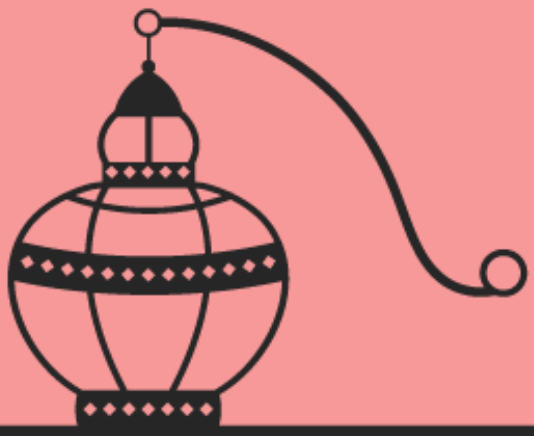


HBase

Succinctly

by Elton Stoneman

Syncfusion | Technology Resource Portal

HBase Succinctly

By

Elton Stoneman

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Darren West, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Elton Stoneman is a Software Architect, Microsoft MVP, and Pluralsight author.

He has been connecting systems since 2000, and has spent the last few years designing and building big data solutions for different clients with a variety of Hadoop technologies, and a mixture of on-premise and cloud deliveries.

His latest Pluralsight courses have covered big data in detail on Microsoft's Azure cloud, which provides managed clusters for Hadoop and parts of the Hadoop ecosystem, including Storm, HBase, and Hive.

HBase Succinctly is Elton's first eBook for Syncfusion, and is accompanied by source code on GitHub and an HBase container image on the Docker Hub.

You can find Elton online on his blog and tweeting @EltonStoneman.

Chapter 1 Introducing HBase

What is HBase?

HBase is a NoSQL database that can run on a single machine, or a cluster of servers for performance and fault-tolerance. It's built for scale—HBase tables can store billions of rows and millions of columns—but unlike other big data technologies, which are batch-oriented, HBase provides data access in real-time.

There are a few key concepts which make HBase different from other databases, and in this book we'll learn all about row key structure, column families, and regions. But HBase uses the same fundamental concepts as other database designs, so it's straightforward to pick it up.

HBase is free, open-source software from the Apache Foundation. It's a cross-platform technology so you can run it on Windows, Linux, or OS/X machines, and there are hosted HBase solutions in the cloud from both Amazon Web Services and Microsoft Azure. In this book we'll use the easiest option of all—running HBase in a Docker container.

You can connect to HBase using many client options, as shown in Figure 1. The Java API is the first-class citizen, but with the Thrift and REST APIs, you can connect with practically any language and platform. We will cover all three APIs in this book, together with the command-line interface that ships with HBase.

Figure 1: HBase APIs and Client Platforms

Why Choose HBase?

HBase is a very flexible and powerful database that integrates with a wide range of technologies and supports many different use cases. HBase leverages the Hadoop Distributed File System for high availability at the storage layer, but presents a real-time interface, so it's useful for transactional systems as well as big data problems.

In a production system, HBase has more complex infrastructure requirements than many databases—there are three server roles (Zookeeper, Master and Region Servers), and for reliability and performance, you need multiple nodes in each role. With a well-designed database, you can scale HBase just by adding more servers, and you should never need to archive old data.

HBase is a Java system that can run in simplified modes for development and test environments, so your non-production landscape can be modest. You can containerize the database, which makes for easy end-to-end testing in your build process and a high level of quality assurance for your code.

HBase Data Structure

HBase was inspired by Google's Big Table, which is the storage technology used for indexing the web. The architecture meets the need for real-time random access to specific items, in databases with terabytes – or even petabytes – of data. In HBase tables with hundreds of millions of rows, you can still expect sub-second access to individual rows.

Tables in HBase are partly structured; they don't have a rigid schema like a SQL database where every column has a data type, but they do have a loose schema where the general structure of the data is defined. In HBase, you define a table in terms of column families, which can contain many columns.

In HBase, tables are collections of similar data, and tables contain rows. Every row has a unique identifier—the row key—and zero or more columns from the column families defined for the table. Column families are dynamic, so the actual columns that contain data can be different from row to row.

A Sample HBase Table

Table 2 shows the schema for an HBase table called **`social-usage`**, which records how much time people spend on social networks. The structure is very simple, containing just a table name and a list of column family names, which is all HBase needs:

Table 1: Table Structure in HBase

Table Name	social-usage
Column Families	i
	fb
	tw
	t

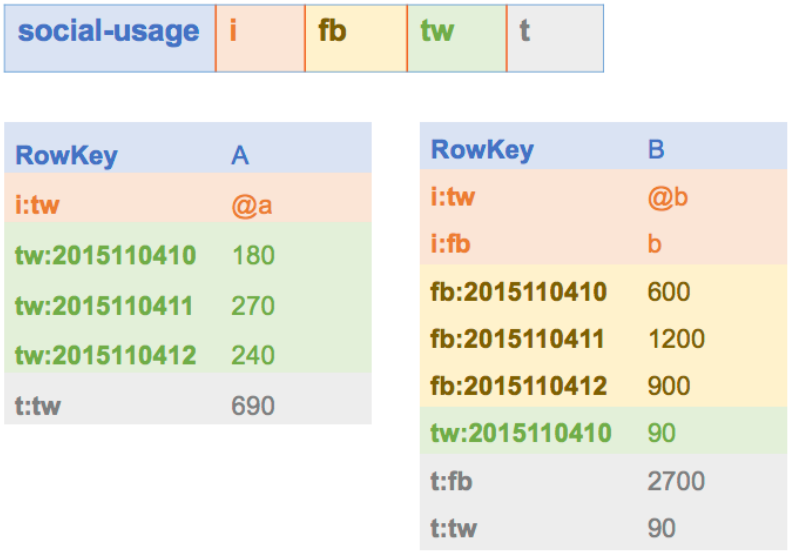
The table will record a row for each user, and the column families have cryptic names. Column names are part of the data that gets stored and transferred with each row, so typically they're kept very short.

This is what the column families are used for:

- **`i`** = identifiers, the user's ID for different social networks

- **`fb`** = Facebook, records the user's Facebook activity
- **`tw`** = Twitter, records the user's Twitter activity
- **`t`** = totals, sums of the user's activity

Figure 2 shows two sample rows for this table for users with very different usage profiles:



social-usage	i	fb	tw	t

RowKey	A
i:tw	@a
tw:2015110410	180
tw:2015110411	270
tw:2015110412	240
t:tw	690

RowKey	B
i:tw	@b
i:fb	b
fb:2015110410	600
fb:2015110411	1200
fb:2015110412	900
tw:2015110410	90
t:fb	2700
t:tw	90

Figure 2: Sample HBase Rows

We can see the power of HBase from these two rows, which tell us a huge amount from a very simple table schema. User A, in the row on the left, has a Twitter account and uses it a lot, but isn't on Facebook. User B, in the row on the right, has accounts for both networks, but spends much more time on Facebook.

Column Families and Qualifiers

A column family is like a hash table or a dictionary. For each row, the column family can contain many values (called cells), which are keyed by name (called the column qualifier). The column families available in a table are fixed, but you can add or remove cells to a column family in a row dynamically.

That means column qualifiers can be used as data stores, as well as cell values. In the **`fb`** column family in the sample, one row has a cell with the qualifier **`fb:2015110410`**, and the value **`600`**. HBase doesn't restrict how you name the qualifiers, so in this case I'm using a date period—in the format *year*, *month*, *day*, *hour*—and the cell value records the total usage in seconds during that hour.

For user B, the Facebook usage column family tells us they spent 10 minutes (600 seconds in the cell value) on Facebook between 10:00 and 11:00 on 4th November 2015, and 20 minutes between 11:00 and 12:00. There's no cell with the qualifier **`fb:2015110409`**, which tells us the user wasn't on Facebook between 09:00 and 10:00.

HBase Table Design

HBase is hugely flexible, but the primary way to access data is by the row key, so the design of your tables is critical to the performance of your solution. In the **`social-usage`** table, we store a single row for each user, and for each reporting period, we will add more cells to the column families.

This is called a **wide table** design—we have a (relatively) small number of rows, but rows can potentially have lots of columns. If we record usage for hourly periods, then for an active user we could be adding 98 cells every week (one cell for every hour of the week). Over a few years, we'll have tens of thousands of cells for that user's row—which is not a problem for HBase to store, but could be difficult for us to use efficiently.

If we wanted to sum the user's activity for a particular month, we'd need to read the row and include just that month's cells by filtering the results by the column qualifier name. Filtering based on columns is much more expensive for HBase than filtering based on rows, so an alternative design would be to use a **tall table** design instead.

Tall tables have larger numbers of rows, each with smaller numbers of columns. Instead of having one row for each user, we could have one row for each period for each user, which would give us the data in Figure 3:

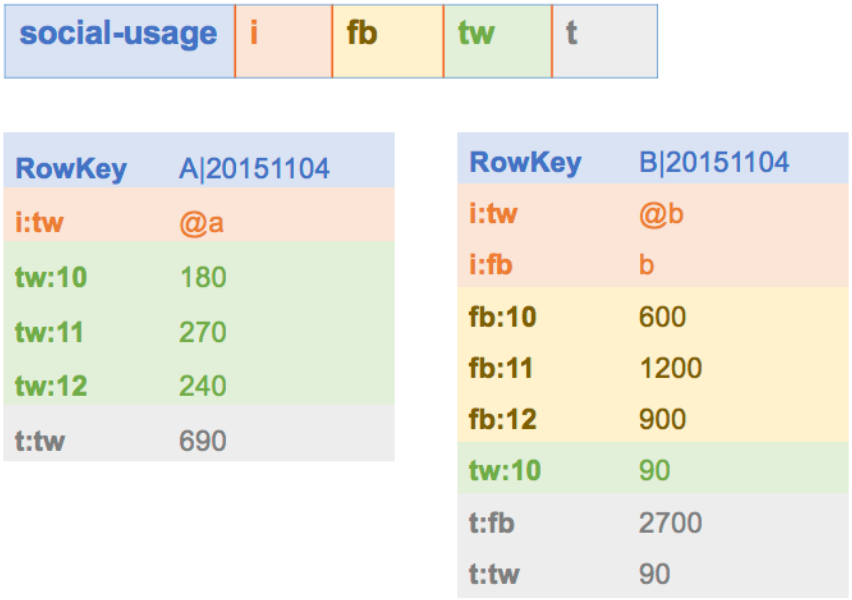


Figure 3: Tall Table Design

Now the row contains a user name and a period, which is the date of the data. Column names are much shorter, as we are only capturing the hour in the column name, and all the data in the row is for the same date. For active users we will only have tens of columns in each row, and hundreds of rows can record a whole year's data.

This design uses a composite row key, in the format **{*userID*}|{*period*}**, so if we want to sum a user's usage for a period, we can do it with a row scan, which is a much cheaper operation than a column filter, both of which we'll see later.

Note: You need to design HBase tables to support your expected data access patterns, but typically tall tables are preferable to wide ones.

Code Listing 1 shows the Data Definition Language (DDL) statement to create the **social-usage** table. It contains the minimum amount you need—the table name, followed by the column family names:

Code Listing 1: Create DDL statement

```
create 'social-usage', 'i', 'fb', 'tw', 't'
```

The schema definition in HBase is deliberately vague. The format of the row key and column qualifiers are not specified and there are no data types.

The conventions for using the table are not explicit in the schema, and in fact we could use the exact same schema for a wide table (using just the user ID for the row key), or a tall table (using user ID and period for the row key).

Data Types

There are no data types in HBase; all data is stored as byte arrays—cell values, column qualifiers, and row keys. Some HBase clients carry that through in their interface; others abstract the detail and expose all data as strings (so the client library encodes and decodes the byte arrays from HBase).

Code Listing 2 shows how a row in the **social-usage** table looks when you access it through the HBase Shell, which decodes byte arrays to strings. Code Listing 3 shows the same row via the REST API, which exposes the raw byte arrays as Base64 encoded strings:

Code Listing 2: Reading Data with the HBase Shell

```
hbase(main):006:0> get 'social-usage', 'A|20151104'

COLUMN                CELL

 tw:10                timestamp=1447622316218, value=180

1 row(s) in 0.0380 seconds
```

Code Listing 3: Reading Data with the REST API

```
$ curl -H Accept:application/json http://127.0.0.1:8080/social-
usage/A%7C20151104

{"Row":[{"key":"QXwyMDE1MTEwNA==","Cell":[{"column":"dHc6MTA=","timestamp":
1447622316218,"$":"MTgw"}]}]}
```

Note that the row key has to be URL-encoded in the REST call, and all the data fields in the JSON response are Base64 strings.

If you work exclusively with one client, then you can store data in the platform's native format. Code Listing 4 shows how to save a decimal value (1.1) in Java to a cell in HBase:

Code Listing 4: Storing Typed Data in HBase

```
Put newLog = new Put(rowKey);

newLog.addColumn(family, qualifier, Bytes.toBytes(1.1));

access_logs.put(newLog);
```

This can cause issues if you work with many clients, because native type encoding isn't the same across platforms. If you read that cell using a .NET client, and try to decode the byte array to a .NET decimal value, it may not have the same value that you saved in Java.

> ***Tip: If you're working in a cross-platform solution, it's a good idea to encode all values as UTF-8 strings so that any client can decode the byte arrays from HBase and get the same result.***

Data Versions

The last feature of HBase we'll cover in this introduction is that cell values can be versioned—each value is stored with a UNIX timestamp that records the time of the last update. You can have multiple versions of a cell with the same row key and column qualifier, with different timestamps recording when that version of the data was current.

HBase clients can fetch the latest cell version, or the last X versions, or the latest version for a specific date. When you update a cell value, you can specify a timestamp or let HBase use the current server time.

The number of cell versions HBase stores is specified at the column-family level, so different families in different tables can have different settings. In recent versions of HBase, the default number of versions is one, so if you need data versioning, you need to explicitly specify it in your DDL.

With a single-version column family, any updates to a cell overwrite the existing data. With more than one version, cell values are stored like a last-in, first-out stack. When you read the cell value you get the latest version, and if you update a cell that already has the maximum number of versions, the oldest gets removed.

Summary

In this chapter, we had an overview of what HBase is, how it logically stores data, and the features it provides. In the next chapter, we'll see how to get up and running with a local instance of HBase and the HBase Shell.

Chapter 2 HBase and the HBase Shell

HBase Run Modes

You can run HBase on a single machine for development and test environments. HBase supports three run modes: Standalone, Pseudo-Distributed, and Distributed. Distributed mode is for a full cluster, backed by HDFS, with multiple servers running different components of the HBase stack, which we will cover in Chapter 7 The Architecture of HBase."

Standalone mode is for a single machine, where all components run in a single Java Virtual Machine, and the local file system is used for storage rather than HDFS. Pseudo-Distributed mode runs each HBase component in a different JVM on one server, and it can use HDFS or the local filesystem.

Tip: Pseudo-Distributed mode is a good option for running locally—you get a production-style separation between components, but without having to run multiple machines.

This HBase documentation covers installing and running HBase locally, so I won't duplicate it here, but the easiest way to run HBase locally is with Docker. There are a few HBase images on the Docker Hub, including one of my own, which I've built to go along with this course.

Docker is an application container technology. A container is a fast, lightweight unit of compute that lets you run multiple loads on a single machine. Containers are conceptually similar to virtual machines, but very much lighter on disk, CPU, and memory usage. Docker runs on Linux, OS/X, and Windows machines. You can get installation instructions here. The Docker Hub is a public registry of pre-built images, and my image for this book is available here.

Running HBase in a Docker Container

The advantage of using a Docker container is that you can spin up and kill instances with very little overhead, and you don't need to worry about any software or service conflicts with your development machine.

The image **`hbase-succinctly`** is one I've put one together specially to go along with this book, which sets up the services we'll be using. To use that image, install Docker and execute the command in Code Listing 5:

Code Listing 5: Running HBase in Docker

```
docker run -d -p 2181:2181 \

 -p 60010:60010 -p 60000:60000 \

 -p 60020:60020 -p 60030:60030 \

 -p 8080:8080 -p 8085:8085 \

 -p 9090:9090 -p 9095:9095 \

 --name hbase -h hbase \

 sixeyed/hbase-succinctly
```

Some of the settings in the **`docker run`** command are optional, but if you want to code along with the sample in this book, you'll need to run the full command. If you're not familiar with Docker, here's what the command does:

- Pulls the image called **`hbase-succinctly`** from the **`sixeyed`** repository in the public Docker Hub
- Runs the image in a container locally, with all the key ports exposed for the servers and Web UI
- Names the image **`hbase`**, so we can control it with other Docker commands without knowing the container ID Docker will assign
- Gives the image the hostname **`hbase`**, so we can access it using that name

The first time that runs, it will take a while to pull the image from the registry to your local machine, but for future runs, the container will start in a few seconds, and you'll have a functioning HBase instance with the Java, REST, and Thrift APIs running.

Note: This Docker command exposes all the ports for Zookeeper, the HBase external APIs, and the Web UIs. In case they conflict with other servers on your machine, you can change the local port by altering the number before the colon on the –p argument. For example, to have Thrift listening on port 91 on the Docker host, instead of 9090, use –p 91:9090.

This image uses the local filesystem for HBase data storage, which isn't preserved when the container is killed, so you can reset your database back to the original state with **`docker kill hbase`** and then repeat the **`docker run`** command.

Using the HBase Shell

HBase comes with a command line interface, the HBase Shell. The Shell can't be used on a remote machine, so you need to run it from the local machine (for HBase Standalone and Pseudo-Distributed modes), or by logging on to the Master Server (for Distributed mode).

From the HBase bin directory, run **`hbase shell`** to start the Shell. If you're running HBase through my Docker image, connect by running the interactive command in Code Listing 6:

Code Listing 6: Running HBase Shell in Docker

```
docker exec -it hbase hbase shell
```

The HBase Shell is based on JRuby, and you can use it to execute script files, as well as interactive commands. A large number of commands are available in the Shell. It's the ideal place to start with HBase, and you will also use it in production for administering the cluster.

We'll cover the main commands for defining tables and reading and writing data in this chapter.

Working with Tables

Three shell commands will get you started with a new HBase database:

- **`list`**
- **`create`**
- **`describe`**

To see all the tables in the database, run **`list`** and you'll see the output in two forms: as a plain text list, and as an array representation. Code Listing 7 shows the sample output:

Code Listing 7: Listing Tables

```
 hbase(main):001:0> list
 TABLE
 api-logs
 social-usage
 2 row(s) in 0.2550 seconds


=> ["api-logs", "social-usage"]
```

The **`list`** command only supplies the table names, and no other details. For a large database you can filter the output by supplying a regular expression for the command to match on table name; e.g. **`list 'so.*'`** will show table names starting 'so'.

To create a table, use the **create** command, specifying the table name and the column family names. Optionally, you can pass configuration settings for the column families, and this is how you can change the number of cell versions that HBase stores. Code Listing 8 shows two **create** commands:

Code Listing 8: Creating Tables

```
hbase(main):007:0> create 'with-default-config', 'cf1'
0 row(s) in 1.2380 seconds


=> Hbase::Table - with-default-config
hbase(main):008:0> create 'with-custom-config', {NAME =>'cf1',
VERSIONS=>3}
0 row(s) in 1.2320 seconds


=> Hbase::Table - with-custom-config
```

The table **with-default-config** has a single column family, **cf1**, with no configuration specified, so it will use the HBase defaults (including having a single cell version). Table **with-custom-config** also has a column family named **cf1**, but with a custom configuration setting specifying three cell versions.

