Community Experience Distilled

Design and implement successful patterns to develop scalable applications with HBase

Mark Kerzner    Sujee Maniyam

HBase Design Patterns

With the increasing use of NoSQL in general and HBase in particular, knowing how to build practical applications depends on the application of design patterns. These patterns, distilled from extensive practical experience of multiple demanding projects, guarantee the correctness and scalability of the HBase application. They are also generally applicable to most NoSQL databases.

Starting with the basics, this book will show you how to install HBase in different node settings. You will then be introduced to key generation and management and the storage of large files in HBase. Moving on, this book will delve into the principles of using time-based data in HBase, and show you some cases on denormalization of data while working with HBase. Finally, you will learn how to translate the familiar SQL design practices into the NoSQL world. With this concise guide, you will get a better idea of typical storage patterns, application design templates, HBase explorer in multiple scenarios with minimum effort, and reading data from multiple region servers.

Who this book is written for

If you are an intermediate NoSQL developer or have a few big data projects under your belt, you will learn how to increase your chances of a successful and useful NoSQL application by mastering the design patterns described in the book. The HBase design patterns apply equally well to Cassandra, MongoDB, and so on.

What you will learn from this book

- Install and configure a Hadoop cluster and HBase
- Write Java code to read and write HBase
- Explore Phoenix open source project to talk to HBase in SQL
- Store single entities, generate keys, use lists, maps, and sets
- Utilize UUID for generic key generation to store data and deal with large files
- Use denormalization to optimize performance
- Represent one-to-many and many-to-many relationships and deal with transactions
- Troubleshoot and optimize your application

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 5 "Time Series Data"
- A synopsis of the book’s content
- More information on HBase Design Patterns

About the Authors

Mark Kerzner holds degrees in law, math, and computer science. He has been designing software for many years and Hadoop-based systems since 2008. He is a cofounder of Elephant Scale LLC, a big data training and consulting firm, as well as the co-author of the open source book Hadoop Illuminated. He has authored other books and patents as well. He knows about 10 languages and is a Mensa member.

I would like to acknowledge the help of my colleagues, in particular Sujee Maniyam, and last but not least, of my multitalented family.

Sujee Maniyam has been developing software for 15 years. He is a hands-on expert of Hadoop, NoSQL, and cloud technologies. He is a founder and the Principal at Elephant Scale (http://elephantscale.com/), where he consults and teaches big data technologies. He has authored a few open source projects and has contributed to the Hadoop project. He is an author of the open source book Hadoop Illuminated (http://hadoopilluminated.com/).

He is the founder of the Big Data Gurus meetup in San Jose, CA. He has presented at various meetups and conferences.

You can find him on LinkedIn at http://www.linkedin.com/in/sujeemaniyam or read more about him at http://sujee.net.

I would like to acknowledge all the kind souls who have helped me along the way. Special thanks go to my co-author, Mark Kerzner, who is also a cofounder and a dear friend and, not the least, thanks goes to my family—my understanding wife and two young kids—who put up with my crazy schedule.
HBase Design Patterns

Software plays a paramount role in today's world, and NoSQL databases are an important part of the modern stack. They are found wherever a subsecond response access to vast amounts of information is needed. However, there is a huge gap between the first "Hello World" example in a NoSQL database and creating practical, scalable, and stable applications. The aim of this book is to fill this gap and to give you practical guidelines for building NoSQL software.

The book is specifically formulated in terms of HBase, and there are a few areas of design where HBase might be different from Cassandra or MongoDB, for example, but most of the design patterns discussed here can be transferred to other NoSQL databases. You are expected to invest efforts in learning, which will lead to rewarding skills in the end.

What This Book Covers

Chapter 1, Starting Out with HBase, covers what HBase is and the various ways in which you can install it on your computer or cluster of computers, with practical advice on the development environment.

Chapter 2, Reading, Writing, and Using SQL, covers the HBase shell and gives the first example of Java code to read and write data in HBase. It also covers using the Phoenix driver for higher-level access, which gives back SQL, justifying the "Not-only-SQL" meaning of NoSQL.

Chapter 3, Using HBase Tables for Single Entities, covers the simplest HBase tables to deal with single entities, such as the table of users. Design patterns in this chapter emphasize on scalability, performance, and planning for special cases, such as restoring forgotten passwords.

Chapter 4, Dealing with Large Files, covers how to store large files in HBase systems. It also covers the alternative ways of storing them and the best practices extracted from solutions for large environments, such as Facebook, Amazon, and Twitter.

