of HBase hardware (HW) requirements, we have a couple of points for you regarding deployment:

- In this chapter, we're assuming that you're concerned primarily about setting up an HBase cluster. Though co-locating HBase with MapReduce is often done, it affects performance and sizing requirements. So if you're serious about maximum HBase performance, consider carefully the additional HW resources you may require or provision a separate cluster for MapReduce and other Hadoop jobs. Then you can keep your HBase cluster separate.
- Deploying Hadoop is the subject of Chapter 16 so we encourage you to check that chapter out. In Chapter 16, you'll find more detail on networking as well as physical HW deployment examples for HBase and Hadoop. We cover Hadoop 1 deployments as well as Hadoop 2 deployments.

Tuning Prerequisites

Any serious HBase installation requires some standard setup on your cluster and on your individual nodes. We give you a few examples here and then point you to the sections in the Apache HBase online documentation you'll need to reference. First take a look at monitoring and management.

Tools to Monitor Your Cluster

If you've had the privilege of engineering a system at some point in your career, you know you face the major challenge of coming up with a rigorous testing procedure to ensure that your system is ready for its production phase. If you don't plan for testing and debugging right up front, you'll likely miss your production deadlines or fail altogether. The HBase and Hadoop committers made sure that you would have a rich metrics subsystem to draw on during the debug and test phase. You can find all the messy details in the Apache HBase online documentation (http://hbase.apache.org/book.html#ops_mgt), especially the sections dealing with HBase Backup and Replication. In this section, we give you an overview of the available tools.

TIP The Cluster Replication feature is a key tool when debugging, tuning or if you want to run Map Reduce against your tables without impacting performance. Obviously, you'll need it for disaster recover as well.

Getting started with the Hadoop management tools set is surprisingly easy. HBase leverages the Java Management Extensions (JMX) technology for exposing key metrics. And with the Java Virtual Machine, you also get the JConsole tool, a free JMX client that you can use to view HBase metrics. The HBase distribution we've been working with (0.94.7) enables access via JConsole by default, so in your standalone environment you simply select the HBase server that you want to monitor and JConsole then presents you with a graphical user interface for viewing key server metrics.

REMEMBER You can start the JConsole tool with the following command:

```
$JAVA_HOME/bin/jconsole
```

Additionally, you should familiarize yourself with these two other open source technologies for monitoring your HBase cluster:

- **Ganglia**: Often used to provide monitoring graphs over time, Ganglia can help you spot problems that occur occasionally or only after days of operation.
- **Nagios**: Nagios is useful if you're an HBase administrator and you want to receive a page on your pager or an e-mail if, say, a RegionServer goes down or you have a garbage collection issue in your cluster.

REMEMBER If you're leveraging HBase as part of a commercial product, be sure to check with your vendor for a tool to monitor and manage HBase.

Cluster Setup

HBase typically deploys on a cluster, and you'll need to make some adjustments on each of your servers to accommodate HBase

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

components. A good first step is insuring that the system clocks on each server in your cluster are in sync. Out of sync system clocks on your servers can really confuse HBase, so check out the Network Time Protocol or NTP for short. Running the NTP on your cluster will take care of any time synchronization issues. Furthermore, HBase is a unique application in certain respects because it stresses your system beyond the level that applications may do. The truth is that HBase is going to be opening a lot of files — that's just the nature of the beast. Given that fact, you need to ensure that your operating systems are configured to handle what is sure to be a far-from-typical file system load. Swapping in your Linux operating systems (moving between disk and memory, in other words) can have very adverse effects on Zookeeper. Finally there's the Java Virtual Machine (JVM) that ultimately runs on each of your nodes and executes the HBase processes. HBase also puts far-from-typical stress on the JVM. (For example, the MemStore cache, which heavily exercises the garbage collection system, is sure to be taxed to the max.)

REMEMBER When the MemStore is committed to HFiles on the HDFS, the Java heap is reclaimed. This can result in long garbage collection pauses if your JVM is not configured correctly.

So for all of these reasons and more you should review these two sections of the Apache HBase online documentation:

- **General Configuration Requirements**: Review Chapter 2 of the Apache HBase online documentation (http://hbase.apache.org/book/configuration.html) and especially section 2.5 entitled "The Important Configurations" - http://hbase.apache.org/book/important_configurations. html.
- **Java Virtual Machine**: Determine which JVM you're running and make sure that it has been tested for compatibility with HBase. As of this writing, the Apache HBase online documentation suggests Java 6 from Oracle because Java 7 hasn't been fully tested. Another JVM we've tested is IBM's J9. If you plan to use J9, review the IBM documentation for the latest command line options when starting your JVMs. If you plan to use Oracle's JVM, review the following sections of the HBase online documentation to familiarize yourself with the proper settings: http://hbase.apache.org/book/jvm.html and http://hbase.apache.org/book/trouble.log.html#trouble. log.gc