Note: The HBase Shell uses Ruby syntax, with curly braces to define objects and properties specified as name-value pairs, separated with '=>' for the value.

To see the configuration for a table, use the **describe** command. The output tells you whether the table is enabled for client access, and includes all the column families with all their settings, as in Code Listing 9:

Code Listing 9: Describing Tables

```
hbase(main):009:0> describe 'with-custom-config'
Table with-custom-config is ENABLED
with-custom-config
COLUMN FAMILIES DESCRIPTION
{NAME => 'cf1', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'ROW',
REPLICATION_SCOPE => '0', VERSIONS => '3', COMPRESS
ION => 'NONE', MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS
=> 'FALSE', BLOCKSIZE => '65536', IN_MEMORY =>
'false', BLOCKCACHE => 'true'}
```

```
1 row(s) in 0.1240 seconds
```

Now that we have some tables with different configurations, we can start adding and retrieving data with the HBase Shell.

Writing Cell Values

With the HBase Shell, you use the **`put`** command to write data for individual cells, specifying the table name, row key, column name, and cell value. The command in Code Listing 10 sets the value of the **`data`** column in the **`cf1`** column family to **`v1`** for the row with key **`rk1`**:

Code Listing 10: Writing data

```
hbase(main):019:0> put 'with-default-config', 'rk1', 'cf1:data', 'v1'

0 row(s) in 0.0070 seconds
```

The **`put`** command works like an insert/update. If the row doesn't exist, the command creates it, creates the column, and sets the cell value. If the row exists but the column doesn't, the command adds the column to the row and sets the cell value.

Using the default column family configuration with a single cell version, if the row and column already exist, then the command overwrites the existing value with the new one. But with a column family with more than one version, the **`put`** command adds a new cell version with the value and sets the timestamp.

You can optionally specify a custom timestamp value to use, as a long integer after the cell value. Code Listing 11 explicitly specifies a timestamp in the past—one millisecond after the UNIX epoch (January 1, 1970):

Code Listing 11: Writing Data with a Timestamp

```
hbase(main):020:0> put 'with-default-config', 'rk1', 'cf1:data', 'v0', 1

0 row(s) in 0.0060 seconds
```

Tip. Be careful about specifying timestamps in updates. If your timestamp is earlier than the other versions of the cell (even if there's only one version configured in the

column family), your data is assumed to be older than the current data, and the cell value won't be overwritten—and your update gets lost.

Reading Data

You use the **`get`** command to read the data on a row, but unlike the **`put`** command, you can use **`get`** to read multiple cell values. To read a whole row, use **`get`** with the table name and row key. HBase will return the most recent version of every cell value in the row, as shown in Code Listing 12:

Code Listing 12: Reading a Whole Row

```
hbase(main):017:0> get 'social-usage', 'a'
COLUMN                          CELL
 i:tw                           timestamp=1446655378543,
value=@EltonStoneman
 t:tw                           timestamp=1446655459639, value=900
 tw:2015110216                  timestamp=1446655423853, value=310
 tw:2015110316                  timestamp=1446655409785, value=270
 tw:2015110417                  timestamp=1446655398909, value=320

5 row(s) in 0.0360 seconds
```

You can restrict the results to specific columns or families, with a comma-separated list of column family names or qualifiers after the row key. In Code Listing 13, we return the whole **`i`** family, and just one column from the **`tw`** family:

Code Listing 13: Reading Specific Columns

```
hbase(main):022:0> get 'social-usage', 'a', 'tw:2015110316', 'i'
COLUMN                          CELL
 i:tw                           timestamp=1446655378543,
value=@EltonStoneman
 tw:2015110316                  timestamp=1446655409785, value=270

2 row(s) in 0.0090 seconds
```

You can also pass objects with many properties, instead of just strings, to return more specific details from the rows. I can return multiple versions for a column by specifying the **`VERSIONS`** property, with the number to return, as shown in Code Listing 14:

Code Listing 14: Reading Multiple Cell Versions

```
hbase(main):027:0> get 'with-custom-config', 'rk1', {COLUMN =>'cf1:data',
VERSIONS => 3}
COLUMN                          CELL
 cf1:data                       timestamp=1446655931606, value=v3
 cf1:data                       timestamp=1446655929977, value=v2
 cf1:data                       timestamp=1446655928221, value=v1
3 row(s) in 0.0120 seconds
```

Scanning Rows

Rows in HBase tables are physically stored in order, sorted by row key. We'll look at the structure of the data, and the indexing approach in Chapter 8 Inside the Region Server," but for now, we just need to know that:

- Sorted tables mean fast direct access by key.
- Sorted tables mean slow searches by value.

In fact, you can't search for row keys matching an arbitrary pattern in HBase. If you have a table storing system access logs, with row keys that start **`{systemID}|{userID}`**, you can't search for the logs for one particular user because the user ID is in the middle of the row key.

With HBase, you find matching rows by scanning the table, providing start and end boundaries for the scan. Logically, HBase then works like a cursor, positioning the table to the start row (or a partial match of the start row) and reading until the end row.

The **`scan`** command is straightforward, but the results can be unexpected for new users of HBase. Table 2 shows some sample data from my **`access-logs`** table:

Row key
jericho|dave|201510
jericho|elton|201510
jericho|elton|201511
jericho|fred|201510

Table 2: Sample Row Keys

We have four row keys here, all for the system called Jericho, for users Dave, Elton and Fred, for the months of October and November, 2015. The rows are listed in Table 2 in the same lexicographical order as they are stored in the HBase table.

To find all the rows for Jericho access, I can scan the table with a **STARTROW** value, as in Code Listing 15:

Code Listing 15: Scanning with a STARTROW

```
 hbase(main):006:0> scan 'access-logs', {STARTROW => 'jericho'}
 ROW                              COLUMN+CELL
  jericho|dave|201510             column=t:3015, timestamp=1446706437576,
 value=60
  jericho|elton|201510            column=t:3015, timestamp=1446706444028,
 value=700
  jericho|elton|201511            column=t:0416, timestamp=1446706449473,
 value=800
  jericho|fred|201510             column=t:0101, timestamp=1446706454401,
 value=450

4 row(s) in 0.0540 seconds
```

To find all Elton's access of Jericho, the **STARTROW** needs to contain the user ID, and I need to add an **ENDROW** value to exclude any rows after Elton. This is where the scan gets more interesting. I could use an **ENDROW** value of **jericho|f**, and that would get me just Elton's logs, as in Code Listing 16:

Code Listing 16: Scanning with an ENDROW

```
 hbase(main):010:0> scan 'access-logs', {STARTROW => 'jericho|elton',
 ENDROW => 'jericho|f'}
 ROW                              COLUMN+CELL
  jericho|elton|201510            column=t:3015, timestamp=1446706444028,
 value=700
  jericho|elton|201511            column=t:0416, timestamp=1446706449473,
 value=800

2 row(s) in 0.0190 seconds
```

That query works for now, but if we later add rows for a user called Ernie, when I run the same query, it would return their logs too. So the **STARTROW** and **ENDROW** for a scan need to be as specific as you can make them, without losing the flexibility of your query.

A query that will return all of Elton's logs for any time period, could use **STARTROW => 'jericho|elton|'** and **ENDROW => 'jericho|elton|x'**. Knowing your ASCII character codes helps here.

The pipe character has a higher value than any alphanumeric characters, so including the pipe after the username ensures no other users' logs will creep into the scan. Character **x** is higher than any numbers, so adding that at the end of the scan means the query will return rows for any year.

One final point about scans: the upper boundary is an 'up to' value, not an 'up to and including' value. If I want to see all Elton's access in October and November of 2015, the query in Code Listing 17 isn't correct:

Code Listing 17: Scanning 'up to' the ENDROW

```
 hbase(main):014:0> scan 'access-logs', {STARTROW =>
 'jericho|elton|201510', ENDROW => 'jericho|elton|201511'}
 ROW                              COLUMN+CELL
  jericho|elton|201510            column=t:3015, timestamp=1446706444028,
 value=700

1 row(s) in 0.0140 seconds
```

Using an **ENDROW** value of **jericho|elton|201511** means HBase reads up to that row and then stops. To include rows from 201511, I need an end row value that goes further than those rows, I could use **jericho|elton|201512** in this case.

Summary

In this chapter, we got started with HBase, running it locally from a Docker container. Then we connected using the HBase Shell and used table management commands to list all tables in the database, create new ones, and describe their configuration.

We saw how to write data using the **put** command, and read it using the **get** command, including working with multiple versions of a cell. And we saw how to scan tables for a range of rows, using start and end row keys for the boundaries of the scan.

We'll return to the HBase Shell in Chapter 9 Monitoring and Administering HBase," which covers more advanced topics. In the next chapter, we'll look more closely at HBase table design, row keys, and regions.

Chapter 3 HBase Table Design

Row Key Structure

There are no relations between tables in HBase, and there are no indexes, data types, default values, computed columns, or any other fancy features you get in a modern SQL database, or even other NoSQL databases. Because of that, data modelling in HBase is deceptively simple—the only constructs you have to work with are tables, rows, and column families.

But access to data in HBase is always via the row key, so how you structure the key has a huge impact on the performance and usability of your database. In the **`access-logs`** table from the last chapter, I used the row key format **`{systemID}|{userId}|{period}`**. That format means the table is good for one read scenario, but bad for others.

If I typically read that table in the context of a specific system and user, then the row key format is great. To get a user's access logs, I can use a scan with boundaries that exclude all other users and systems, and I'll get a very quick response. Even if the table has hundreds of millions of rows, a scan that returns a few hundred rows will take seconds to run.

But if I want to find all the systems that *anyone* used on a particular *day*, this row key structure is not optimal. Because the period is at the end of the key, I can't set boundaries for a scan and I'll need to read every row. With hundreds of millions of rows, that will likely take hours. If that's how I want to use the table, then a row key structure of **`{period}|{systemID}|{userID}`** would be better.

That alternative structure has its own problems, as we'll see shortly. It also means queries by period are fast, but if I want to find one user's access logs for one system, then I'll be reading every row again.

Tip: The only "right" way to design row keys is to know how you'll be accessing the table, and model that access in the key structure. Often you'll need to compromise secondary read scenarios in favor of the primary scenario.

Read and Write Performance

Data access patterns are about what your queries are doing, and also how many queries you're running—and your row key structure has a big impact on that, too. HBase is a distributed database and it has the capacity to support high concurrency for reads and writes, but only if your table design allows it.

A single, logical table in HBase is actually split at the storage layer and stored in many parts, called regions. Access to the data in a region is provided by one instance of an HBase Region Server, and in a production environment, you will have many Region Servers running in a cluster. If you design your tables correctly, different regions can be hosted by different Region Servers, giving you high performance for concurrent reads and writes to that table.

But that also depends on your row key. Tables are split by row key, with each region having a start and end row key. If your design means your all your row keys start with similar values, then they won't be distributed among many regions and you won't get high concurrency.

Table 3 shows some more sample row keys in the access-logs table, using different structures (with spaces added around the pipes for readability):

Option 1: {period}|{systemID}|{userID}	Option 2: {systemID}|{userID}|{period}	Option 3: {userID}|{systemID}|{period}
201510 | jericho | dave	jericho | dave | 201510	dave | jericho | 201510
201510 | jericho | elton	jericho | elton | 201510	elton | jericho | 201510
201511 | discus | elton	discus | elton | 201511	elton | discus | 201511
201510 | jericho | fred	jericho | fred | 201510	fred | jericho | 201510

Table 3: Row Key Designs

With Option 1, every row starts with the same five characters; the distance between the rows is very small, so they will probably all be in the same region. That means they'll all be served by the same Region Server, and if we read and write a set of rows like that simultaneously, we won't balance the load across multiple Region Servers, and we won't get maximum performance.

Tip. Don't use sequential values (based on dates or timestamps) as row keys for tables that need high performance. Sometimes you need to use sequential values to support the way you want to read the data, but be aware that you are limiting concurrent performance when you do.

Option 2 is better—from just the first character, we have two different values. In this case, we might find three rows in one region and one in another. If we had many systems that we were logging access for, and the IDs were sufficiently different, we could have dozens of regions, and support a high level of concurrent read/write access, balanced across multiple Region Servers.

Option 3 is the best if we need maximum performance. We're likely to have more users than systems, so the superset of user IDs would be in the hundreds or thousands of values, and they're all likely to have good distance between them. We could have many hundreds of regions with this approach, and if we were to scale out a production cluster by adding more Region Servers, we would still have multiple regions in each server and support high concurrency.

Note: You need to balance performance considerations for concurrency with scan performance. A row key design that supports thousands of concurrent writes per second is good for event streams, but if it doesn't let you read data efficiently for your scenario, it's not the right choice.

Region Splits and Pre-Splitting

HBase runs its own administration jobs to maintain or improve performance as data volumes increase, and one part of that administration is splitting large tables into regions. Technically you can leave it to HBase to automatically create regions for your tables, but you can also do it yourself explicitly when you create tables.

With the simple **`create`** statements we used in Chapter 2, our tables are created with a single region, and they won't be split until they grow beyond the configured size for a region and HBase splits them in two. The maximum size varies, but it's typically in multiples of 128MB, which is usually too large—you've lost a lot of performance by the time the split happens, so it's better to pre-split your table.

Pre-splitting means you tell HBase, in the **`create`** statement, how many regions the table should start with, and the upper boundary for the row key in each region. That works most effectively if you can define boundaries, meaning each region should be approximately the same size.

That's not as hard as it may sound. Universally Unique IDs (UUIDs), or partial UUIDS, are a good choice for row keys (or the first part of row keys). With a hex representation, you can split on the first character and you'd have 16 regions in your table. Code Listing 18 shows the **`create`** command with the **`SPLITS`** property defining the boundaries for regions:

Code Listing 18: Create Table with Pre-split Regions

```
create 'access-logs', 't', {SPLITS => ['1', '2', '3', '4', '5', '6', '7',
'8', '9', 'a', 'b', 'c', 'd', 'e', 'f']}
```

The HBase load balancer does a good job of spreading regions around the server estate, so in a small cluster with four region servers, we can expect four **`access-logs`** regions running on each server; if we scale up to 16 region servers, we'd have a single region on each server, and we can expect high concurrency for reads and writes.

If your data is not evenly distributed among the regions, then you will get hot-spots—cases where some regions have more than their fair share of data, and have to process more load. HBase still runs the admin jobs for pre-split tables, so if you do have an oversize region, then at some point it will be automatically split by HBase.

Tip. You can create many more regions, but every region has its own memory and disk allocation, so there's a cost to having more regions. A good rule of thumb is to aim for around 100 regions in total per region server—so you should allocate more regions to the most performance-critical tables.

Using UUIDs will give you good data distribution among your regions if you generate them randomly, but if you want good distribution and want to use transactional IDs for keys, you can hash them to produce the row key.

Table 4 shows how rows from the **`access-logs`** table would be distributed if we build the row key from an MD5 hash of the user ID. MD5 gives a simple way of generating deterministic hashes; the algorithm is not suitable for securing data, but it's extremely unlikely that you'll get collisions with small input values like user IDs:

Row key	Part-hashed row key	Region
dave | jericho | 201510	16108 | jericho | 201510	1 (keys 0 to 1)
elton | jericho | 201510	d5fe7 | jericho | 201510	13 (keys d to e)
fred | jericho | 201510	570a9 | jericho | 201510	5 (keys 5 to 6)

Table 4: Part-hashing Row Keys

Column Families

As we'll see in Chapter 8 Inside the Region Server," column families within regions are the physical storage unit in HBase. All the columns in a single family for one region are stored in the same file, so columns that are typically accessed at the same time should be located in the same column family.

Column families can contain large numbers of columns, but like regions, there is an overhead with having many column families. Typically, one column family per table is all you need, and the official HBase documentation recommends no more than three families.

You add more than one column family to a table if the data has different access patterns, like in the **`social-usage`** table from Chapter 1, which has separate families for Facebook and Twitter usage.

In the original sample, I included extra column families to help illustrate how tables are structured in HBase, but in a real system, I would rationalize the design, removing the column families for identifiers and totals and storing that data as columns within the other families.

The revised design would have just two column families, **`fb`** and **`tw`**, consolidating all the data:

`fb` = Facebook, user's Facebook details and activity

fb:id = user ID
fb:t = total usage
fb:{period} = usage within that period
tw = Twitter, the user's Twitter details and activity
tw:id = user ID
tw:t = total usage
tw:{period} = usage within that period

This design would be suitable if we expected to capture similar amounts of data for each social network. But if the usage was heavily skewed, we could have a problem with the split of column families within regions.

If our table grew to 100 million rows and we were running a large cluster, we may decide to split it into 1,000 regions. That would give us 1,000 data files for the Facebook column family, and 1,000 for Twitter (that's not exactly correct, but we'll unpack that statement in Chapter 9).

Within those 100 million rows, if only one million contain any Twitter data, then our region splits will negatively impact performance when we query for Twitter data. If we do a large scan for Twitter rows, it could require reads across all 1,000 regions.

For one million rows, we might find optimal performance with just 100 regions, so by having multiple column families in the table, we've harmed performance for the less-populated family. In that case, the better design would be two tables, **facebook-usage** and **twitter-usage**, each with a single column family, so they can be independently tuned.

Tip. Design tables with a single column family unless you know the data will have separate access requirements but similar cardinality. Tables with two- or three-column families work best if the majority of rows have data in every column family.

Columns

All data in HBase is stored as byte arrays, so cell values can be representations of any type of data. Strings are the most portable data type; you can use a standard encoding (like UTF-8), and different clients will be able to work with the data in the same way.

If you work with a single HBase client, then using native data types rather than strings will give you a small optimization in the amount of storage you use, but it's probably a micro-optimization that's outweighed by the advantage of having a standard data type in all your columns.

Consistently using one data type makes your data access code simpler—you can centralize the encoding and decoding logic, and not have different approaches for different columns. Your string values could be complex objects too—storing JSON in HBase cells, which you then de-serialize in the client—is a perfectly valid pattern.

In HBase terms, whatever you store is a byte array, and although the client may interpret them differently, to the server the only difference with different data types is the amount of storage they use. HBase doesn't enforce a limit on the size of the byte array in a single cell, but you should keep cell sizes under 10MB for best performance.

Table 5 shows a sample HBase table we could use to store all the books in Syncfusion's Succinctly series, in a simple table with a single column family **b**:

Column Qualifier	Client Data Type	Contents
b:t	String	Book title
b:a	String	Author name
b:d	Long	Release date (UNIX timestamp)
b:c	Byte array	Cover image (PNG)
b:f	Byte array	Download file (PDF)

Table 5: Syncfusion's Succinctly Library

Counter Columns

There's one exception to HBase's bytes-in, bytes-out approach to cell values: counter columns. Counter columns live inside a column family in a table in the same way as other columns, but they are updated differently—HBase provides operations for atomically incrementing the value of a counter without having to read it first.