Chapter 5, Time Series Data, shows that stock market, human health monitoring, and system monitoring data are all classified as time series data. The design patterns for this organize time-based measurements in groups, resulting in balanced, high-performing HBase tables. Many lessons are learned from OpenTSDB.
Chapter 6, *Denormalization Use Cases*, discusses one of the most common design patterns for NoSQL denormalization, where the data is duplicated in more than one table, resulting in huge performance benefits. It also shows when to unlearn one's SQL normalization rules and how to apply denormalization wisely.

Chapter 7, *Advanced Patterns for Data Modeling*, shows you how to implement a many-to-many relationship in HBase that deals with transactions using compound keys.

Chapter 8, *Performance Optimization*, covers bulk loading for the initial data load into HBase, profiling HBase applications, benchmarking, and load testing.
In this chapter, we will cover the following topics:

- What we mean by time series data
- The special design challenges that time series data presents
- Which important performance considerations apply
- The best practices to consider when dealing with time series data

All of us are familiar with time series data. Although we will provide a definition and examples later, you can think of it as minute-by-minute trading data. Such data presents a number of challenges for the HBase designer, as they need to create a proper schema for the data, find a balance between convenience and performance, and also keep factors such as overheating of specific region servers and bloated bloom filters in mind.

If all of this sounds obscure at the moment, we will try to make it clear by the end of this chapter. As we will see, HBase is perfect for storing time series data. However, one needs to pay attention to a number of pesky details.

Let's start with a definition of time series data. Time series data is a type of data that is recorded at regular time intervals. Some examples of time series data include logs, sensor data (power grid), stock ticks, monitoring systems, and many others.

Please note that logs can have a few different meanings. In the modern world, when talking about logs, we are usually referring to web logs or the recording of events that happens when users browse a website and a web server sends them the pages and records their actions.
Time Series Data

The second type of log, however, is the application log. It's a file created by an application that is executing and using a logging system. These are managed by software developers to record what is going on, for compliance reasons, and to help them debug issues.

There is another type of log too, namely the well log. This happens when a well-logging tool is pulled up the oil well and it is recording measurements such as resistivity or magnetic resonance. These are later used to analyze the formation of oil deposits and find them. This subject is dear to my heart, because I have written a book on this, *Image Processing in Well Log Analysis*, and it has recently been republished.

To summarize, there are vast varieties of time series data, and each one of them is important in its own right. Each poses a specific interesting challenge when HBase is used to store this type of data. Therefore, we will discuss the following areas that relate to time series data:

- Using time-based keys
- Avoiding region hotspotting
- Tall and narrow rows versus wide rows
- OpenTSDB principles
Using time-based keys to store time series data

The most natural key to use in order to store time series data is time (for example, in milliseconds). This is guaranteed to be unique, since this is exactly what you are recording—a measurement at a particular time. To give our example a practical flavor, let’s take a look at a well log, as all well logs are perfect examples of time series data.

(The source of this image is Baird Petrophysical found at http://www.bairdpetro.com/ and has been used with their due permission.)
As you can see, well logs record a certain measurement (in this case, it is sonic waves) on the millisecond scale. So, let's imagine for the sake of our example that we are recording the data every millisecond, and the HBase table contents look similar to this:

<table>
<thead>
<tr>
<th>Key (time, milliseconds)</th>
<th>Family:Column name</th>
<th>Family:Column name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1402282760</td>
<td>Logs:Lith, Logs:SW, Logs:Res</td>
<td>Other: Depth, Other: acceleration</td>
</tr>
<tr>
<td>1402282761</td>
<td>Logs:Lith, Logs:SW, Logs:Res</td>
<td>Other: Depth, Other: acceleration</td>
</tr>
<tr>
<td>1402282762</td>
<td>etc.</td>
<td>etc.</td>
</tr>
</tbody>
</table>

Okay, so this looks great and clear. You can get the data at any millisecond in one read. You can also perform a nice scan on the range of key values. Since all the rows that are stored in HBase are already sorted by key, your scans are guaranteed to be fast.

However, there are two problems with this approach. They are as follows:

- The first problem is that it can result in the overloading (overheating) of some of the region servers, because during writes, all the data is concentrated in the regions they serve and no data is recorded in the other regions. Similarly, during typical reads against the most recent data, we will be querying a small number of regions.