Enabling Compression

Compression boosts HBase performance by reducing overall disk input/output. Consider enabling compression unless your data doesn't compress well (images, for example) or if your RegionServers cannot handle the additional CPU load that compression and decompression requires. Compression can be enabled via the HBase shell command, as we explain in the "Taking HBase for a test run" section, earlier in this chapter, when we tell you how to leverage the `describe` shell command (see Listing 12-10) to view our Customer Contact Information table descriptors:

Listing 12-10: The *describe* Shell Command

```
hbase(main):018:0> describe 'CustomerContactInfo'
... {NAME => 'ContactInfo', REPLICATION_SCOPE => '0', KE true
 EP_DELETED_CELLS = 'false', COMPRESSION = 'NONE',...
```

By default, compression is disabled per column family. The supported compression types are Gzip, LZO and Snappy (with some other derivatives available and more on the way). GZIP is best overall for achieving a good compression ratio, but LZO and Snappy are faster. Keep in mind, though, that both LZO and Snappy compression codecs must be installed separately; only Gzip works without further configuration steps. Listing 12-11 shows the steps you'd need to enable Gzip compression on the Customer Contact Information table:

Listing 12-11: Enabling Gzip Compression

```
hbase(main):007:0> disable 'CustomerContactInfo'
hbase(main):010:0> alter 'CustomerContactInfo', { NAME => 'CustomerName',
                COMPRESSION => 'GZ' }
hbase(main):014:0> describe 'CustomerContactInfo'
...{NAME => 'CustomerName', REPLICATION_SC
 OPE => '0', KEEP_DELETED_CELLS => 'false', COMPRESSION => 'GZ',...
hbase(main):017:0> enable 'CustomerContactInfo'
```

Understanding Your Data Access Patterns

Achieving peak performance with HBase requires an understanding of your data access patterns. How will your application or clients query HBase? Is your data ingested in bulk or gradually over time? Are the patterns mostly reads or writes or a mix of both? Are the queries random or sequential? How much data is read or written per query? Often no clear answer to these questions emerges or the answer varies per table. However, the good news is that you can tune on a per table basis, so choosing a few key tables to tune can help a great deal.

REMEMBER It is beyond the scope of this section and somewhat unrealistic to cover all tuning scenarios, but we do want to provide some general guidance which will (hopefully) help you focus on the most important issues and tuning parameters:

- As mentioned earlier in this chapter, the Apache HBase online guide has a whole section on performance tuning that's worth your while to check out:

  http://hbase.apache.org/book.html#performance

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

- While you're at it, review the online guide's Apache HBase configuration coverage, especially the section about HBase configuration variables:

  http://hbase.apache.org/book/config.files.html

- Commonly used variables are in the section about performance tuning:

  http://hbase.apache.org/book/perf.configurations.html

Here's what we recommend for some common situations:

- **Read Heavy Workloads**: If the read workload is random, consider increasing the `hfile.block.cache.size` setting and shrinking the `hbase.regionserver.global.memstore.upperLimit` and `hbase.regionserver.global.memstore.lowerLimit` settings.

  You can also keep part or all of a column family in memory by setting the `in_memory` descriptor to `true` while disabling the block cache altogether for other tables' column families.

  ```
  { NAME => 'columnfamily', IN_MEMORY => 'true' }
  { NAME => 'columnfamily', BLOCKCACHE => 'false' }
  ```

  If the read workload is sequential, caching will most likely not help your performance so look at increasing the HFile block size to achieve more data per read. The HBase API docs suggest numbers between 8KB and 1MB, with the default setting of 64KB. (We suggest going with 128KB in the example below.) Also review your `hbase.client.scanner.caching` setting to ensure that it fits your sequential read patterns.

  ```
  { NAME => 'columnfamily', BLOCKSIZE => 131072 }
  ```

- **Write Heavy Workloads**: With write heavy workloads, the MemStore configuration becomes quite important, so review all settings that affect the MemStore write cache — things like the `hbase.regionserver.global.memstore.lowerLimit` setting and the `hbase.regionserver.global.memstore.upperLimit` setting.

  JVM garbage collection must also be configured correctly so that large garbage collection pauses don't occur — and slow your application. Worse yet, such large pauses can confuse Zookeeper into believing that your RegionServer has failed and then your HBase experience gets *ugly*. Finally you should have a strategy in place for handling region splits. HBase-generated region splits is a beautiful thing with respect to automatic scalability but if you're doing lots of writes, you'll want to control when your regions split. You can handle it in a couple of ways:

  - *Increase the region size parameter*, `hbase.hregion.max.filesize`. The default size was increased to 10GB recently, though, so it depends on the amount of data you want to ingest.