With the HBase Shell, the **incr** command increments a counter cell value, or creates a counter column if it doesn't exist. You can optionally specify an amount to increment; if you don't, then HBase will increment by one. Code Listing 19 shows this with two commands that add counter cells to the row **rk1**—the first adds a new cell, **c:1**, with the default increment of 1, and the second adds the cell **c:2** with an increment of 100:

Code Listing 19: Incrementing Counter Columns

```
hbase(main):006:0> incr 'counters', 'rk1', 'c:1'
COUNTER VALUE = 1
0 row(s) in 0.0130 seconds

hbase(main):007:0> incr 'counters', 'rk1', 'c:2', 100
COUNTER VALUE = 100

0 row(s) in 0.0090 seconds
```

The response from the Shell displays the value of the counter after updating it. Counters are a very useful feature, especially for high volume, high concurrency systems. In the **`access-logs`** table, I could use counter columns to record how much time the user spent on the system.

A system auditing component would need to add the current usage to any already recorded for the period, and if the user has multiple sessions open we could have concurrent updates for the same cell. If different instances of the audit component manually read the existing value, add to it and put updates at the same time, then we will have a race condition and updates will be lost.

With a counter column, each instance of the auditing component issues an increment command without having to read the existing value, and HBase takes care of correctly updating the data, preventing any race conditions.

Counter columns can be read in the same way as other cell values, although in the HBase Shell they are shown as a hex representation of the raw byte array, as shown in Code Listing 20, which reads the values set in the previous command. Note that the HBase Shell displays the data using an unusual ASCII/binary encoding, so **`x01`** is the value 1 in **`c:1`**, and **`x00d`** is the value 100 in **`c:2`**:

Code Listing 20: Counter Column Values

```
hbase(main):008:0> get 'counters', 'rk1'
COLUMN                          CELL
 c:1                            timestamp=1446726973017,
value=\x00\x00\x00\x00\x00\x00\x00\x01
 c:2                            timestamp=1446726979178,
value=\x00\x00\x00\x00\x00\x00\x00d

2 row(s) in 0.0140 seconds
```

Note: You should always create a counter column using the increment command from the client. If you create it as a normal column with a custom value and then try to increment it, you will get the following error: "Attempted to increment field that isn't

64 bits wide." This is HBase saying you can't increment a value that isn't in a counter column.

Summary

In this chapter we looked at the key parts of table design in HBase: structuring the row key, pre-splitting regions, and using columns and column families.

There is no single design that fits all HBase problems; you need to be aware of the performance and usage considerations, particularly with your row key design. You need to design tables based on the access patterns you expect, and it's not unusual to redesign your tables during development as you learn more about your requirements.

Now we have a good working knowledge of how HBase stores data; in the next few chapters, we'll look at accessing that data remotely, with the APIs that HBase provides out of the box: Java, Thrift, and REST.

Chapter 4 Connecting with the Java API

Overview

The native API for client connections to HBase is the Java API. The functionality can be divided into two parts—metadata and admin functions—which connect to the Master Server, and data access functions, which connect to the Region Servers (we'll cover those servers in more depth in Chapter 7 The Architecture of HBase").

You don't need to do anything to support the Java API from the server, other than ensure the ports are open (by default, 2181 for Zookeeper, 60000 for the Master, and 60020 for the Region Servers).

The HBase client package is in the Maven Repository, with JAR files for all the released versions of HBase. At the time of writing, the latest version is 1.1.2 (which is the version I use in the Docker container for the course), but 0.9x versions are common and still available in Maven.

Note: I won't cover getting up and running with Maven or a Java IDE here, but the source code for the book contains a NetBeans Java project with sample code that uses Maven.

Connecting to HBase with the Java Client

The HBase Java client is available in the Maven Central repository, and is versioned so the client version number matches the server version. For the current version of HBase (and the version running in the **`hbase-succinctly`** Docker container), we need a dependency to version 1.1.2 of the **`org.apache.hbase.hbase-client`** package, as in Code Listing 21:

Code Listing 21: The Maven HBase Client Dependency

```
<dependency>
   <groupId>org.apache.hbase</groupId>
   <artifactId>hbase-client</artifactId>
   <version>1.1.2</version>

 </dependency>
```

With the Java API, you start with a **`Configuration`** object, which contains the connection details for the server, and you use that when you create client objects for tables or administration. When you create a configuration object, by default it will look for an **`hbase-site.xml`** file in the resources of the running app that contains the configuration settings.

The **hbase-site.xml** configuration file also lives on the server, and you can use the same contents for the client connection—it specifies key details like the server ports and the Zookeeper quorum addresses. Code Listing 22 shows some sample properties from the site file:

Code Listing 22: The hbase-site.xml Config File

```
<configuration>
    <property>
        <name>hbase.cluster.distributed</name>
        <value>true</value>
    </property>
    <property>
        <name>hbase.master.port</name>
        <value>60000</value>
    </property>
    ...

</configuration>
```

The Java client only needs to know the Zookeeper quorum addresses; it gets the Master and Region Server addresses from Zookeeper.

Note: Zookeeper stores addresses as host names rather than IPs, so you need to be sure the host names of the Region Servers are accessible to the machine running the Java client. If you are using the Docker run command from Code Listing 6, then the host name will be hbase, and you should add a line to your hosts file, associating hbase to 127.0.0.1

You connect to HBase by using the **ConnectionFactory** class to create a **Connection** object, which uses the configuration from the local **hbase-site.xml** file, as in Code Listing 23:

Code Listing 23: Getting a Connection to HBase

```
Configuration config = HBaseConfiguration.create();

Connection connection = ConnectionFactory.createConnection(config);
```

You can set properties on the configuration object in code, but using the server's XML configuration file is more manageable.

Connection objects are expensive, and should be re-used. They are used to create **Table** and **Admin** objects for DML and DDL operations. The **Connection** object should be closed when your data access is finished, typically calling **close()** within a **finally** block.

Reading Data with Java

Using the **Connection** object, you can get a reference to a specific table, and you can use that to read and write data. The Java API works at the byte array level, so you need to decide on how to encode your data (natively, or converting all values to strings), and encode and decode all data.

There are helper classes in the HBase client package, which simplify that. Code Listing 24 shows how to get a **Table** object from the connection, and fetch a whole row using a **Get** object:

Code Listing 24: Reading a Row with Java

```
Table access_logs = connection.getTable(TableName.valueOf("access-logs"));
Get get = new Get(Bytes.toBytes("elton|jericho|201511"));

Result result = access_logs.get(get);
```

Note that the table name is created using the **TableName** class, and the row key in encoded to bytes using the **Bytes** utility class. When this code runs, the whole row will be in the result object, which contains the full byte array.

The **Result** class has a **listCells()** method, which returns a list of **Cell** objects; navigating the byte arrays in those objects is cumbersome, but another helper class, **CellUtil**, simplifies it. Code Listing 25 shows how to navigate the **Cell** array, printing out the column name and value for each cell:

Code Listing 25: Reading Cell Values with Java

```
for (Cell cell : result.listCells()){
  System.out.println(Bytes.toString(CellUtil.cloneFamily(cell)) + ":" +
                     Bytes.toString(CellUtil.cloneQualifier(cell)) + " = "
+
                     Bytes.toString(CellUtil.cloneValue(cell)));
}

//output -
//t:1106 = 120
//t:1107 = 650
```

The **Get** class can be used to return a restricted set of cells from the row. Code Listing 26 shows the use of the **addFamily()** method to return the cells in one column family for the row, and the **addColumn()** method to limit the response to a single cell.

Again, the identifiers need to by byte arrays, so the **Bytes** class is used to encode string values:

Code Listing 26: Reading Specific Cells with Java

```
get = new Get(Bytes.toBytes("elton|jericho|201511"));
get.addFamily(Bytes.toBytes("t"));
result = access_logs.get(get);
printCells(result);

//output - single column family:
//t:1106 = 120
//t:1107 = 650

get = new Get(Bytes.toBytes("elton|jericho|201511"));
get.addColumn(Bytes.toBytes("t"), Bytes.toBytes("1106"));
result = access_logs.get(get);
printCells(result);

//output - single column:
//t:1106 = 120
```

Tip. If you see code samples using the HTable class and instantiating them directly with Configuration objects, that code is deprecated in the API. The newer way, which I'm using, is with the ConnectionFactory, Connection, and Table classes.

Working with Scanners in Java

To scan for a range of rows, you create a **`Scan`** object with the start and (optionally) stop row key boundaries, and pass it to the **`getScanner()`** method on the **`Table`** class. That creates the scanner on the server and returns a **`ResultScanner`** object you can use to iterate the rows.

Each iteration returns a **`Result`** object, as in Code Listing 27, where I use a helper method **`printCells()`** to write the output:

Code Listing 27: Scanning Rows with Java

```
Table access_logs = connection.getTable(TableName.valueOf("access-logs"));
 Scan scan = new Scan(Bytes.toBytes("elton|jericho|201510"),
                      Bytes.toBytes("elton|jericho|x"));
 ResultScanner scanner = access_logs.getScanner(scan);
 for (Result result : scanner) {
     printCells(result);
 }

 //output - three cells, two whole rows:
 //[elton|jericho|201510] t:2908 = 80
 //[elton|jericho|201511] t:1106 = 120
 //[elton|jericho|201511] t:1107 = 650
```

You can tune the scanner performance by specifying properties on the **`Scan`** object:

- **`setCaching`** – Specify how many rows to cache on the server. Larger cache values mean the client can iterate the scanner more quickly, at the cost of server memory.
- **`setMaxResultSize`** – Specify the maximum number of cells the whole scanner should return. Useful to verify logic over a subset of data in a large table.
- **`setBatch`** – Specify the maximum number of cells to return for each iteration of the batch.

Note that the **`Scan`** instance is modified as the scanner is iterated, so you should create new instances for each scan you want to do. Other than the original **`Connection`** object, the client objects are cheap to create and don't need to be reused.

You can also restrict the cells in the results from the scanner with the **`addFamily()`** and **`addColumn()`** methods, which work in the same way as for the **`Get`** class.

Scanners and Filters in Java

By adding a filter to a scanner, you can perform complex queries. All the selection happens server-side, but you need to remember that while the row scan is fast, the column filtering is slower, especially for wide tables with lots of columns.

Filters are strongly typed in the Java API, inheriting from the abstract **`FilterBase`** class. There are a number of filters of varying usefulness—the tree view in the Javadoc for the package **`org.apache.hadoop.hbase.filter`** is a good place to examine them.

The **`ValueFilter`** is a useful example; it filters cells by comparing their value to a provided comparison operator, and another filter. If you store cell values as strings, you can filter the response to column values that match a regular expression, as in Code Listing 28:

Code Listing 28: Scanning and Filtering Rows with Java

```
scan = new Scan(Bytes.toBytes("elton|jericho|201510"),
                Bytes.toBytes("elton|jericho|x"));
scan.setFilter(new ValueFilter(CompareOp.EQUAL,
                new RegexStringComparator("[5-9][0-9]0")));
scanner = access_logs.getScanner(scan);
for (Result result : scanner) {
    printCells(result);
}

//output - one cell:

//[elton|jericho|201511] t:1107 = 650
```

The combination of **`ValueFilter`** and **`RegexStringComparator`** means cells will only be included in the response if they have a three-digit value between 500 and 990, ending in zero. That filter works across all columns in all families; a family name or qualifier isn't required.

With the Java API, you can also combine many filters using a **`FilterList`** object, and specify the inclusion criteria, whether rows must match all of the filters or just one.

You can combine any filters in a list. Code Sample 29 shows a list that filters on column qualifier name and cell value, both using regular expressions:

Code Listing 29: Scanning with Multiple Filters in Java

```
 FilterList filterList = new
  FilterList(FilterList.Operator.MUST_PASS_ALL);
 filterList.addFilter(new QualifierFilter(CompareOp.EQUAL,
                       new RegexStringComparator("[0-9]{2}0[7-8]")));
 filterList.addFilter(new ValueFilter(CompareOp.EQUAL,
                       new RegexStringComparator("[0-9]0")));
 scan = new Scan(Bytes.toBytes("elton|jericho|201510"),
                 Bytes.toBytes("elton|jericho|x"));
 scan.setFilter(filterList);
 scanner = access_logs.getScanner(scan);
 for (Result result : scanner) {
     printCells(result);
 }

 //output - two cells:
 //[elton|jericho|201510] t:2908 = 80
 //[elton|jericho|201511] t:1107 = 650
```

Writing Data in Java

The Java API provides basic data updates, much like the **put** command in the HBase Shell, but also adds some more flexible functionality.

The **Put** class is the write equivalent of the **Get** class. You instantiate it for a specified row key, and then you can add one or more column values, before making the changes by calling the **put()** method on the relevant **Table** object, as in Code Listing 30:

Code Listing 30: Updating Cells with Put in Java

```
Table access_logs = connection.getTable(TableName.valueOf("access-logs"));
 Put log = new Put(Bytes.toBytes("elton|jericho|201511"));
 log.addColumn(Bytes.toBytes("t"),      //family
               Bytes.toBytes("1621"),   //qualifier
               Bytes.toBytes("340"));   //value
 access_logs.put(log);

 //result - updated cell value:

 //t:1621 = 120
```

You can add multiple cell values to a **Put** object, which will automatically set multiple values on a single row, and an overload of the **addColumn()** method allows you to specify a timestamp for the cell.

The **Put** object is also used in the **Table** method **checkAndPut()**, which makes a conditional update to a cell. The method takes a column name and cell value to check before making the update. If the provided value matches, then the put is automatically made; if not, the row is not changed.

Code Listing 31 shows how **checkAndPut()** is used to add a new cell to a row, but only if an existing cell (in that row, or in another row of the table) has the expected value. In this case, I'm telling HBase to add a column **t:1622**, but only if the value of **t:1621** is **34000**, which it isn't, so the update should not be made:

Code Listing 31: Updating Conditionally with Java

```
Put newLog = new Put(Bytes.toBytes("elton|jericho|201511"));
 log.addColumn(Bytes.toBytes("t"),
               Bytes.toBytes("1622"),
               Bytes.toBytes("100"));
 access_logs.checkAndPut(Bytes.toBytes("elton|jericho|201511"),
                         Bytes.toBytes("t"), //family
                         Bytes.toBytes("1621"),
                         Bytes.toBytes("34000"),
                         newLog);


 //result - not updated, checked value doesn't match
```

Code Listing 32 shows the result of running the two **put** methods, from the HBase Shell. Cell **t:1621** has the value **340**, so the new cell **t:1622** hasn't been added:

Code Listing 32: Fetching Cells Updated with Java

```
hbase(main):002:0> get 'access-logs', 'elton|jericho|201511'
COLUMN                CELL
 t:1106               timestamp=1447703111745, value=120
 t:1107               timestamp=1447703111735, value=650
 t:1621               timestamp=1447709413579, value=340

3 row(s) in 0.0730 seconds
```

The Java API also lets you make multiple updates to different rows in a single batch. The same **Put** class is used to define the changes, and multiple **Put** objects are added to a list. The list is used with the **batch()** method on the **Table** class, which writes the updates in a single server call, as shown in Code Listing 33:

Code Listing 33: Batch Updating Cells with Java

```
List<Row> batch = new ArrayList<Row>();

Put put1 = new Put(Bytes.toBytes("elton|jericho|201512"));
put1.addColumn(Bytes.toBytes("t"),
               Bytes.toBytes("0109"),
               Bytes.toBytes("670"));
batch.add(put1);

Put put2 = new Put(Bytes.toBytes("elton|jericho|201601"));
put2.addColumn(Bytes.toBytes("t"),
               Bytes.toBytes("0110"),
               Bytes.toBytes("110"));
batch.add(put2);

Put put3 = new Put(Bytes.toBytes("elton|jericho|201602"));
put3.addColumn(Bytes.toBytes("t"),
               Bytes.toBytes("0206"),
               Bytes.toBytes("500"));
batch.add(put3);

Table access_logs = connection.getTable(TableName.valueOf("access-logs"));
Object[] results = new Object[batch.size()];

access_logs.batch(batch, results);
```

You can include other operations in a batch, so you could add **`Delete`** objects with **`Put`** objects. The batch can include **`Get`** objects to return a set of results, but the ordering of the batch is not guaranteed—so if a **`Get`** contains the same cells as a **`Put`**, you may get the data in the state it was before the **`Put`**.

In Code Listing 34, there's the result of executing that batch seen in a **`scan`** command from the HBase Shell:

Code Listing 34: Fetching Cells from a Java Batch Update

```
hbase(main):003:0> scan 'access-logs', {STARTROW => 'elton|jericho|201512'}
ROW                    COLUMN+CELL
 elton|jericho|201512 column=t:0109, timestamp=1447710255527, value=670
 elton|jericho|201601 column=t:0110, timestamp=1447710255527, value=110
 elton|jericho|201602 column=t:0206, timestamp=1447710255527, value=500

3 row(s) in 0.0680 seconds
```

Tip: This is the first mention of deleting data in HBase. I haven't covered it because I find you do it rarely, but you can delete individual cells and rows in tables. HBase

uses a delete marker to flag deleted values, rather than immediately removing the data from disk, so the delete operation is fast.

Summary

The Java API is the richest client interface for HBase, and there are many more features than I've covered here, including incrementing counter columns, functions for accessing random data items (useful for integration testing), and a set of admin operations.

Java access is direct to the region servers (or master server for admin functions), and it's the most efficient API. It doesn't require any additional JVMs to run on the region servers, and the client is region-aware, so it requests data directly from the server that is hosting the region.

Java is also the native language for extending HBase on the server. Co-processors are an advanced topic that I won't cover in this book, but they are valuable in many scenarios. You write a co-processor in Java and make the package available to the Region Servers on HDFS. Then your code can be invoked on the server in response to data operations, like when rows are added or cell values are changed, similar to triggers in SQL databases.

Even if you work exclusively in Java, it's good to know what the external HBase APIs offer and how they work. In the next chapter, we'll cover the Thrift API, which has near-parity with the Java feature set, and can be consumed from many client libraries. We'll cover using Thrift with Python.

Chapter 5 Connecting with Python and Thrift

Overview

HBase is well suited for cross-platform solutions, and the Thrift API is an alternative to the Java API. Thrift is Apache's generic API interface, which supports client connections from different languages to Java servers. We'll use Python in this chapter, but you can use any language with Thrift binding (including Go, C#, Haskell, and Node).

The Thrift API is an external interface, so it requires an additional JVM to run. You can start it with the HBase daemon script **`hbase-daemon.sh start thrift`**. It can be hosted separately to the rest of the HBase cluster, or it can be run on the Region Servers. Thrift doesn't have a native load-balancer, but the transport is TCP, so you can use an external load balancer (like HAProxy).

By default, the Thrift server listens on port 9090, and it's already running on the **`hbase-succinctly`** Docker image.

Thrift is more lightweight than REST, so it can offer better performance, but it's not so user-friendly. To consume a Thrift API, in most cases you need to build Thrift from source, generate a binding to the API from the public **`.thrift`** file that describes the interface, and then import the Thrift transport and the bindings for your client app.

Note: The Thrift API is documented in the .thrift file. That file doesn't ship with the HBase binaries, so you'll need to fetch the correct version from source. For release 1.1.2, the file is on GitHub here.