- The second problem with this approach is that you are storing relatively few columns in each row, and this might be very inefficient due to very little information being read at one time and due to the presence of too many bloom filter values.

Each of the potential problems can kill the performance of our application. It is, therefore, important to understand both well.

**Avoiding region hotspotting**

This refers to our first problem, as monotonically increasing key values are bad. Why is that?

This is very well explained by Ikai Lan, a Google engineer at the time when he wrote this explanation, and he's currently working for Developer Relations at Google NYC. Ikai was also an early inspiration for the doodles and cartoons used in this book and my other big data cartoon series, which can be found at http://shmsoft.blogspot.com/search/label/Hadoop%20cartoons, so he deserves a special acknowledgement.
When you write all the new rows sequentially, they all end up being on the same server, because they are sorted and this forces them to be close to each other. It is true that there is sharding, and moreover, there is automatic sharding. This means that the new regions (the areas of the hard drive where the data is written) will eventually come into play, but this happens only later. Right now, you have got a hotspot.

Practically, you won't notice this under low write speeds. If you have less than a hundred writes per second, this is not important since your RegionServer copes quite all right. This number (hundred writes per second) might change on your hardware and HBase, but it will be in the low range, because it means that you are not utilizing your entire HBase cluster but only one server. This is how Ikai illustrates this:
Time Series Data

(The source of this image is http://ikaisays.com/2011/01/25/app-engine-datastore-tip-monotonically-increasing-values-are-bad/, and the image has been used with their due permission.)

Actually, Ikai is talking in terms of Google’s BigTable, the design on which HBase is based. BigTable shards (divides the load) not only when it outgrows the region size but also when it sees a disproportionately high load on one table. As far as I know, HBase does not have such a provision. All the more, we should be careful not to concentrate our keys. So, what should you do?

You can do one of the following things:

- **Avoid indices if you can:** In your case, you will have to record the row using something else instead of time as the key. This is not the perfect approach in the case of well logs, since you need to know when the values were recorded. The depth of the measurement will give you the same problem.

- **Randomize your writes:** Even pseudorandomization will help you to offload some of the servers. Again, in our case, the sorted time is essential information; we will be hard pressed to do without it. The two pieces of advice given so far might be good for other situations (such as writing dictionary words in a random sequence), but not for true time series data. Your writes will be faster, but your reads will be slower, because you will have to collect the information from many places.

- **Prefix a shard identifier to your key:** You can distribute the load between multiple servers yourself. Now, when you are reading the data back, you will have to read it from each server, prefix all the possible server numbers to your time, and combine the query results in memory. A bit of a bother, but this will work.

The three pieces of the preceding advice are actually good, each for their own situation. It’s just that for time series data, only the last one is practical. We will see how to write the code for this later in this chapter.

A partial solution to the problem can be provided by preloading, which we will discuss later. In brief, preloading, or more technically bulk loading, comes into play when you already have a lot of data to write to HBase. In this case, you can select the number of regions to be created. Then, all of them are used. If you combine this with the real read/write load, you might be lucky and your workload might already be distributed between the regions (shards) that you have previously created.
Tall and narrow rows versus wide rows
This section refers to our second problem of records being too short.

If the record that you write for one time series set is too small, you run into cache-hit problems and very large bloom filters. This is easy to understand. Hadoop reads blocks from HDFS, which usually range in size between 64 MB to 256 MB. If your data reads and writes are much smaller than this, you are bound to see inefficiencies and hence the cache-hit problems. Bloom filters answer the question—based on the key, is it possible that your data resides in the given region? The answer no is definite, but the answer yes should be understood as maybe yes, necessitating a read of the region and a search. If your keys are responsible for very thin rows with little information, you will have too many bloom filter keys, which will both take the hard drive space and reduce the efficiency using bloom filters in the first place.

What are the practical limits? Anything starting with the HDFS block size and ending in tens of thousands of rows. Even though, theoretically, you can have millions of columns, practical researcher Patrick McFadin reports that after tens of thousands of columns, you start eating into 95 percentiles of your read latencies and this is mostly due to deserialization costs on the larger indexes.

OpenTSDB principles
OpenTSDB is a set of tools that allow you to store and retrieve time series data. It uses HBase for data storage and retrieval, but isolates you, the user, from HBase completely. Thus, you don't have to know or care about HBase (other than administer it). To the user, it is a very simple tool, which asks them to send the time series data and then allows all kinds of displays.

