Chapter No. 2
"Defining the Schema"
In this package, you will find:
The author’s biography
A preview chapter from the book, Chapter no.2 "Defining the Schema"
A synopsis of the book’s content
Information on where to buy this book

About the Author

**Nishant Garg** has over 14 years of experience in software architecture and development in various technologies such as Java, Java Enterprise Edition, SOA, Spring, Hibernate, Hadoop, Hive, Flume, Sqoop, Oozie, Spark, Shark, YARN, Impala, Kafka, Storm, Solr/Lucene, and NoSQL databases including HBase, Cassandra, MongoDB, and MPP Databases such as GreenPlum.

He received his MS degree in Software Systems from Birla Institute of Technology and Science, Pilani, India, and is currently working as a technical architect in Big Data R&D Group in Impetus Infotech Pvt. Ltd.

Nishant, in his previous experience, has enjoyed working with the most recognizable names in IT services and financial industries, employing full software life cycle methodologies such as Agile and Scrum. He has also undertaken many speaking engagements on Big Data Technologies and is also the author of *Apache Kafka*, Packt Publishing.

For More Information:
I would like to thank my parents, Shri. Vishnu Murti Garg and Smt. Vimla Garg, for their continuous encouragement and motivation throughout my life. I would also like to say thanks to my wife, Himani, and my kids, Nitigya and Darsh, for their never-ending support, which keeps me going.

Finally, I would like to say thanks to Vineet Tyagi, head of Innovation Labs, Impetus Technologies, and Dr. Vijay, Director of Technology, Innovation Labs, Impetus Technologies, for having faith in me and giving me an opportunity to write.

For More Information:
HBase Essentials

Apache HBase is an open source distributed, Big Data store that scales to billions of rows and columns. HBase sits on top of clusters of commodity machines.

This book is here to help you get familiar with HBase and use it to solve your challenges related to storing a large amount of data. It is aimed at getting you started with programming with HBase so that you will have a solid foundation to build on about the different types of advanced features and usages.

What This Book Covers

Chapter 1, Introducing HBase, introduces HBase to the developers and provides the steps required to set up the HBase cluster in the local and pseudo-distributed modes. It also explains briefly the basic building blocks of the HBase cluster and the commands used to play with HBase.

Chapter 2, Defining the Schema, this answers some basic questions such as how data modeling is approached and how tables are designed in the first half of the chapter. The next half provides the examples of CRUD operations in HBase using the Java-based developers API provided by HBase.

Chapter 3, Advanced Data Modeling, takes the concepts discussed in the previous chapter into more depth. It explains the role of different keys in HBase and later picks up advanced features such as table scan and filters in detail.

Chapter 4, The HBase Architecture, provides an insight into the HBase architecture. It covers how data is stored and replicated internally in HBase. It also discusses how to secure HBase access and explains HBase and MapReduce over Hadoop integration in detail.

Chapter 5, The HBase Advanced API, shares the advanced features such as counters, coprocessors, and their usage using the HBase developers' API. It also discusses the API available for the HBase administration.

Chapter 6, HBase Clients, discusses in detail various clients that are available for HBase. The HBase client list includes HBase shell, Kundera, REST clients, Thrift client, and Hadoop ecosystem clients.

Chapter 7, HBase Administration, focuses on HBase administration. It provides details about the HBase cluster management, monitoring, and performance tuning. In the end, it talks about cluster troubleshooting.

For More Information:
Defining the Schema

In this chapter, we are going to learn some of the basic concepts of the column family database, that is, HBase, and cover the following topics:

- Data modeling
- Designing tables
- CRUD operations

Let's dive in and start off by taking a look at how we can model data in HBase.

Data modeling in HBase

In the RDBMS world, data modeling has principles around tables, columns, data types, size, and so on, and the only supported format is structured data. HBase is quite different in this aspect, as in each row, it can store different numbers of columns and data types, making it ideal for storing so-called semi-structured data. Storing semi-structured data not only impacts the physical schema but also the logical schema of HBase. For the same reason, some features such as relational constraints are also not present in HBase.

Similar to a typical RDBMS, tables are composed of rows and these rows are composed of columns. Rows in HBase are identified by a unique rowkey and are compared with each other at the byte level, which resembles a primary key in RDBMS.

In HBase, columns are organized into column families. There is no restriction on the number of columns that can be grouped together in a single column family. This column family is part of the data definition statement used to create the HBase table.

For More Information:
At the storage level, all columns in a column family are stored in a single file, called HFile, as key-value pairs in the binary format. These HFiles are ordered immutable maps which are internally represented as data blocks with a block index.

In HBase, the placeholder for the column value is called cell. Each cell stores the most recent value and the historical values for the column. These values are placed in a descending order on the timestamp and ensure a faster read performance.