Working with Thrift Clients

The generated code from the **`.thrift`** definition file contains classes for working with the Thrift server at a relatively low level. Figure 4 shows the sequence for connecting to a server:

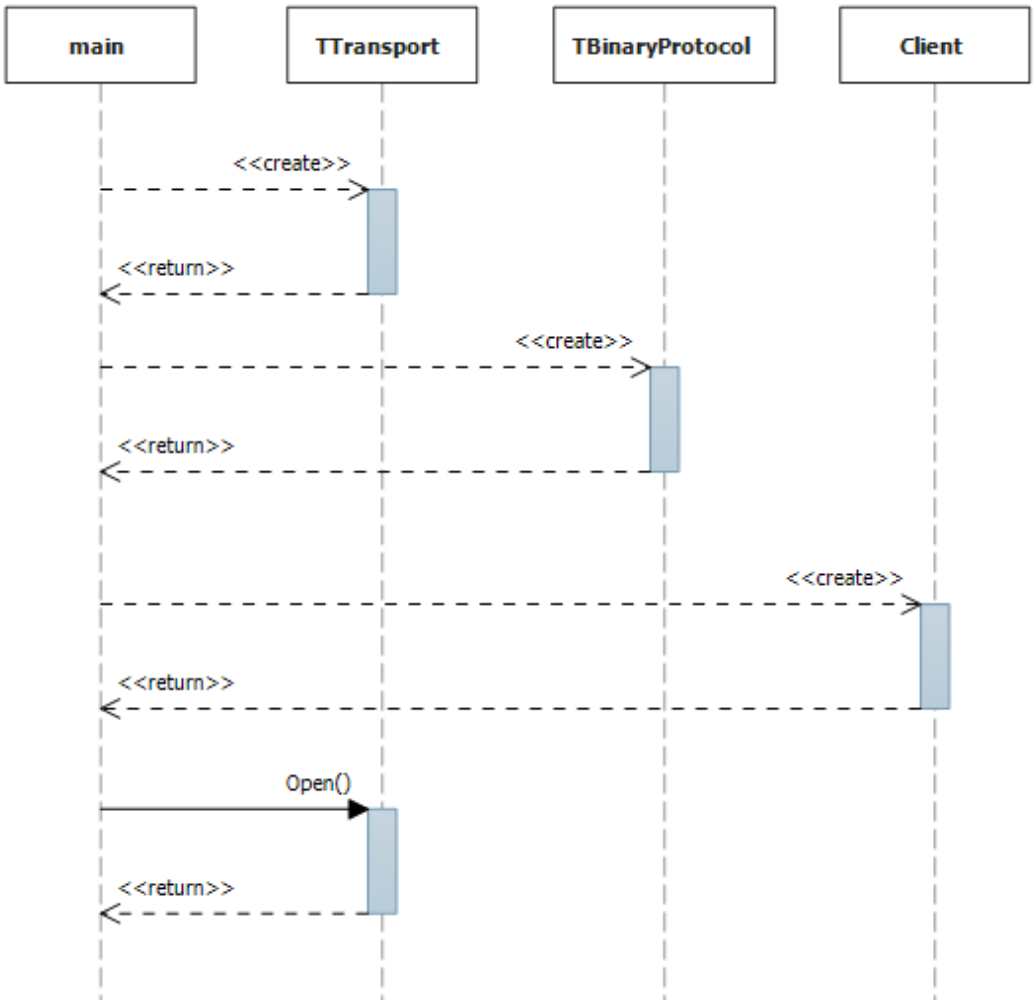


Figure 4: Connecting to HBase with Thrift

First, you need to create an instance of **`TTransport`**, and then an instance of **`TBinaryProtocol`** using the transport. Then you can create an HBase client using the protocol and open the transport to start the connection. Thrift provides a lot of functionality and benefits from wide platform reach, but in use the client can be cumbersome, and it takes a lot of code to do straightforward tasks.

It's worth looking for a community wrapper for Thrift in the language you want to use. In this chapter I'll use the HappyBase library for Python, which is a wrapper for Thrift that takes care of binding generation and imports, and also exposes a much friendlier client API than the raw Thrift version.

Connecting to Thrift with HappyBase

You need to install HappyBase in your environment. It's publically available on the Python Package Index, so assuming you already have Python and Pip (the Python package manager), you can install it with the command in Code Listing 35:

Code Listing 35: Installing the HappyBase Python package

```
$ pip install happybase
```

Now you can start Python and set up all the dependencies by importing the HappyBase package with **`import happybase`**. HappyBase is designed to expose HBase features in a Python-like way, so in Code Listing 36, we create a **`connection`** object that will automatically connect to the HBase Thrift server running locally:

Code Listing 36: Connecting to HBase with HappyBase

```
>>> connection = happybase.Connection('127.0.0.1')
```

The **`Connection`** object is the starting point for Thrift connections. From the **`Connection`** object, you can access **`Table`** objects, which are used for DDL and DML statements, and get **`Batch`** objects, which are used for batched data updates.

Reading Data with Python

HappyBase makes HBase interaction very simple. Use the **`table()`** method on a connection to get a **`Table`** object, and you can use the **`row`** method on the table to read individual cells, column families, or a whole row. Results are returned in dictionaries; Code Listing 37 shows the values returned for a row, column family, and cell:

Code Listing 37: Reading a Row with HappyBase

```
>>> table = connection.table('access-logs')
>>> print table.row('elton|jericho|201511')
{'t:1106': '120', 't:1107': '650'}
>>> print table.row('elton|jericho|201511', ['t'])
{'t:1106': '120', 't:1107': '650'}
>>> print table.row('elton|jericho|201511', ['t:1106'])
{'t:1106': '120'}
```

You can also read multiple rows by providing a list of keys to the **`rows()`** method, which returns a list containing a tuple for each row. The tuple contains the row key and a dictionary of column values, as in Code Listing 38, where two rows are returned:

Code Listing 38: Reading Multiple Rows with HappyBase

```
>>> print table.rows(['elton|jericho|201511', 'elton|jericho|201510'])

[('elton|jericho|201511', {'t:1106': '120', 't:1107': '650'}),
('elton|jericho|201510', {'t:2908': '80'})]
```

The key list is an explicit set of keys, not the start and end points for a range (for that you need a scanner, which we'll use in the next section). If a key you request doesn't exist, then it isn't returned in the response. If none of the keys you request exist, then you get an empty list back.

The **rows()** method also allows filtering by column family or column; if you request columns which don't exist for a row in the row key list, that row isn't returned in the response. In Code Listing 39, the request is for the **t:1106** column from two rows, but only one row has that column, so the other row isn't returned:

Code Listing 39: Filtering Columns from Multiple Rows with HappyBase

```
>>> print table.rows(['elton|jericho|201511', 'elton|jericho|201510'],
['t:1106'])

[('elton|jericho|201511', {'t:1106': '120'})]
```

The **row()** and **rows()** methods can include an option to return the timestamp for each cell in the response, but if you have a table with multiple versions in a column family, these methods only return the most recent version.

To read more than one version from a column, HappyBase has the **cells()** method, which takes a row key and column name, together with the number of versions to return (and optionally the timestamp of the data), as shown in Code Listing 40:

Code Listing 40: Reading Multiple Cell Versions with HappyBase

```
>>> versionedTable = connection.table('with-custom-config')
>>> print versionedTable.cells('rk1', 'cf1:data', 3)
['v2', 'v1', 'v0']

>>> print versionedTable.cells('rk1', 'cf1:data', 3,
include_timestamp=True)
[('v2', 1447399969699), ('v1', 1447399962115), ('v0', 1447399948404)]
```

The **cells()** method returns cell versions in descending order of timestamp.

For rows with counter columns, the data will be returned in the **row()** and **cells()** methods, but in an unfriendly hex format. HappyBase also includes a **counter_get** method to read the current value of a counter column as a long integer.

Code Listing 41 shows the different results for reading a counter column:

Code Listing 41: Reading Counter Columns with HappyBase

```
>>> counterTable = connection.table('counters')
>>> print counterTable.row('rk1')
{'c:1': '\x00\x00\x00\x00\x00\x00\x00\x01'}

>>> print counterTable.counter_get('rk1', 'c:1')
1
```

Working with Scanners in Python

The **`Table`** object has a **`scan()`** method to create a scanner on the region server, which you can iterate through on the client. You can use **`scan`** in the same way as the HBase Shell, passing start and stop rows to define the boundaries, as in Code Listing 42:

Code Listing 42: Scanning Rows with HappyBase

```
>>> access_logs = connection.table('access-logs')
>>> scanner = access_logs.scan('elton|jericho|201510',
'elton|jericho|x')
>>> for key, data in scanner:
...     print key, data
...
elton|jericho|201510 {'t:2908': '80'}
elton|jericho|201511 {'t:1106': '120', 't:1107': '650'}
```

There are some friendly additions to the **`scan()`** method. You can pass a row key prefix instead of start and stop rows, and HappyBase sets the boundaries for you; you can also pass a list of column family names or column names to restrict the data in the response, as in Code Listing 43:

Code Listing 43: Scanning Rows by Prefix with HappyBase

```
>>> scanner = access_logs.scan(row_prefix='elton|jericho|',
columns=['t:1106'])
>>> for key, data in scanner:
...     print key, data
...

elton|jericho|201511 {'t:1106': '120'}
```

Scan returns an iterable object, which you can loop through as a single result set, although HappyBase will actually read the result in batches from Thrift. You can specify a **batch_size** argument to tune the reads from the scanner; this defaults to 1,000, which is a reasonable assumption to favor large batches over multiple reads.

If you are working with a wide table or with large cell sizes, then you may need to reduce the batch size to improve overall performance. If you are reading small cell values from many rows, a larger batch size may be better.

Scanners and Filters in Python

Thrift supports scanners with filters that run on the Region Server. The scanner reads rows efficiently from the supplied row key boundaries, and the filter extracts just the rows or columns you want returned.

HappyBase lets you create filtered scanners from the client in the **scan()** method. This is one area where HappyBase doesn't abstract the complexity, and you have to construct the filter as a string, in line with the Thrift API filter language.

The general format of the filter string is **{filter name} ({arguments})**. Code Listing 44 shows a filter that returns only the rows in the scan that have values in columns starting with the prefix "11":

Code Listing 44: Scanning and Filtering Rows with HappyBase

```
>>> access_logs = connection.table('access-logs')
>>> scanner = access_logs.scan('elton|jericho|201510', 'elton|jericho|x',
filter="ColumnPrefixFilter('11')")
>>> for key, data in scanner:
...     print key, data
...

elton|jericho|201511 {'t:1106': '120', 't:1107': '650'}
```

Tip: The Thrift API is well documented in the HBase online documentation, and the available filters and the arguments you need to supply are covered here.

Writing Data from Python

The **put()** method on the HappyBase table object works a lot like the **put** command in the HBase Shell, taking the row key, column name, and value. With HappyBase though, you can update and insert multiple cell values with a single statement, passing a dictionary of *key:value* pairs as in Code Listing 45:

Code Listing 45: Updating Data with HappyBase

```
>>> access_logs.put('elton|jericho|201511', {'t:1309':'400',
't:1310':'200'})
>>> print access_logs.row('elton|jericho|201511', ['t:1309', 't:1310'])
{'t:1310': '200', 't:1309': '400'}
```

The **put()** method is limited to a single row, but HappyBase provides a useful mechanism for batching updates. This is a common requirement in HBase clients, particularly in event-streaming applications where you may receive hundreds or even thousands of events per second that you want to buffer in your processor.

The **Batch** class in HappyBase allows you to do that without writing custom code to maintain the buffer of pending updates. You can create a batch object from a table and use it within a context manager block. When the block ends, the **send()** method is called on the batch, which sends all the updates to the Thrift server, as shown in Code Listing 46:

Code Listing 46: Batch Updating Data with HappyBase

```
>>> with access_logs.batch() as batch:
...     batch.put('elton|jericho|201512', {'t:0110':'200'})
...     batch.put('elton|jericho|201512', {'t:0210':'120', 't:0211':'360'})
...
>>> print access_logs.row('elton|jericho|201512')
{'t:0211': '360', 't:0210': '120', 't:0110': '200'}
```

The **put** method on the **Batch** class has the same signature as on the **Table** class, so you can make one or many updates to a row with each **put**.

Note: Batching is supported by the Thrift API, so when a batch of updates (called mutations in the native Thrift API) are sent, that's done in a single call to the Thrift connection.

Thrift also supports incrementing counter columns, which you can do in HappyBase with the **counter_inc** method, optionally providing an amount to increment by, as shown in Code Listing 47:

:

Code Listing 47: Incrementing Counters with HappyBase

```
>>> counterTable.counter_get('rk1', 'c:1')
1
>>> counterTable.counter_inc('rk1', 'c:1')
2
>>> counterTable.counter_inc('rk1', 'c:1', 100)
102
```

Note that the **`counter_inc`** method returns the cell value after the increment is applied, unlike the **`put()`** method, which has no return.

Summary

In this chapter, we looked at the principal external API for HBase, the Thrift server. Thrift can run on the Region Servers and provides a fully-featured alternative to the Java API. You generate a Thrift client in your preferred language, and then you work with the classes natively.

The Thrift client API operates at a low level, and in many languages there are community wrappers to make the generic Thrift classes easier to work with, and to give them a more native feel in the platform. We used the HappyBase library, which is a wrapper for the Python client.

Thrift provides all the key features you need to read, scan, and write data—all neatly wrapped in HappyBase to give an intuitive interface. You can fetch rows by key, selecting the columns to return, and you can scan for a set of rows with a more complex filter applied at the server.

You can write cell values individually or in bulk, and for many-row updates, Thrift supports a batch interface that allows multiple updates from a single server call. Thrift also supports counter columns, so you can read and increment counters with HappyBase.

The biggest downside with Thrift is getting started with the Thrift client and generating the language bindings for HBase. If there's no friendly wrapper in your preferred language, then the amount of setup work can be quite involved.

In the next chapter, we'll look at the other external API for HBase—the REST API, Stargate. That provides access to HBase data over HTTP, so it offers an even simpler cross-platform experience, although without all the features provided by Thrift.

Chapter 6 Connecting with .NET and Stargate

Overview

Stargate is the name of the HBase REST API, which makes data available to read and write over HTTP. Stargate exposes data from URLs that match the table structure (e.g., **`/access-logs/rk1`** would get the row with key rk1 from the **`access-logs`** table).

The HTTP verbs **`GET`**, **`POST`** and **`DELETE`** are used to work with data as resources, which gives Stargate a nicely RESTful interface. You can work with rows and cells in JSON, but the downside with the API is that all data is represented as Base64 strings, which are encoded from the raw byte arrays in HBase. That makes the API awkward if you just want to browse with a REST client like Postman or cURL.

Like the Thrift API, Stargate is a separate service, which you can start with **`hbase-daemon.sh start rest`**. By default, it listens on port 8080 (and is already running on the **`hbase-succinctly`** Docker image).

In production you can run Stargate on the Region Servers, but if you want to use it as your primary interface for HBase, you should consider load balancing with a separate server.

Tip: I walk through creating a load balancing reverse-proxy using Nginx to front Stargate in this blog post.

You can use any framework with an HTTP client to talk to Stargate. In this chapter we'll use cURL to see the raw HTTP data, and a .NET client library to work at a higher level of abstraction.

Reading Data with cURL

Code Listing 48 shows a **`GET`** request (the default verb in cURL) to the root Stargate URL. The response is the same as a list command in the HBase Shell:

Code Listing 48: Listing tables with cURL

```
$ curl http://127.0.0.1:8080
access-logs

social-usage
```

An HTTP **GET** request is equivalent to a **get** command in the HBase shell. You can add an **Accept** header to specify the format you want in the response, but there are some restrictions to the amount of data Stargate will serve.

If you try to request a whole table, e.g. **http://127.0.0.1:8080/access-logs**, you'll get an error response, with status code 405, meaning the method isn't allowed. You can't fetch a whole table in HBase, and the 405 is Stargate's implementation of that.

You can fetch a whole row by adding the row key after the table name, but if you have any HTTP-unfriendly characters in your row format, you'll need to escape them in the URL. You also can't get a plain-text representation of a row; you need to make the request with a data-focused format like JSON.

Code Listing 49 shows a whole row read from Stargate. Note that the pipe character in the row key has been escaped as **%7C**, and the data values in the response—row key, column qualifiers and cell values—are all Base64-encoded strings (I've applied formatting to the response; there's no whitespace returned by Stargate).

Code Listing 49: Reading a row with cURL

```
$ curl -H accept:application/json http://127.0.0.1:8080/access-
logs/elton%7Cjericho%7C201511

{
    "Row": [{
        "key": "ZWx0b258amVyaWNob3wyMDE1MTE=",
        "Cell": [{
            "column": "dDoxMTA2",
            "timestamp": 1447228701460,
            "$": "MTIw"
        }, {
            "column": "dDoxMTA3",
            "timestamp": 1447228695240,
            "$": "NjUw"
        }]
    }]

}
```

The response is a JSON object that contains an array of **Row** objects. Each row has a **key** field and an array of **Cell** objects. Each cell has a column qualifier, a cell value (stored in the **$** field), and a timestamp. Timestamps are the only values HBase stores with an interpretation; they are long integers, storing the UNIX timestamp when the row was updated.

Other values are Base64 strings, which means you need to decode the fields in a **GET** response. Table 6 shows the decoded values for one column in the response:

Field	Value	Decode value
Row.key	ZWx0b258amVyaWNob3wyMDE1MTE=	elton|jericho|201511
Cell.column	dDoxMTA2	t:1106
Cell.$	MTIw	120

Table 6: Encoded and Decoded Stargate Values

Note that the column value in the cell is the full name (column family plus qualifier, separated by a colon), and the numeric cell value is actually a string.

You can also fetch a single-column family in a row from Stargate (with the URL format **`/{table}/{row-key}/{column-family}`**, or a single cell value. For single cells, you can fetch them in plain text and Stargate will decode the value in the response, as in **Error! Reference ource not found.**:

Code Listing 50: Fetching a single cell value

```
$ curl http://127.0.0.1:8080/access-logs/elton%7Cjericho%7C201511/t:1106
120
```

Updating Data with cURL

The semantics of a **`PUT`** request with Stargate are much the same as a **`put`** command in the HBase Shell. You specify the desired end state in your request, and HBase does the rest—creating the row or column if it doesn't exist, and setting the value.

You have to use a data format to make updates through Stargate; otherwise you'll get a 415 error, “Unsupported Media Type.” Using JSON, you mirror the format from a **`GET`** response, so you can send multiple cell values for multiple rows in a single request.

The URL format still requires a row key as well as a table, but with a **`PUT`** the key in the URL gets ignored in favor of the key(s) in the request data. Code Listing 51 shows a **`PUT`** request that updates the two cells in my row:

Code Listing 51: Updating cells with Stargate

```
$ curl -X PUT -H "Content-Type: application/json" -d '{
    "Row": [{
        "key": "ZWx0b258amVyaWNob3wyMDE1MTE=",
        "Cell": [{
            "column": "dDoxMTA2",
            "timestamp": 1447228701460,
            "$": "MTMw"
        }, {
            "column": "dDoxMTA3",
            "timestamp": 1447228695240,
            "$": "NjYw"
        }]
    }]

}' 'http://127.0.0.1:8080/access-logs/FAKE-KEY'
```

Tip: Stargate is very handy for ad-hoc requests, but working with Base64 can be difficult. I've blogged about making it easier with a couple of simple tools here.

That **PUT** request increments the numeric values that are actually strings in my row. The **incr** command to atomically increment counter columns isn't available through Stargate, so if you need to increment values, then you have to read them first and then **PUT** the update.