We will use OpenTSDB, as promised, for two purposes:

- To teach you the use of the tool for your needs
- To elucidate on the design principles so that you can use them in your own coding, if you cannot use OpenTSDB tools directly but need a similar functionality in your application
Let's first take a look at how to use the tools of OpenTSDB without any additional code development. This is what the tools allow you to do:

- Collect the data from various sources for OpenTSDB. For example, you can use any of the following provided tools:
  - **Vacuumetrix**: This pulls data from various cloud services or APIs and stores the results in the backend, such as Graphite, Ganglia, and OpenTSDB
  - **Statsd publisher**: This publishes data to TSD with StatsD
  - **Flume module**: This writes data from Flume to TSD

- Display the data in a nice graphical form, such as the one shown next, using gnuplot:
• Use the standard OpenTSDB graph, such as the one shown here, for monitoring, which is the primary OpenTSDB use case:

![OpenTSDB Graph Example](image)

However, here we are more interested in the OpenTSDB design principles, since this shows us the experience of others rather than our limited experience. So, the designers give us those pointers (with the explanation).

**The overall design of TSDB**

Here are the lessons that we can glean from the overall design of TSDB. All OpenTSDB data points are stored in a single, massive table, named `tsdb` by default. This is to take advantage of HBase's ordering and region distribution. All the values are stored in the `ts` column family.

The lesson that we learned here is to use a simple schema, as you will have enough complexity anyway. So, go with the recommended HBase design.

**The row key**

Row keys are byte arrays that comprise of the metric UID, a base timestamp, and the UID for tag key-value pairs, for example, `<metric_uid><timestamp><tagk1><tagv1>[...<tagkN><tagvN>]`. By default, UIDs are encoded on three bytes.
Time Series Data

The lesson that we learn here is to use composite keys, take care of load balancing by adding the UID at the beginning, put the timestamp in the key, and add additional information at the end. Note that, ordering is done inside the composite key, thus reflecting the types of queries we anticipate.

The timestamp

The timestamp is a Unix epoch value in seconds, encoded on 4 bytes. Rows are broken up into hour increments, reflected by the timestamp in each row. Thus, each timestamp will be normalized to an hour value, for example, 2013-01-01 08:00:00. This is to avoid stuffing too many data points in a single row as that would affect region distribution. However, note that it can result in a large number of data points if the frequency of data generation is high.

Also, since HBase sorts the data on the row key, the data for the same metric and time bucket, but with different tags, will be grouped together for efficient queries. This assumes that the number of tags is small, and indeed OpenTSDB limits it to eight tags.

When storing time series data, implement the following best practices:

- Store a reasonable time interval per row. The amount of data should not make the table too tall and thin or too narrow and wide. One hour was chosen here.
- Use tags to store the time interval designation.
- Use your own data encoding, since we deal with binary data here.
- Take advantage of the natural sorting of columns in the row.
- Design for efficient access.

Compactions

Why is compaction required? The answer is to reduce the storage (as the key is repeatedly stored for each column). If compactions have been enabled for a TSD, a row might be compacted after its base hour has passed or a query has run over the row. The lesson here is that in your design, keep the compactions, both minor and major, in mind, because they will affect the performance.
The UID table schema
A separate, small table called tsdb-uid stores UID mappings, both forward and reverse. Two columns exist, one called name that maps a UID to a string and another called id that maps strings to UIDs.

Here, we will learn to keep the lookup tables in the same HBase. A small table can be directed to be served from the memory, so you can use HBase for its in-memory database capabilities too. Additional information can be gleaned from the OpenTSDB design documentation found at http://opentsdb.net/docs/build/html/user_guide/backends/hbase.html.

For practical purposes, if your NoSQL database is Cassandra and not HBase, you will find that there is a Cassandra implementation of TSDB, called KairosDB, which can be found at https://code.google.com/p/kairosdb.

Summary
We learned the important NoSQL and HBase design principles related to storing and accessing time series data. We described how to utilize time-based keys, how to avoid region hotpotting, how to properly balance tall and narrow rows versus wide rows, and we also saw how to glean additional information from other systems on top of HBase, such as OpenTSDB. All this together should make us masters at storing and using time series data in HBase and in NoSQL databases in general.

In the next chapter, we will deal with the most common design principle, that is, denormalization. We will also discuss how to store all the objects for a user, popularity contest, and how to store tags efficiently.
Where to buy this book

You can buy HBase Design Patterns from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.