  - *Pre-split your regions to distribute them across the cluster*. You can do this if you have enough RegionServers. We discuss this technique in the next section.

- **Mixed workloads**: If your workload is mixed, you're in good company! Most clusters serve more than a single purpose. For mixed workloads, you need to follow best practices. First and foremost you'll need a good row key design to match your table access patterns. (We discuss row key design in the later section "The importance of row key design.") We've already discussed compression which improves performance.

Pre-Splitting Your Regions

HBase scales automatically by splitting regions when their size reaches the value configured in the `hbase.hregion.max.filesize` parameter. Regions are evenly distributed across the cluster by the load balancer process which runs on the MasterServer. This automation is very valuable for most HBase use cases, but you may (during bulk ingest operations or heavy writes, for example) want to manually control the whole process. In this case, you would set the `hbase.hregion.max.filesize` parameter to a very high value that you do not anticipate you will reach. After this is done, you can then manually split and compact your regions using the HBase shell commands. (See the "Tuning major compactions" section later in this chapter for more details.) You may also choose to pre-split your table(s) and distribute them across all available RegionServers right up front so that you can leverage the full power of the cluster immediately. If this tuning concept fits your application, then you can leverage one of the approaches in this list.

- **Use the HBase shell to create a table with pre-split regions, like this**:

  ```
  hbase(main):021:0> create 'Pre-Splits-Table',
          'OneColumnFamily', { SPLITS => ['A999',
          'B999', 'C999', 'D999'] }
  0 row(s) in 1.1720 seconds
  ```

- Note that you can also create a `splits` file where each line has a starting row key and then point to the file using the HBase shell `create` *`table`* command, as in this example.

  ```
  hbase(main):021:0> create 'Pre-Splits-Table',
          'OneColumnFamily', { SPLITS_FILE => 'mySplitsFile' }
  ```

- **Leverage the** org.apache.hadoop.hbase.util.RegionSplitter **utility**.

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

For documentation on the utility go to: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/ util/RegionSplitter.html.

- **Leverage a** createTab**le method from the** org.apache.hadoop.hbase.client.HBaseAdmin **class**

TIP If you decide that manual intervention into HBase region splitting is right for you, check out Hannibal, a very cool little tool for monitoring region splitting that helps you better manage the overall process. Hannibal is on GitHub at https://github.com/sentric/hannibal.

The Importance of Row Key Design

Proper row key design is central to creating any table in HBase. How you design the row key affects your performance, how you query your data, and the complexity of your application or client access approach. Also, if you find that you need to transition from a relational data model to an HBase model, you'll find that "intelligent" row keys are quite helpful. We stress this fact in the section "Transitioning from a RDBMS To HBase," earlier in this chapter. Given the importance of row key design, we could dedicate an entire section or even chapter to the subject but the point of this section is to highlight the key points around HBase deployment and performance tuning and then point you to further information where appropriate. So as you might expect by now, the Apache HBase online guide has an entire section dedicated to row key design case studies and you'll definitely want to review it for a thorough understanding of row key design. It's part of Apache's HBase and schema design overview; see http://hbase.apache.org/book/schema.casestudies.html.

What we want to do here is highlight the key points and considerations for proper row key design and introduce you to some tools at your disposal.

Making Your Row Key Fit Your Query Patterns

The first thing you have to consider is how you intend to query data that's stored in the table. Will the row key enable targeted access to a particular row you're looking for or will you have to scan large numbers of rows and look for the key value pair you need? Remember that HBase is row level atomic (meaning operations like `get`, `put,` and `scan` are guaranteed to successfully complete or completely fail — no partial results allowed), so it may be critical to target individual rows rather than perform scans for certain queries. Can you combine two or more unique identifiers to create a composite row key? When we show you how to work with the service order database earlier in this chapter, we combined the service order number with the customer number to create an intelligent composite row key, like this

A100|00001 Customer Name Contact Info Status

This approach enables queries based only on product number or customer number. This can be accomplished by leveraging HBase row key filters. Listing 12-12 shows simple table that illustrates this row key design.