You can do more with cURL, like sending **DELETE** requests to remove data, and creating scanners to fetch multiple rows, but the syntax gets cumbersome. Using a wrapper around the RESTful API in Stargate is a better option.

Using a Stargate NuGet Package with .NET

NuGet is the Package Manager for .NET apps, and there are a couple of open-source packages that are wrappers for accessing Stargate. Microsoft has a package specifically for HBase clusters running on the Azure cloud, and there's a third-party package from authors “The Tribe” for working with Stargate generically.

The package does a good job of abstracting the internals of Stargate and lets you work with HBase data intuitively. It's IoC-aware (using Autofac by default), so you can easily tweak the HTTP setup and build a data-access layer, which you can mock out for testing.

To add that package and its dependencies to your .NET app, you can use the NuGet Package Manager Console command in Code Listing 52:

Code Listing 52: Adding a NuGet Reference for Stargate

```
Install-Package "HBase.Stargate.Client.Autofac"
```

In the GitHub repository for this book, there's a .NET console app that uses The Tribe's client to connect to Stargate, running in the **`hbase-succinctly`** Docker container.

Connecting to Stargate

To setup the Stargate client, you need to configure the server URL and build the container. Code Listing 53 shows how to do that with Autofac:

Code Listing 53: Configuring the Stargate client

```
var builder = new ContainerBuilder();
builder.RegisterModule(new StargateModule(new StargateOptions
{
      ServerUrl = "http://127.0.0.1:8080"
}));

var container = builder.Build();

var stargate = container.Resolve<IStargate>();
```

The **`StargateOptions`** object contains the Stargate (or proxy) URL, and the **`StargateModule`** contains all the other container registrations. The **`IStargate`** interface you get from the container provides access to all the Stargate client operations, with a neat abstraction and with all data items encoded as strings.

Reading Data with .NET

The Stargate client has two methods for reading data. The simplest is **`ReadValue()`**, which fetches a specific cell value, passing it the table name, row key, column family, and qualifier. This is functionally equivalent to a **`GET`** request with a URL containing the table name, row key, and column name, which returns a single cell value encoded as a string, as in Code Listing 54:

Code Listing 54: Reading a cell with IStargate

```
var value = stargate.ReadValue("access-logs", "elton|jericho|201511", "t",
"1106");
```

```
//value is "120"
```

Alternatively, you can fetch a **CellSet** object using the **FindCells()** method, which returns a collection of rows and cells. This is like issuing a **GET** request for a full row, or a column family in a row. The **CellSet** is an enumerable collection, which you can query with LINQ as in Code Listing 55:

Code Listing 55: Finding cells with IStargate

```
var cellSet = stargate.FindCells("access-logs", "elton|jericho|201511");
var value = cellSet
             .First(x => x.Identifier.CellDescriptor.Qualifier ==
"1106").Value;

//value is "120"
```

Note that the **FindCells()** call makes the **GET** request to Stargate, which returns all the data in the **CellSet**, and the LINQ query runs over the **CellSet** in memory on the .NET client.

The **ReadValue()** call will return quickly because it's fetching a single piece of data, and the **FindCells()** call will be quick for Stargate to serve, but could take longer for the client to receive if there are many cells that contain a lot of data.

An alternative way to fetch data from Stargate is to create a row scanner, which is like server-side cursor that you can use to read multiple rows, and the scanner can optionally have a filter to limit the number of cells that get returned.

Working with Scanners in .NET

There are two parts to scanning rows with Stargate. Firstly, you create the scanner, which runs on the server. Stargate gives you a reference to the scanner, which you can use to fetch rows. In the .NET client, you use a **ScannerOptions** object to specify the start and end rows for the scan.

Code Listing 56 shows how to create a scanner to retrieve all the access logs for one user for one system from October 2015 onwards:

Code Listing 56: Creating a Scanner with IStargate

```
var options = new ScannerOptions
{
      TableName = "access-logs",
      StartRow = "elton|jericho|201510",
      StopRow = "elton|jericho|x",
};
```

```
var scanner = stargate.CreateScanner(options);
```

When Stargate returns, the scanner is running on the server and you can loop through rows from the start up to the end row key, using the **`MoveNext()`** method to fetch the next set of cells from Stargate, as in Code Listing 57:

Code Listing 57: Iterating Through a Scanner with IStargate

```
var totalUsage = 0;
while (scanner.MoveNext())
{
      var cells = scanner.Current;
      foreach (var cell in cells)
      {
            totalUsage += int.Parse(cell.Value);
      }
}

//totalUsage is 850
```

Note: The cells returned when you iterate through a scanner could be from multiple rows, so don't assume MoveNext (or the equivalent in other clients) moves on to the next row—it moves on to the next set of cells, which could be from many rows.

Scanners and Filters in .NET

Scanning by row key is the fastest way to read data from Stargate, but if you need to additionally restrict the results by the data in the columns (the qualifiers or the cell values), you can specify a filter when you create a scanner.

The filter also runs server-side in Stargate, so it's faster than fetching whole rows and extracting certain columns in the client, but it does use additional server resources (important if you're running Stargate on the Region Servers).

There are various filter types that are well documented for the Stargate API, but some of the most useful are:

- **`KeyOnlyFilter`** – returns only the row keys in the scanner
- **`QualifierFilter`** – returns cells that match the specified column qualifier
- **`ColumnPrefixFilter`** – returns cells where column names match the specified prefix

*In the **`access-logs`** table, the period in the row key specifies the year and month of the usage record, and the column qualifier contains the day and hour. In*

Code Listing 58, I add a column prefix filter to the scanner, which filters the results so only cells with column names that start with the provided prefix are returned. In this case, only cells from the 11th day of each month will be included in the results:

Code Listing 58: Creating a Filtered Scanner with IStargate

```
var options = new ScannerOptions
{
      TableName = "access-logs",
      StartRow = "elton|jericho|201510",
      StopRow = "elton|jericho|x",
      Filter = new ColumnPrefixFilter("11")
};

var scanner = stargate.CreateScanner(options);
```

Writing Data from .NET

The **IStargate** interface has two methods for writing data. The simplest is **WriteValue()**, which is the equivalent of a **PUT** request for a specific cell in a row. Stargate creates the row and/or column if needed, and sets the value, as in Code Listing 59:

Code Listing 59: Updating a Cell Value with IStargate

```
stargate.WriteValue("100", "access-logs", "elton|jericho|201510", "t",
"2908");

//cell value is now "100"
```

A more complex and flexible method is **WriteCells()**, which takes a **CellSet** object and can update multiple values with a single API call. That mix of values can include updates and inserts for different rows, but all the rows must be in the same table.

Code Listing 60 shows an update to an existing cell value, and an insertion of a new row in a single call to **WriteCells()**:

Code Listing 60: Updating a Cell Value with IStargate

```
var update = new Cell(new Identifier
{
      Row = "elton|jericho|201510",
      CellDescriptor = new HBaseCellDescriptor
      {
            Column = "t",
            Qualifier = "2908"
      }
}, "120");
```

```
var insert = new Cell(new Identifier
{
      Row = "elijah|jericho|201511",
      CellDescriptor = new HBaseCellDescriptor
      {
            Column = "t",
            Qualifier = "1117"
      }
}, "360");

var cells = new CellSet(new Cell[] { update, insert});
cells.Table = "access-logs";


stargate.WriteCells(cells);
```

The Stargate API is stateless (even scanners run on the Region Server, which is not necessarily the Stargate server), and so is the client, so there is no caching of data locally, unless you retain cells in memory yourself. Every call to the Stargate client to read or write data results in a REST call to Stargate.

Summary

In this chapter we looked at Stargate, the REST API that HBase provides. It exposes data from rows as resources, with a URL format that describes the path to the data, always including the table name and row key, and optionally a column family and qualifier.

The API supports different data formats, including JSON and Google's Protocol Buffers, and for simple reading and writing, you can use plain text (although the feature set is more limited). Stargate passes the cURL test—if you can use an API with cURL, then it has a usable RESTful design.

As Stargate provides a standardized approach to accessing data, it can be easily wrapped into a client library, and we covered one option with a .NET NuGet package. Stargate provides much of the functionality of other clients (including DDL functions that we haven't had room for, like creating tables), but one feature not supported at the time of writing is incrementing counter columns.

Now we have a good understanding of what you can store in HBase and the client options for accessing it. In the next chapter, we'll step back and look at the architecture of HBase.

Chapter 7 The Architecture of HBase

Component Parts

In a distributed deployment of HBase, there are four components of the architecture, which together constitute the HBase Server:

- Region Servers – compute nodes that host regions and provide access to data
- Master Server – coordinates the Region Servers and runs background jobs
- Zookeeper – contains shared configuration and notifies the Master about server failure
- Hadoop Distributed Files System (HDFS) – the storage layer, physically hosted among the Master and Region Servers

Figure 5 shows the relationship between the components in a distributed installation of HBase, where the components are running on multiple servers:

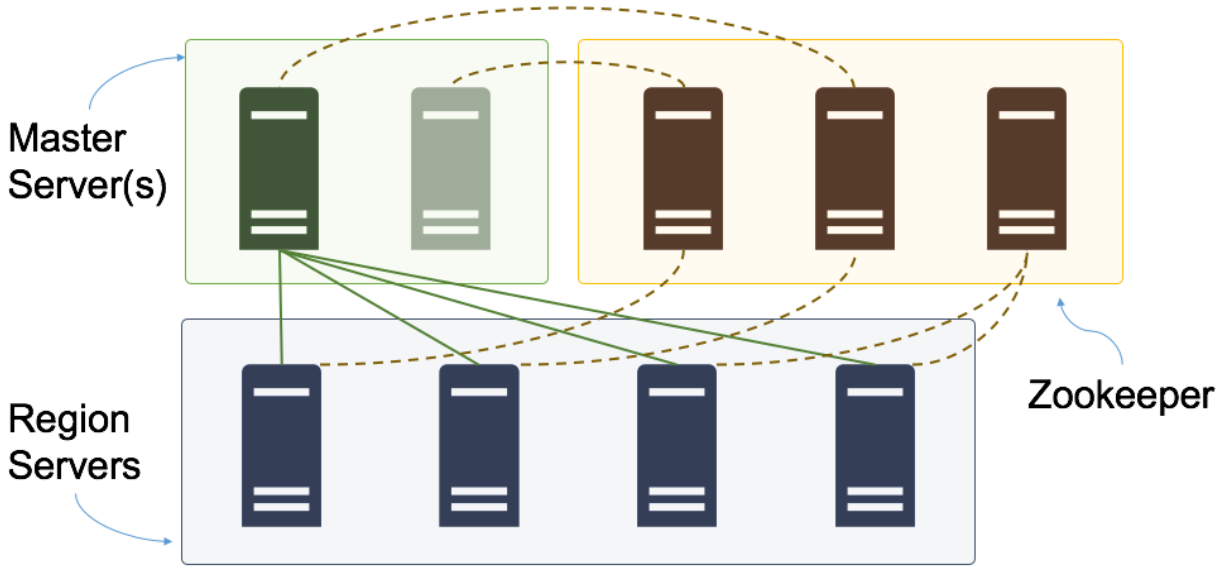


Figure 5: A Distributed HBase Cluster

Hosting each component on multiple servers provides resilience and performance in a production environment. As a preferred minimum, clusters should have two Master Servers, three Zookeeper nodes, and four Region Servers.

Note: Every node in an HBase cluster should have a dedicated role. Don't try to run Zookeeper on the Region Servers—the performance and reliability of the whole

cluster will degrade. Each node should be on a separate physical or virtual machine, all within the same physical or virtual network.

Running HBase in production doesn't necessarily require you to commission nine servers to run on-premise. You can spin up a managed HBase cluster in the cloud, and only pay for compute power when you need it.

Master Server

The Master Server (also called the "HMaster") is the administrator of the system. It owns the metadata, so it's the place where table changes are made, and it also manages the Region Servers.

When you make any administration changes in HBase, like creating or altering tables, it is done through the Master Server. The HBase Shell uses functionality from the Master, which is why you can't run the Shell remotely (although you can invoke HMaster functions from other remote interfaces).

The Master Server listens for notifications from Zookeeper for a change in the connected state of any Region Servers. If a Region Server goes down, the Master re-allocates all the regions it was serving to other Region Servers, as shown in Figure 6:

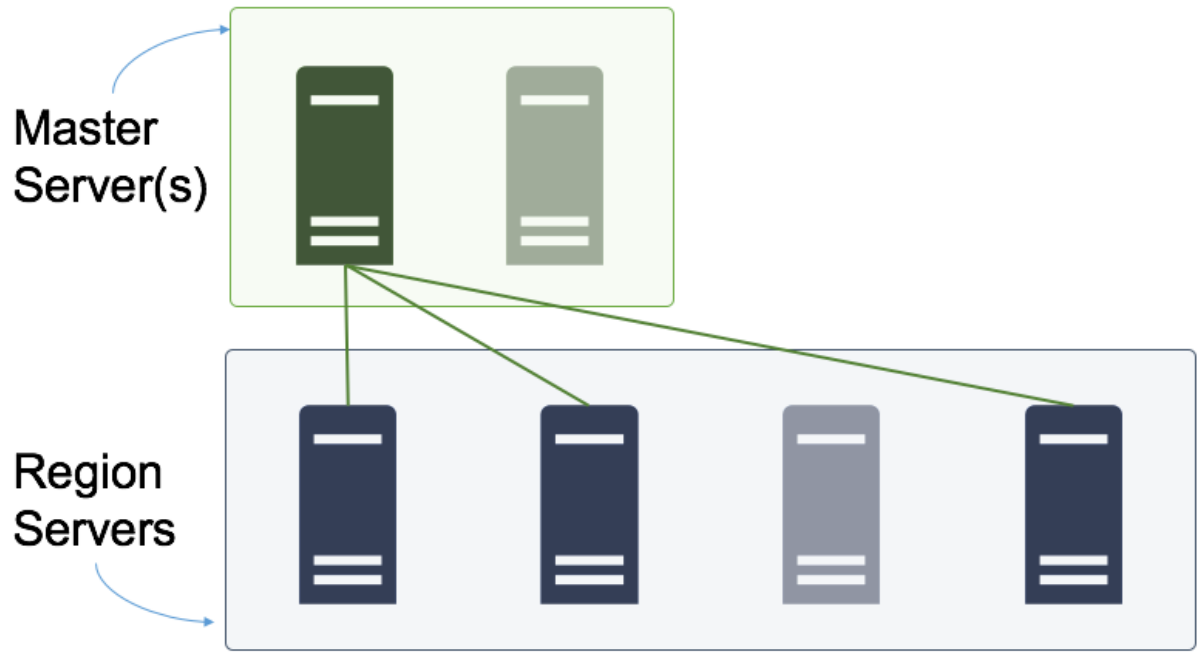


Figure 6: Re-allocating Regions

When a Region Server comes online (either a lost server rejoins, or a new server in the cluster), it won't serve any regions until they are allocated to it. You can allocate regions between servers manually, but the Master periodically runs a load-balancer job, which distributes regions as evenly as it can among the active Region Servers.

There is only one active Master Server in a cluster. If there are additional servers for reliability, they run in active-passive mode. The passive server(s) listen for notifications from Zookeeper on the active Master's state, and if it goes down, another server takes over as the new Master.

You don't need several Master Servers in a cluster. Typically, two servers are adequate.

Client connections are not dependent on having a working Master Server. If all Master Servers are down, clients can still connect to Region Servers and access data, but the system will be in an insecure state and won't be able to react to any failure of the Region Servers.

Region Server

Region Servers host regions and make them available for client access. From the Region Server, clients can execute read and write requests, but not any metadata changes or administration functions.

A single Region Server can host many regions from the same or different tables, as shown in Figure 7, and the Region Server is the unit of horizontal scale in HBase. To achieve greater performance, you can add more Region Servers and ensure there is a balanced split of regions across servers.

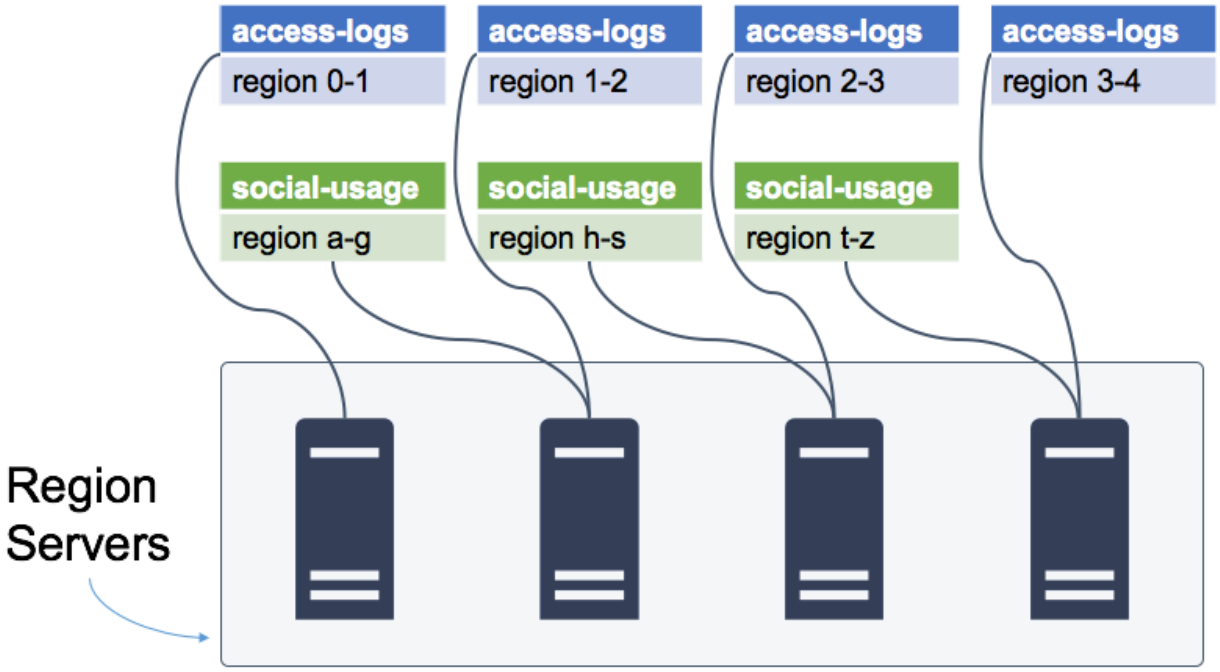


Figure 7: Regions Distributed Across Region Servers

Each region of a table is hosted by a singe Region Server, and client connections to other Region Servers will not be able to access the data for that region.

To find the right Region Server, clients query the HBase metadata table (called **`hbase:meta`**), which has a list of tables, regions, and the allocated Region Servers. The Master Server keeps the metadata table up-to-date when regions are allocated.

As we've seen, querying the HBase metadata to find a Region Server is usually abstracted in client code, so consumers work with higher-level concepts like tables and rows, and the client library takes care of finding the right Region Server and making the connection.

Region Servers are usually HDFS Data Nodes, and in a healthy HBase environment, every Region Server will host the data on local disks, for all the regions it serves. This is 100 percent data locality, and it means for every client request, the most a Region Server has to do is read from local disks.