Listing 12-12: Illustrating Intelligent Row Key Design

```
hbase(main):124:0> scan 'RowKeyTest'
ROW                 COLUMN+CELL
 A100|00005             column=cf:service-order, timestamp=1373463418241, value=brokenc
 B100|00003             column=cf:service-order, timestamp=1373463447048, value=brokeng
 B102|00004             column=cf:service-order, timestamp=1373463409362, value=brokenb
 C201|00001             column=cf:service-order, timestamp=1373463173365, value=brokena
4 row(s) in 0.0140 seconds
Now, use the <span cssStyle="text-decoration:line-through"<scan command
              to determine whether a service call has been placed for product
              number <span cssStyle="text-decoration:line-through">A100.

hbase(main):127:0> scan 'RowKeyTest', { FILTER => PrefixFilter.new(Bytes.
                 toBytes('A100')) }
ROW                 COLUMN+CELL
 A100|00005             column=cf:service-order,
                 timestamp=1373463418241, value=brokenc
1 row(s) in 0.0090 seconds
Use the same <span cssStyle="text-decoration:line-through">scan command
             to determine whether customer number <span cssStyle="text-
             decoration:line-through">00001 has placed a service call.

hbase(main):128:0≫ scan 'RowKeyTest', { FILTER => RowFilter.
                 new(CompareFilter::CompareOp.valueOf('EQUAL'),
                 SubstringComparator.new('00001')) }
ROW                                 COLUMN+CELL
 C201|00001                         column=cf:service-order, timestamp=1373463173365, value=brokena
1 row(s) in 0.0080 seconds
```

The two examples above illustrate how awesome composite row keys can be when combined with filters! Instead of having multiple tables or multiple rows to store customer and service order information, you can store it all in a single row and single table, thereby reducing overall disk and network traffic within your HBase cluster. Targeted or pointed HBase queries are the goal; you don't want to be doing extra IO operations

Reprinted for MC2WS/6275136415, Verizon John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited

while you search for your data.

Making Your Row Key Design Leverage the Performance Potential of the Cluster

A common challenge you face when designing your row keys is region *hotspotting*, where one or more RegionServers get overloaded with requests while the others sit idle. This is not what we want to see in HBase-land; we'd rather have every RegionServer pulling its own weight so users see the maximum performance! This performance issue will occur during large sequential writes or when you are reading continually from a small subsection of the table.

REMEMBER You learned about the MasterServer web interface at `http://bivm:60010/` in the "Taking HBase for a Test Run" section, and about the byte lexicographical sorting of row keys in the "Understanding the HBase Data Model" section. Now we're showing you some practical uses for this knowledge.

Now that the regions are pre-split (Step 1 we completed in the "Pre-Splitting your regions" section), Step 2 involves the row key. Keep in mind that, with byte lexicographical sorting, row keys are sorted from left to right. You can pre-split your tables, but if you don't ensure that your row keys are designed to distribute evenly across the splits, you'll still have a region hotspotting problem.

TECHNICAL STUFF HBase is a highly configurable and therefore a flexible technology, and the Load Balancer is no exception. You can either let the MasterServer automatically balance your regions or manually control the balancer via the shell.

Tuning Major Compactions

We introduce you to minor and major *compactions* — the process by which HBase cleans up after itself — in the "Understanding the HBase Architecture" section, earlier in this chapter, but in this section we want to briefly explain how you can control major compactions, because the impact to cluster performance is "major" — pun intended!

The approach is very straightforward — simply turn off automatic major compactions and issue the command manually against your tables at an appropriate time. To turn off major compactions, set the `hbase.hregion.majorcompaction` parameter to `0` in your `hbase-site.xml` file and restart HBase. To manually run a major compaction, simply issue the `major_compact` command from the HBase shell.

```
hbase(main):018:0> major_compact 'CustomerContactInfo'
0 row(s) in 0.0480 seconds
```

An obvious solution would be to script the major compaction shell commands and run the script using a scheduling utility like cron at the appropriate time. Simply put the preceding command in a file, name it `major-compact.rb`, add an `exit` command and execute it with this command:

```
$INSTALL_DIR/bin/hbase shell major-compact.rb
```

You can also trigger a major compaction via one of several `major_compact` methods in the Java client `org.apache.hadoop.hbase.client.HBaseAdmin` class.

TIP For a quick review of the HBaseAdmin class (it's well worth your time), see: http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HBaseAdmin.html.

How HBase is being used in the marketplace

The truly innovative technology HBase is a vital part of the Hadoop ecosystem. Here's a web page for you to check out: http://wiki.apache.org/hadoop/Hbase/PoweredBy. If you're wondering how HBase is now used in the marketplace, you'll be pleasantly surprised to know that you've probably already counted on HBase as part of your social media experience. Popular sites such as Facebook, Yahoo, Twitter, Meetup, StumbleUpon, Adobe, and many others now leverage HBase in production today. We trust that you'll be inspired to leverage HBase for your big data storage needs and start contributing to the ever growing and vibrant HBase community!

Reprinted for MC2WS/6275136415, Verizon
John Wiley & Sons (US), John Wiley & Sons, Inc. (c) 2014, Copying Prohibited