When regions are moved between servers, then data locality degrades and there will be Region Servers that do not hold data for their regions locally, but need to request it from another Data Node across the network. Poor data locality means poor performance overall, but there is functionality to address that, which we'll cover in Chapter 9 Monitoring and Administering HBase," when we look at HBase administration.

Zookeeper

Zookeeper is used for centralized configuration, and for notifications when servers in the cluster go offline or come online. HBase predominantly use ephemeral **`znode`**s in Zookeeper, storing state data used for coordination among the components; it isn't used as a store for HBase data.

All the Master Servers and Region Servers use Zookeeper. Each server has a node, which is used to represent its heartbeat, as shown in Figure 8. If the server loses its connection with Zookeeper, the heartbeat stops, and after a period, the server is assumed to be unavailable.

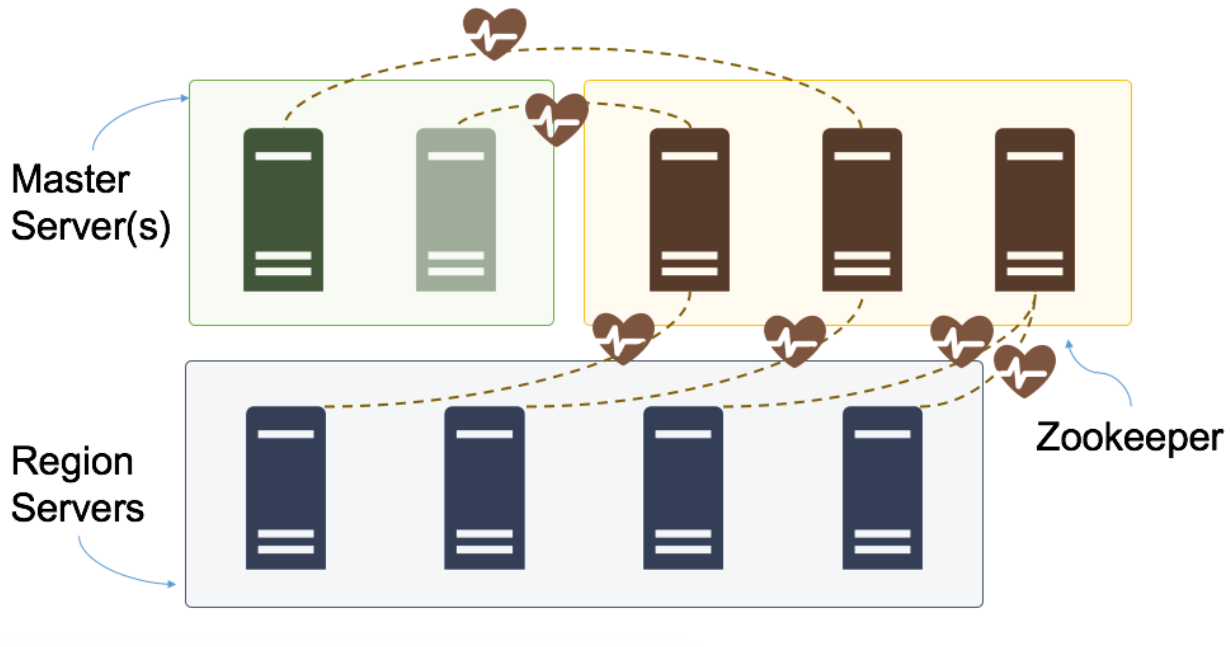


Figure 8: Heartbeat Connections to Zookeeper

The active Master Server has a watch on all the Region Server heartbeat nodes, so it can react to servers going offline or coming online, and the passive Master Server has a watch on the active Master Server's heartbeat node, so it can react to the Master going offline.

Although Zookeeper isn't used as a permanent data store, its availability is critical for the healthy performance of the HBase cluster. As such, the Zookeeper ensemble should run across multiple servers, an odd number to maintain a majority when there are failures (a three-node ensemble can survive the loss of one server; a five-node ensemble can survive the loss of two).

HDFS

A production HBase cluster is a Hadoop cluster, with the HDFS Name Node running on the Master Server and the Data Nodes running on the Region Servers.

The files which contain the data for a region and column family are stored in HDFS. You don't need a thorough understanding of HDFS to know where it fits with HBase. It's sufficient to say that data in HDFS is replicated three times among the Data Nodes, as shown in Figure 9. If one node suffers an outage, the data is still available from the other nodes.

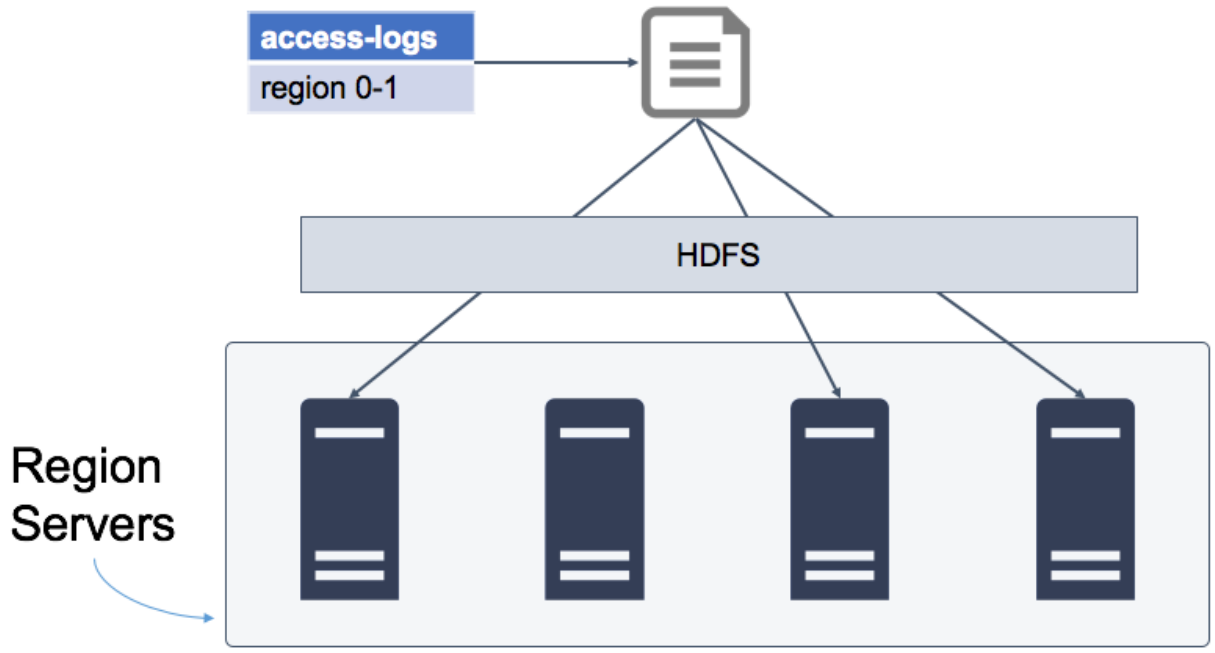


Figure 9: HBase Data Stored in HDFS

HBase relies on HDFS for data integrity, so the storage parts of the Region Server are conceptually very simple. When a Region Server commits a change to disk, it assumes reliability from HDFS, and when it fetches data from disk, it assumes it will be available.

Summary

In this chapter, we walked through the architecture of a distributed deployment of HBase. Client interaction is usually with the Region Servers, which provide access to table regions and store the data on the cluster using HDFS. Scaling HBase to improve performance usually means adding more Region Servers and ensuring load is balanced between them.

Region Servers are managed by the Master Server, if servers go offline or come online the Master allocates regions among the active servers. The Master Server also owns database metadata, for all table changes. Typically, the HBase Master is deployed on two servers, running in active-passive formation.

Lastly there's Zookeeper which is used to coordinate between the other servers, for shared state, and to notify when servers go down so the cluster can repair itself. Zookeeper should run on an odd number of servers, with a minimum of three, for reliability.

In the next chapter we'll zone in on the Region Server, which is where most of the work gets done in HBase.

Chapter 8 Inside the Region Server

Cache and Cache Again

Although the Region Server leverages HDFS so that it has a clean interface with the storage layer, there is added complexity to provide optimal performance. The Region Server minimizes the number of interactions with HDFS, and keeps hot data in caches so it can be served without slow disk reads.

There are two types of caches in each Region Server, shown in Figure 10:

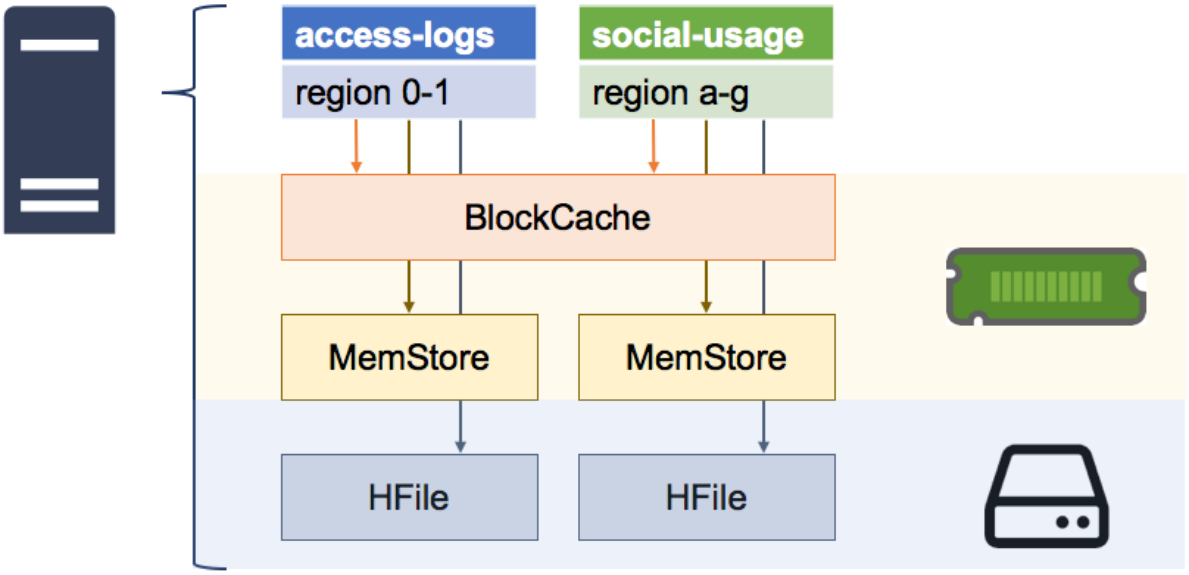


Figure 10: Caches in the Region Server

The BlockCache is a read cache that keeps recently fetched data in memory; the MemStore is a write cache that keeps recently written data in memory; and ultimately, there's the HFile, which contains the data on disk.

It's important to understand how those pieces fit together because they impact performance and will feed back into your table design and performance tuning.

The BlockCache

HBase uses an in-memory cache for data that has recently been fetched. Each Region Server has a single BlockCache, which is shared between all the regions the server is hosting, shown in Figure 11:

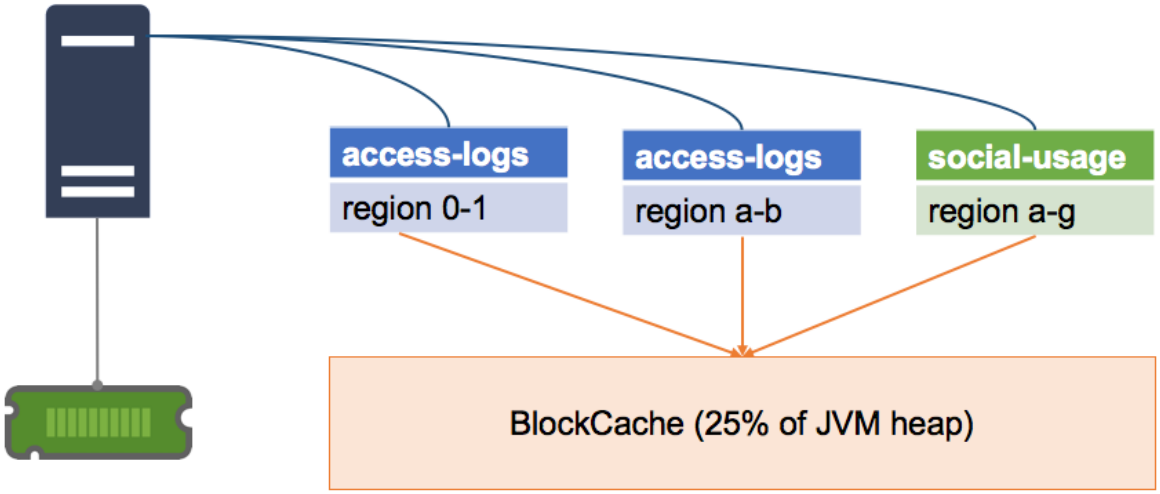


Figure 11: One BlockCache per Server

By default, the memory allocated to the BlockCache is 25 percent of the heap for the Java Virtual Machine. That size is limited by the physical memory in the server, which is one reason why having an excessive number of regions impacts performance—all the regions will compete for a limited amount of memory space.

When the Region Server gets a read request, it checks the BlockCache to see if the data is in memory. If so, it returns and the read will be very fast. If not, then the server checks the MemStore for the region to see if the data has been recently written. If not, then the server reads the data from the relevant HFile on disk.

The BlockCache uses a Least Recently Used eviction policy, so data items which are repeatedly accessed stay in the cache, and those which are infrequently used are automatically removed (although since caching is so critical to high performance, HBase allows you to tune the BlockCache and use different caching policy algorithms).

The MemStore

The MemStore is a separate in-memory cache, for storing recently written data, and a Region Server maintains one MemStore for each region it hosts, as shown in Figure 12:

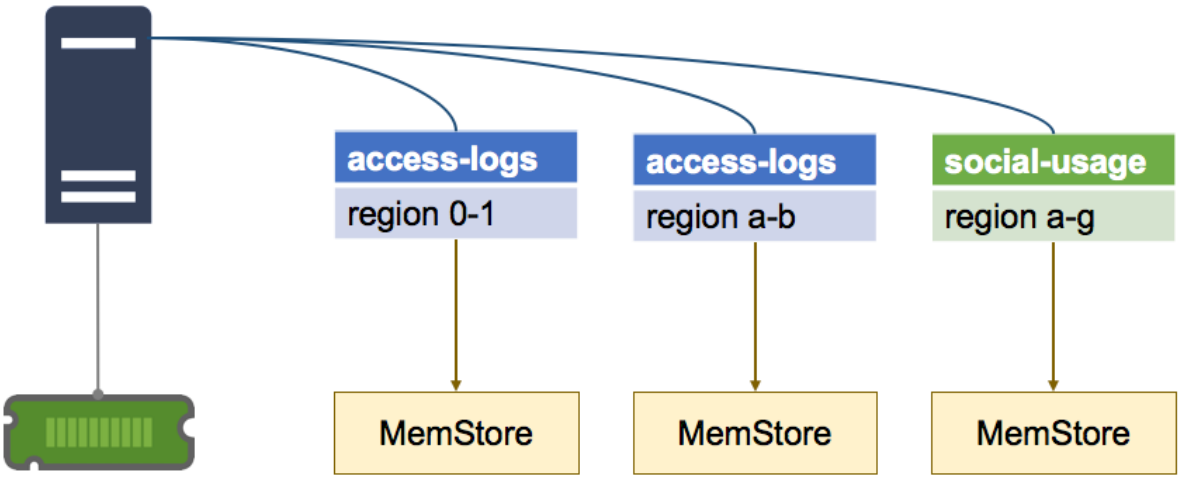


Figure 12: One MemStore per Region

The MemStore serves two purposes—the first is as a cache, so recently written data can be fetched from memory when it gets accessed, which reduces disk reads.

The second, more significant role of the MemStore is as a write buffer. Writes to a region aren't persisted to disk as soon as they're received—they are buffered in the MemStore, and the data in the buffer is flushed to disk once it reaches a configured size.

By default, the write buffer is flushed when it reaches 128MB, as shown in Figure 13:

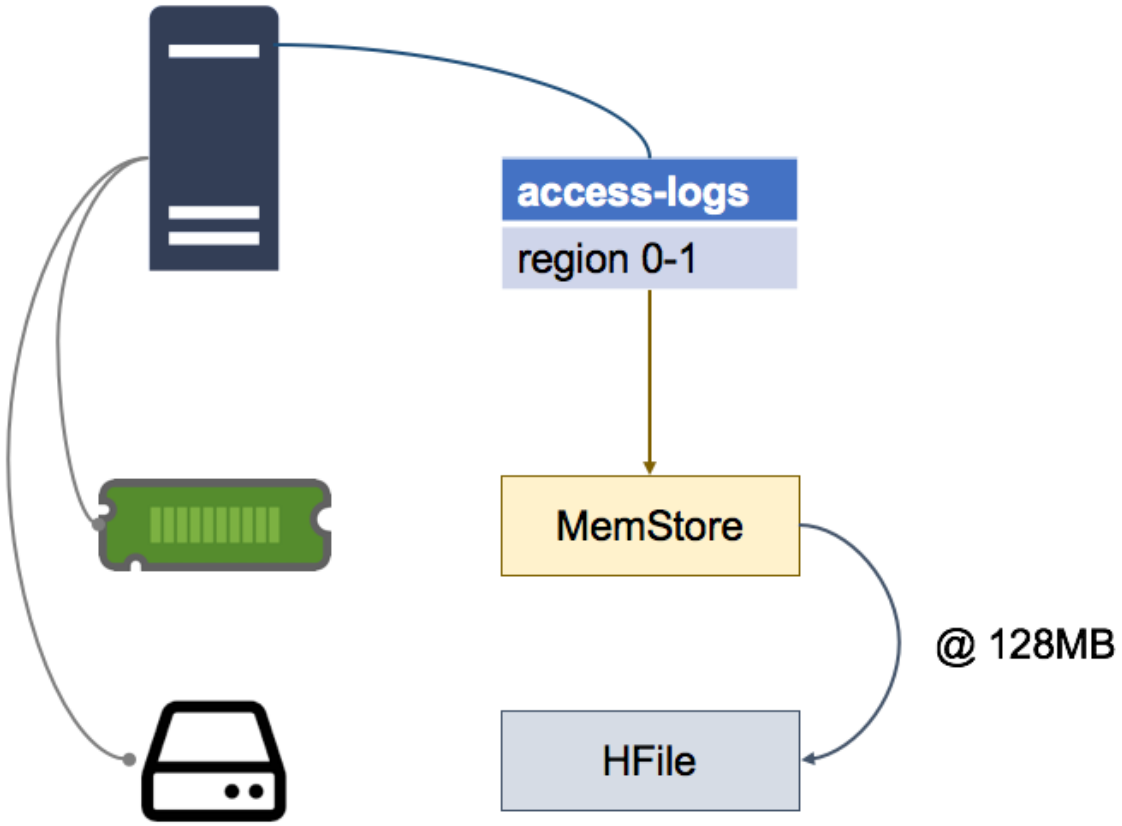


Figure 13: Flushing the MemStore

Data in the MemStore is sorted in memory by row key, so when it does get flushed, HBase just makes a series of fast, sequential reads to HDFS to persist the data, which is already in the correct order.

The data on disk is called an HFile, and logically a single HFile contains all the data for one column family in one region of one table. Because of the MemStore and the buffer-then-flush pattern, HFiles can be composed of many physical files, called Store Files, each containing the data from one MemStore flush.

Having multiple Store Files for a single HFile can affect read performance, as a fetch from a Region Server may require it to read many files on disk, so periodically HBase compacts the Store Files, combining small files into one large one.

You can also manually force a compaction (which we'll cover in Chapter 9 Monitoring and Administering HBase"). After a major compaction, each HFile will be contained in a single Store File on disk, which is the optimum for read performance.

Buffering and the Write Ahead Log

Buffering data as it's written and periodically flushing it to disk optimizes write performance of the Region Server, but it also creates the potential for data loss. Any writes that are buffered in the MemStore will be lost if the Region Server goes down, because they won't have been persisted to disk.

HBase has that scenario covered with the Write Ahead Log (WAL), stored in HDFS as a separate physical file for each region. Data updates are buffered in the MemStore, but the request for the data update gets persisted in the WAL first, so the WAL keeps a log of all the updates that are buffered in the MemStore, as in Figure 14:

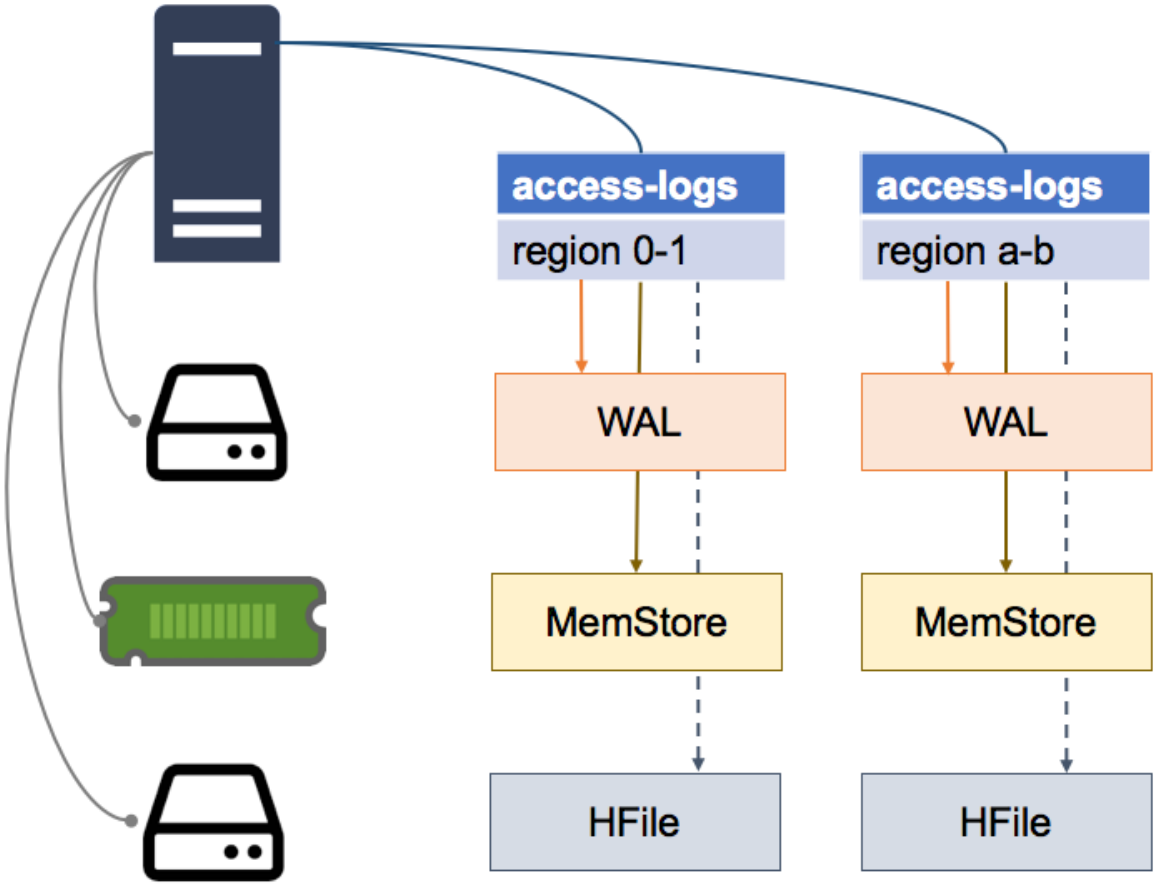


Figure 14: Write-Ahead Log Files

If a Region Server goes down, when the region is allocated to another server it checks the WAL before bringing the region online. If there are updates logged in the WAL, the new Region Server reads them and persists them all to the HFile before making the region available.

Region Servers don't acknowledge a write request until the update is committed to the WAL, and new Region Servers don't accept any requests while a region is being brought online (including flushing the WAL to disk).

In the event of server failure, there will be a period of downtime when data from the regions it hosted is unavailable, but data will not be lost.

HFiles and Store Files

The structure of an HFile is designed to minimize the amount of disk reads the Region Server has to do to fetch data. An HFile only contains the data for one column family in one region, so the Region Server will only access a file if it contains the data it needs.

Data in HFiles is stored in blocks, and sorted by row key. Each HFile contains an index with pointers from row keys to data blocks, as we see in Figure 15:

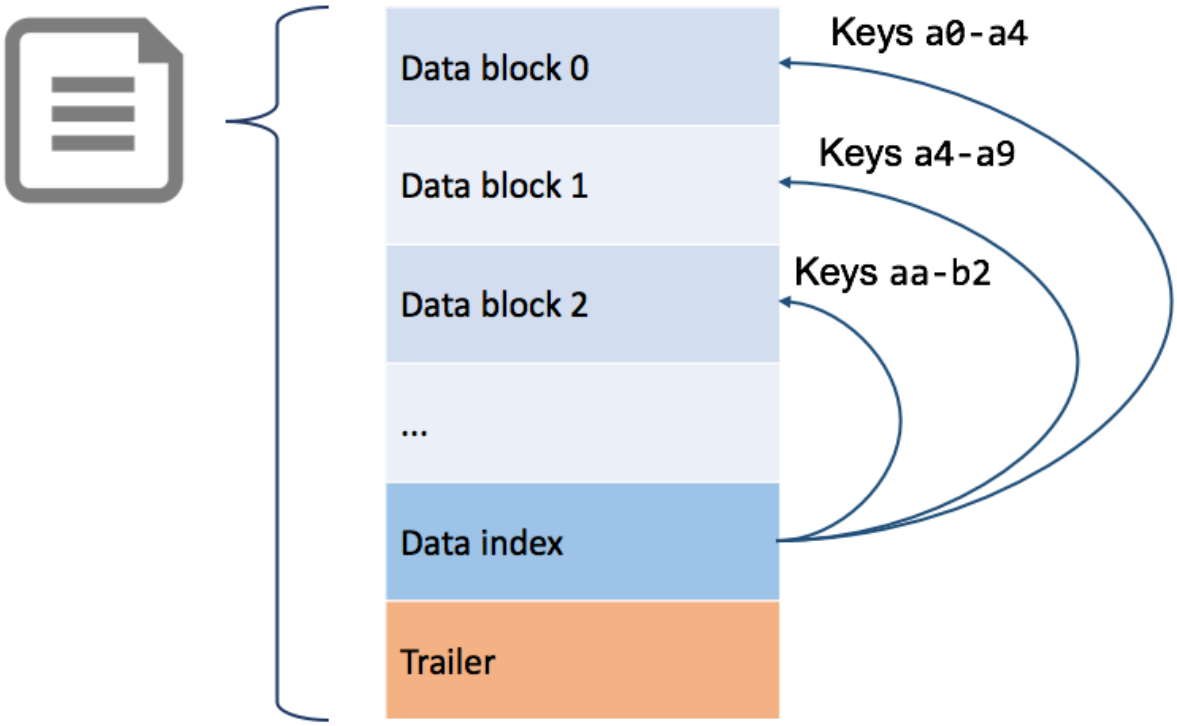


Figure 15: Structure of the HFile

Region Servers keep the HFile indexes for all the regions they serve in memory, so when data does have to be read from disk, the read can be targeted to the specific block that contains the data.

Over time, the data for a region can be fragmented across many sources, which puts additional work on the Region Server and degrades performance—this is called read amplification.

Read Amplification

For a table with intensive read and write access, the data in a region could be scattered across all the data stores in a Region Server. Recently fetched data will be in the BlockCache; recently written data in the MemStore; and old data in the HFile.

The HFile could also be composed of multiple Store Files, and for a single row we could have parts of its data in each of those locations, as in Figure 16, where different columns for the same row are spread across four stores:

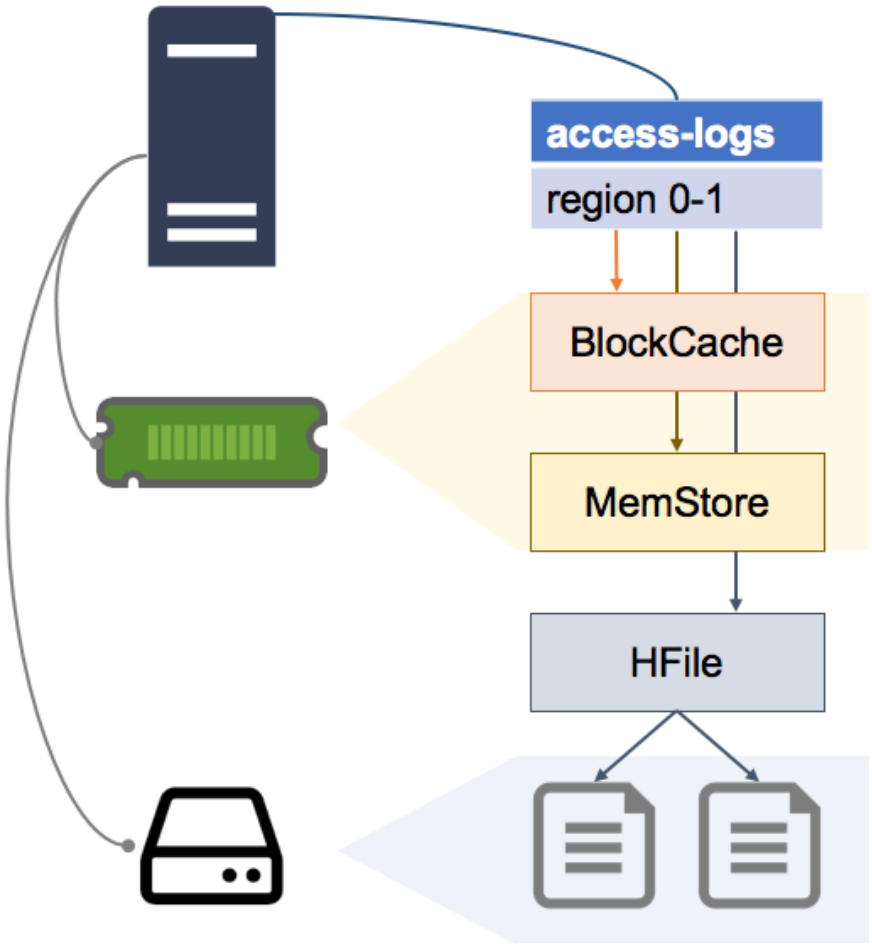


Figure 16: Read Amplification

A read request for the whole column family for that row will require the Region Server to read from all those stores, and merge the columns for the response.

Having many Store Files for each HFile means a single read request could amplify to many disk reads. In the worst case scenario, when the region is newly-allocated to a Region Server, the files may not be local, which means multiple network calls, and multiple disk reads on a remote server.

You can repair the performance degradation from this situation by running a major compaction manually, which we will cover in the final chapter.

Summary

In this chapter we looked inside the Region Server to see how data is actually stored in HBase, and how the Region Server processes requests to read and write data.

We saw that HBase has a read cache in every Region Server, and a write buffer for every region to improve performance, and a Write Ahead Log to ensure data integrity if a server goes down.

Ultimately, data is stored on disk in HFiles, where one HFile logically contains all the data for one column family in one region of one table. But the buffering pattern means a logical HFile could be split into multiple Store Files on disk, and this can harm read performance.

In the next chapter we'll look at monitoring and administering HBase through the HMaster Web UI and the HBase Shell, including finding and fixing those performance issues.

Chapter 9 Monitoring and Administering HBase

HBase Web UIs

All the HBase services provide a simple Web UI, which gives you basic information about how the service is running and gives you access to the logs. These are useful for troubleshooting and diagnostics, and the core services—the Master and Region Servers—have much richer Web UIs that tell you a lot about the health of the database.

The Web UIs are embedded using the Jetty Web Server (which has a low memory footprint and fast response times), so they start when you run the relevant service. They are all unsecured sites; they run with no authentication over HTTP. Each UI runs on a non-standard HTTP port, so if you want to limit the availability of the UI(s), you need to block external access to the port(s).

The Docker container **`hbase-succinctly`** has all the HBase servers running and all the UI ports exposed, so you can run an instance and browse to each of the UIs.

External API UIs

The Thrift and Stargate UIs provide the same basic information for each server. Figure 17 shows the home page for the Thrift UI, which runs on port 9095:

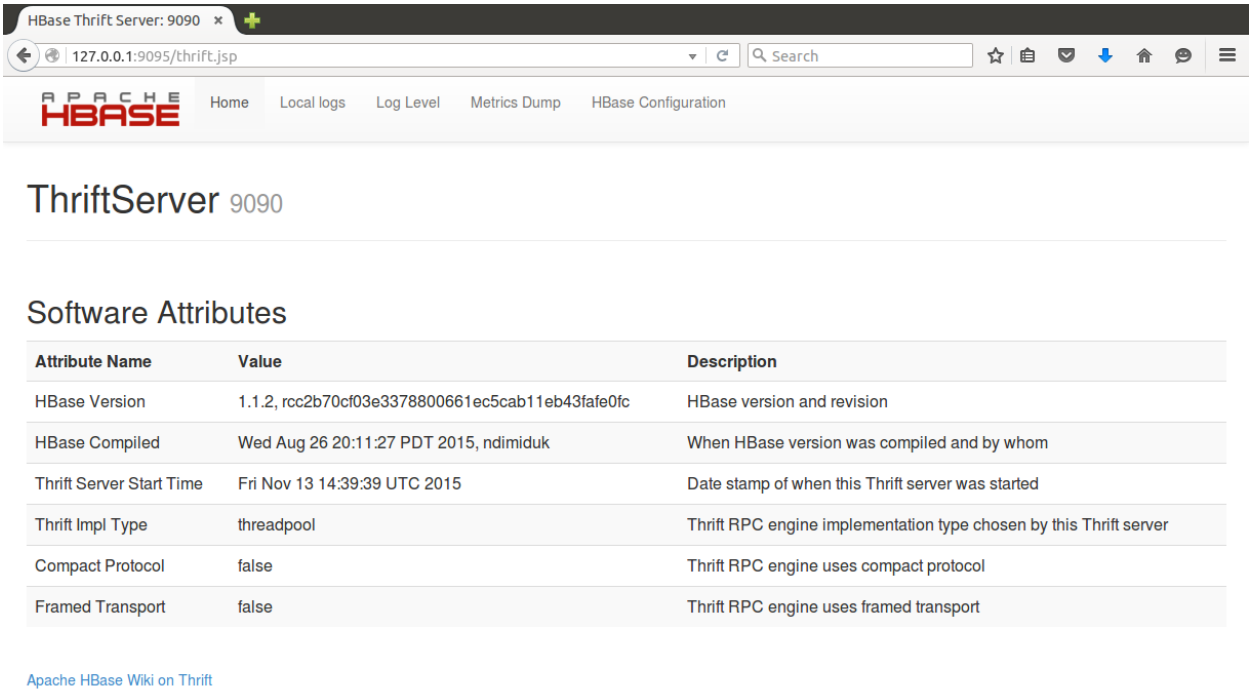


Figure 17: The Thrift Server UI

A similar interface is shown in Figure 18, the Stargate UI, which runs on port 8085:

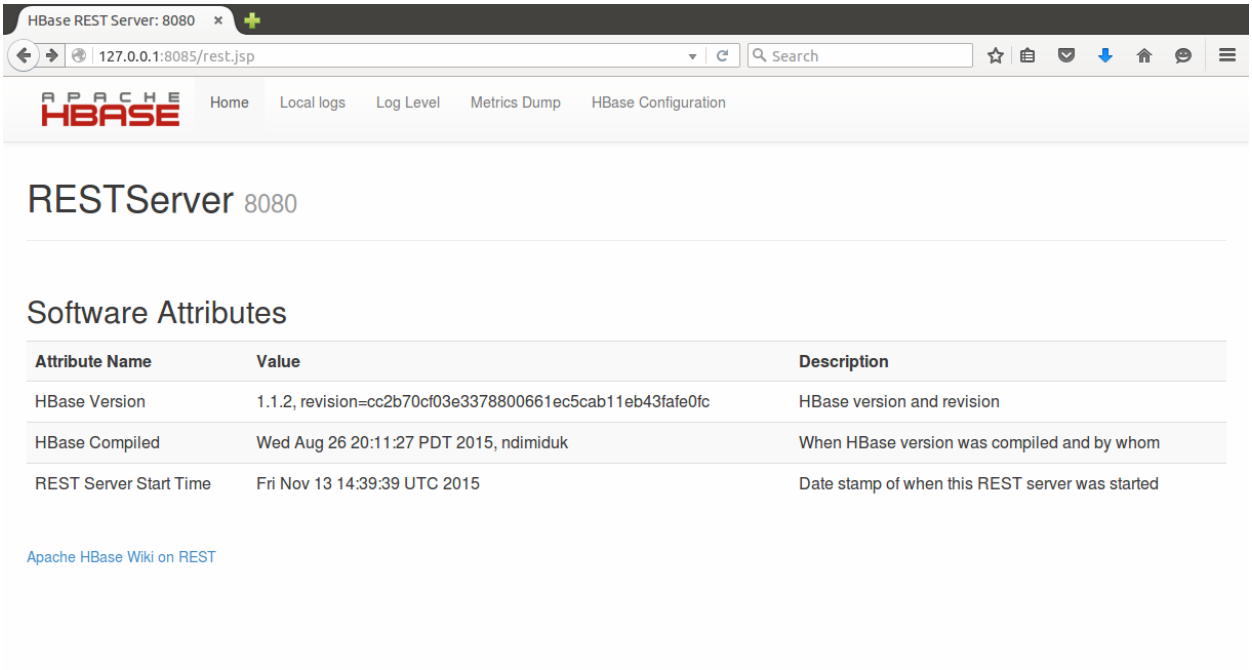


Figure 18: The Stargate UI

Both UIs show the exact version of HBase that's running, including the git commit ID (in the revision field), and the uptime of the server. If the server isn't running you won't get a response from the Web UI.

The Thrift UI also shows some key setup details (the transport and protocol types it accepts), and both UIs have the same set of links:

- Local logs – to view the log files on the server
- Log Level – to get or set the log levels for the server
- Metrics Dump – provides a detailed dump of the stats for the server
- HBase Configuration – shows all the configuration settings for the server

The last two links are worth looking at individually. Metrics Dump (URL is **`/jmx`** from the base path) gives a JSON output containing a huge amount of information, captured from Java Management Extensions running in the server.

You can use a JMX client to navigate the output, or add custom monitoring to query the server and parse the JSON as part of an automated diagnostic test. Code Listing 61 shows the JVM heap usage part of the output from the Metrics Dump:

Code Listing 61: JMX Metrics Dump

```
"HeapMemoryUsage": {
    "committed": 62652416,
    "init": 64689792,
    "max": 1018560512,
    "used": 16339616

}
```

The HBase Configuration link (**`/conf`** from the base URL) returns the active HBase configuration in XML. Each property contains the current value and the source of the value (which could be the server's **`hbase-site.xml`**, or the HBase default), which can be very useful for tracking down misconfigured environments.

Code Listing 62 shows the Zookeeper address from the HBase Configuration:

Code Listing 62: XML Configuration Settings

```
<property>
 <name>hbase.zookeeper.quorum</name>
 <value>localhost</value>
 <source>hbase-default.xml</source>
</property>
```

Region Server UI

On the Region Servers, the Web UI has the same log, metrics, and configuration links as the external API UIs, but it contains a lot more detailed information about the performance of the server and the data it's providing.

Figure 19 shows the home page of the Region Server UI, which runs on port 60030:

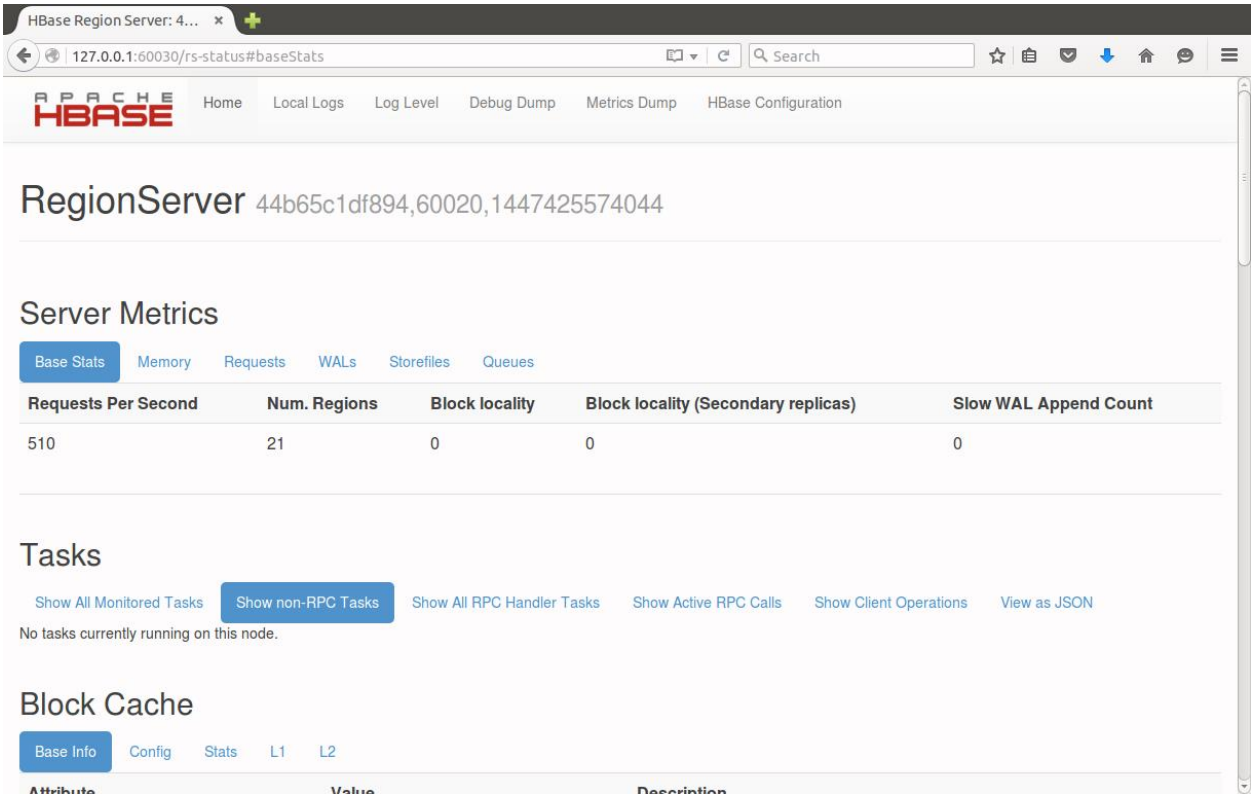


Figure 19: The Region Server Web UI

The Server Metrics section tells you how hard this one Region Server is working—how many regions it is hosting, and how many requests-per-second it is currently serving. These are requests from any of the APIs, so if you are using Java, Thrift, and Stargate clients, the total usage will show in here.

You can drill into the metrics to check the memory usage of the server and to see the breakdown of the WAL and Store Files that the regions are using. The Requests tab, shown in Figure 20, displays the cumulative requests served:

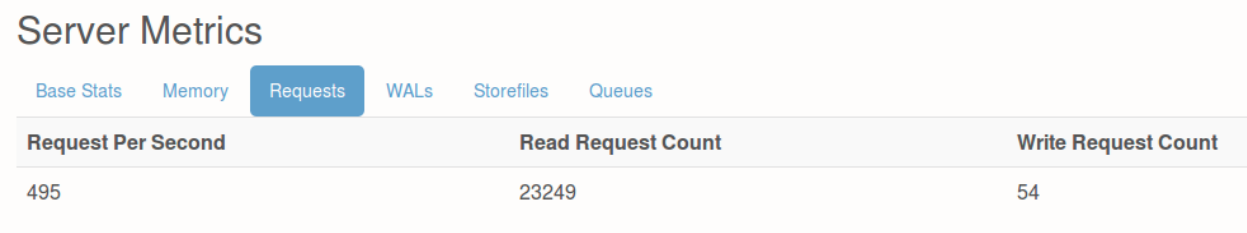


Figure 20: Server Metrics in the Region Server UI

Those stats are the combined read and write count across all regions, since the server has been running.

The home page also shows any long-running tasks, and their status. In Figure 21, the green bar highlights that the Block Cache for a region has finished flushing (in this case the metadata table, **`hbase:meta`**):

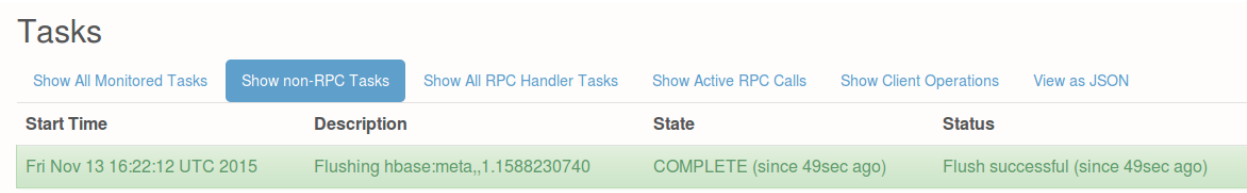


Figure 21: Task Status in the Region Server UI

The Region Server UI is worth exploring further. You can also get statistics on how well the caches are working and a breakdown of all the regions the server is hosting (with the start and end row keys), and the number of requests per region.

Those details are useful to see how your regions have been balanced across servers, and will help you detect any hot spots, as in Figure 22, where region **`e`** of the **`access-logs`** table has had 75K read requests, compared to 0 for the neighboring regions:

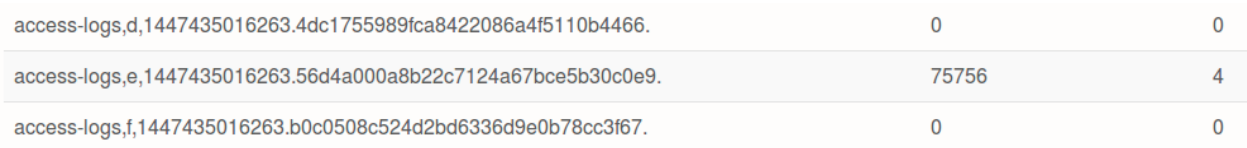


Figure 22: A Region Hot Spot in the UI

Master Server UI

Although the Region Server UI is useful, the Master Server UI runs at a higher level, displaying performance and metrics for all the Region Servers. It's the first place to look to check the health of your HBase system.

By default, the Web UI runs on port 60010 on the Master Server. If you are running multiple masters, only the active master shows any information. The home screen, shown in Figure 23, opens with a summary of all your Region Servers, stating how many regions each one is serving, and how many requests per second they are currently handling:

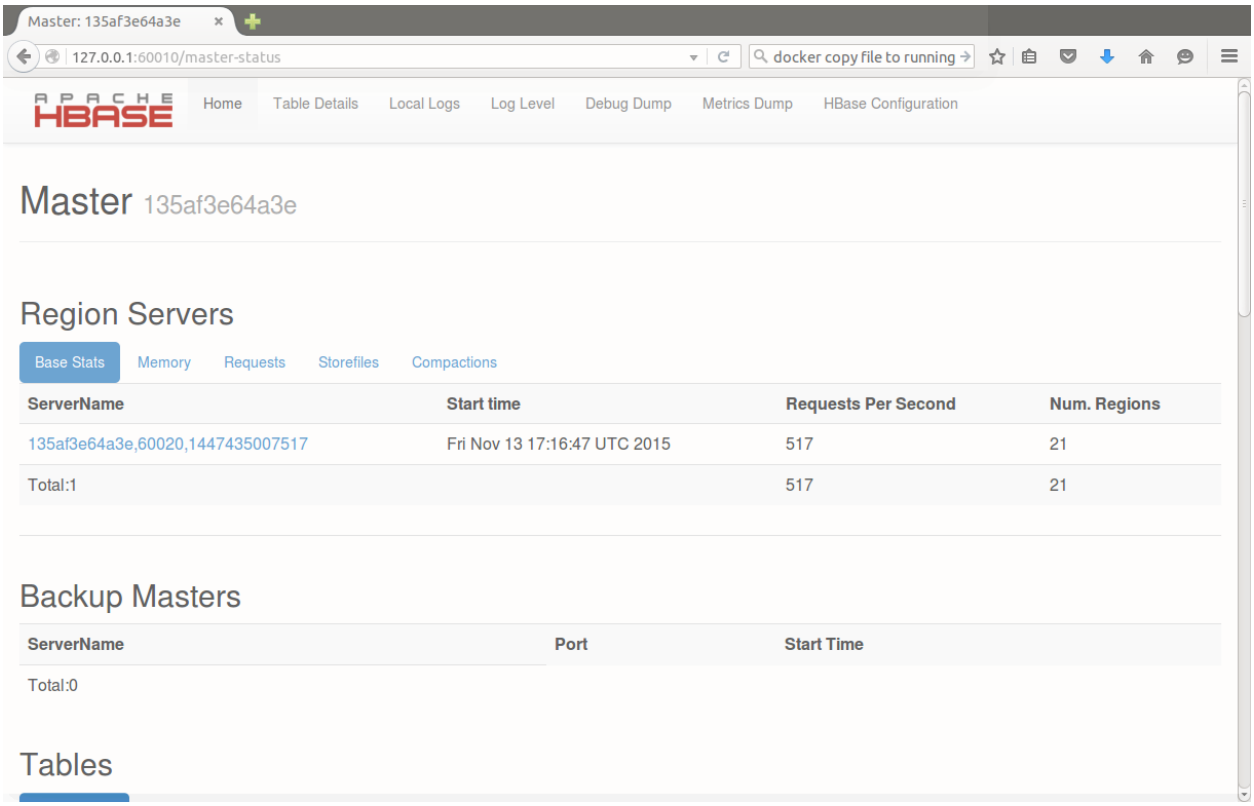


Figure 23: The Master Web UI

In the Region Servers section, the Master UI also surfaces the combined details from the Region Servers for their memory usage, individual region requests, and Store Files. The Compactions tab tells you if any major compactions are happening, and what status they're in.

From the top level navigation, you can access the local log files for the Master and change the log levels, as well as seeing the metrics dump and the HBase Configuration. The Debug Dump gives you very low-level detail on what the server is doing (which you rarely need unless it's misbehaving), but the Table Details link gives you the useful information shown in Figure 24:

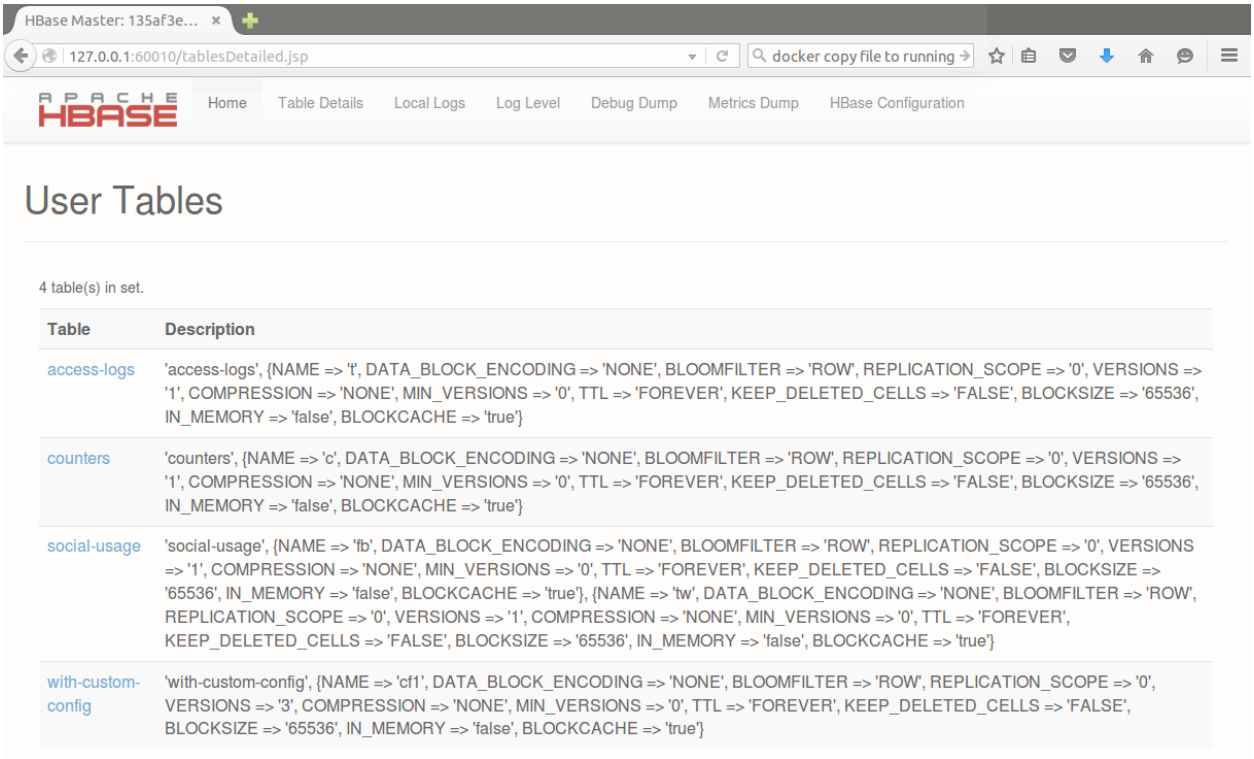


Figure 24: Table Details in the Master UI

This page shows each table defined in the database, and the full configuration (you can see the **`with-custom-config`** table from Chapter 2 has three versions specified for the column family, compared to the default of one in other tables).

Administering HBase with the Shell

All the client APIs we saw in Chapters 4, 5, and 6 have DDL features, but I prefer to keep client code free of admin work and use HBase Shell scripts for DDL and admin. There are also administrative functions that are only available from the Shell.

Typically, the admin functions you need to perform on HBase are for when your tables aren't being used in the way you had expected, and performance is suffering. Hot spots where tables have more active data in one region than others can be rectified by manually splitting the table.

The **`split`** command can be used at different levels, either splitting a named region or a named table. When you split an existing table, it's often because you're better informed on how the data is distributed, and HBase lets you provide a split key for the new region, as in Code Listing 63:

:

Code Listing 63: Splitting an Existing Table

```
hbase(main):006:0> split 'access-logs', 'ej'

0 row(s) in 0.4650 seconds
```

Figure 25 shows the table details from the Master UI, after the split where the original **e** region is now two separate regions:

access-logs,e,1447437294457.92e382a4ba6204c73e92d09bf22211e7.	135af3e64a3e:60020	e	ej
access-logs,ej,1447437294457.64d1f73621802d9974eaf17362baa857.	135af3e64a3e:60020	ej	f

Figure 25: Region Splits in the Master UI

When a region is manually split, the new region will be on the same server as the original region. That preserves data locality, as the original data was all on the one server, but it doesn't help if you want to remove a hot spot and share the data across more servers.

Any time your regions are not well balanced (for example, if you've manually split them, or added new tables), you can force a rebalance using the **`balancer`** command, which runs with no parameters, shown in Code Listing 64:

Code Listing 64: Initiating the Load Balancer

```
hbase(main):007:0> balancer
true

0 row(s) in 0.0680 seconds
```

The **`balancer`** command returns **`true`** if it was able to start the rebalancing process. The balancer actually runs asynchronously, and will continue after command returns. If regions are already being moved by the Master from another process, you can't start a rebalance and the command will return **`false`**.

The final situation that can harm performance is if your regions are split into many Store Files, and/or you have poor data locality, due to regions being moved or servers being added to the cluster. To fix that you need a major compaction, which combines two or more store files into a single HFile for a region and column family, and ensures each HFile is local to the Region Server that is hosting it.

HBase will run major compactions on a schedule, approximately every 24 hours, which will impact performance if it happens during peak access times. If you know your data is fragmented, then it's better to run a manual compaction when the server load is low.

Compaction is started with the **major_compact** command, and you can compact a whole table, a column family, a region, or just one column family within a region. Code Listing 65 shows a compaction starting for the **access-logs** table:

Code Listing 65: Starting a Major Compaction for a Table

```
hbase(main):002:0> major_compact 'access-logs'

0 row(s) in 0.2990 seconds
```

Compaction can take a long time, especially for large tables with many regions. The Region view in the Region Server UI shows you the compaction progress, but it all happens in the background and the compacting region(s) remain available to read and write while the compaction is running.

Summary

With embedded Web UIs in all the server processes, HBase gives you a lot of information on how the database is performing. Metrics are also collected using JMX, which makes it easy to plug them into a third-party monitoring tool like Ganglia.

The Thrift and Stargate UIs give you a basic overview to show how those servers are performing, but the Master and Region UIs give you rich insight into how your data is structured, and how it's being used. You can easily see hot spots where one Region Server is working harder than it should from the admin UIs.

With the HBase Shell you can rectify the more common performance problems, and we saw how to manually split tables, force the Master to rebalance regions across servers, and run a major compaction to optimize data at the storage level.

Next Steps

There's a lot more to HBase than I've covered in this short book, and plenty to explore if you think HBase will be a good fit for your next data store. The Docker image that accompanies this book is a good place to start if you want to try HBase out with your preferred client.

If you want to try HBase in a production environment without commissioning a suite of servers, you can run managed HBase clusters in the cloud with Elastic MapReduce on AWS, and HDInsight on Azure.

Both clouds offer a simple way to get started with a high-performance cluster, and with Azure you can sign up for a new account and try HBase on HDInsight free for one week. If you want to see how HBase works on Azure with other parts of the Hadoop ecosystem, I built a full solution in my Pluralsight course, HDInsight Deep Dive: Storm, HBase and Hive, which is out now.