Each value contained within a cell in the table can be represented by a combination of the rowkey, column family, column key, and timestamp. The following image of a table shows the organization of values in the table:

<table>
<thead>
<tr>
<th>Row Keys</th>
<th>Column Family :: Customer</th>
<th>Name</th>
<th>Email</th>
<th>Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROW 1</td>
<td></td>
<td>David</td>
<td><a href="mailto:david@gmail.com">david@gmail.com</a></td>
<td>982 765 2345</td>
</tr>
<tr>
<td>ROW 2</td>
<td></td>
<td>John</td>
<td><a href="mailto:john@rediff.com">john@rediff.com</a></td>
<td>763 456 1234</td>
</tr>
<tr>
<td>ROW 3</td>
<td></td>
<td>Elan</td>
<td><a href="mailto:elan@hotmail.com">elan@hotmail.com</a></td>
<td>554 123 0987</td>
</tr>
<tr>
<td>ROW 4</td>
<td></td>
<td>Maria</td>
<td><a href="mailto:maria@test.net">maria@test.net</a></td>
<td>763 451 4587</td>
</tr>
</tbody>
</table>

Each Cell may have multiple version of data distinguished by time stamp

Like column families that group columns, HBase has a concept called regions, where it groups the continuous range of rows and stores them together at lower levels in region servers. Regions can also be thought of as data partitions in the RDBMS world and help to achieve scalability in the overall HBase architecture. A maximum size is defined for regions, and once the limit is exceeded, the region is split into two from the middle. This process is synonymous to auto-sharding in the RDBMS world.

In HBase, records are stored in HFiles as key-value pairs, and this HFile is, in turn, stored as a binary file. Records from a single column family might be split across multiple HFiles, but a single HFile cannot contain data for multiple column families.

For More Information:
The preceding diagram shows how different data storage level components are organized. The region server contains the region sales, having two column families, customer and orders. Each column family has an in-memory storage, and single column family records might have been written to multiple HFiles. In HBase, the region server is treated as a datanode and supports multiple regions (represents tables here). When data is written, by default, it goes to the write ahead log (WAL) and the MemStore. HFile is created once the data present in the MemStore is flushed to the disk and in case of flush failure, data is retrieved from WAL (we will discuss all these components in detail in Chapter 4, The HBase Architecture).

Designing tables
In HBase, when modeling the schema for any table, a designer should also keep in mind the following, among other things:

- The number of column families and which data goes to which column family
- The maximum number of columns in each column family
- The type of data to be stored in the column
- The number of historical values that need to be maintained for each column
- The structure of a rowkey

For More Information:
Defining the Schema

Once we have answers, certain practices are followed to ensure optimal table design. Some of the design practices are as follows:

- Data for a given column family goes into a single store on HDFS. This store might consist of multiple HFiles, which eventually get converted to a single HFile using compaction techniques.
- Columns in a column family are also stored together on the disk, and the columns with different access patterns should be kept in different column families.
- If we design tables with fewer columns and many rows (a tall table), we might achieve O(1) operations but also compromise with atomicity.
- Access patterns should be completed in a single API call. Multiple calls are not a good sign of design.

We not only need to design the table schema to store data in a column-family layout but also consider the read/write pattern for the table, that is, how the application is going to access the data from an HBase table. Similarly, rowkeys should be designed based on the access patterns, as regions represent a range of rows based on the rowkeys and the HFiles store the rows sorted on the disk. Hence, the rowkey is a crucial element to the performance of I/O interactions with HBase.

HBase doesn't support cross-row transactions, so the client code should avoid any kind of transactional logic to support simplicity.

HBase drives the design from BigTable of Google as one-row-per-account which might easily hold multiple terabytes in a single row with no problems or with a poor design. However, the same information can also be stored in a tall table (lots of rows with fewer columns), which also provide performance benefits. This performance benefit also comes with a cost of atomicity. The physical storage for both the table designs is essentially the same.

For More Information:
Accessing HBase

In the previous chapter, we saw how to create a table and simple data operations using the HBase shell. HBase can be accessed using a variety of clients, such as REST clients, Thrift client, object mapper framework—Kundera, and so on. HBase clients are discussed in detail in Chapter 6, HBase Clients. HBase also offers advanced Java-based APIs for playing with tables and column families. (HBase shell is a wrapper around this Java API.) This API also supports metadata management, for example, data compression for column family, region split, and so on. In addition to schema definition, the API also provides an interface for a table scan with various functions such as limiting the number of columns returned or limiting the number of versions of each cell to be stored. For data manipulation, the HBase API supports create, read, update, and delete operations on individual rows. This API comes with many advanced features, which will be discussed throughout this book.

In most parts of the book, all of the sample code or full examples will be using Java-based HBase API only. There are many other options layered on top of the Java API to access HBase, that is, ORM—Kundera, REST gateway, Phoenix, and so on. These clients are covered in Chapter 6, HBase Clients, in detail.

Establishing a connection

Before performing any kind of operation in an HBase table using Java-based HBase API, a connection needs to be established with the help of the HConnection class. This class is managed by the shared HConnectionManager class. Once the connection is established, it returns an HTable instance located in the org.apache.hadoop.hbase.client package. This class provides the user with all the functionality needed to store and retrieve data:

```java
HTableInterface usersTable = new HTable("Costumers");
```

From the preceding code, we can verify that the usage of the HConnection and HConnectionManager classes is not mandatory as the HTable constructor reads the default configuration to create a connection. If there is a need to use or define the connection explicitly, the following code can be used with a custom configuration:

```java
Configuration newConfig = new Configuration(defaultConfig);
HConnection connection =
    HConnectionManager.createConnection(newConfig);
HTableInterface table = connection.getTable("Costumers");
```
Defining the Schema

The HTable class is not thread-safe as concurrent modifications are not safe. Hence, a single instance of HTable for each thread should be used in any application. For multiple HTable instances with the same configuration reference, the same underlying HConnection instance can be used.

Creating HTable instances also comes at a cost. Creating an HTable instance is a slow process as the creation of each HTable instance involves the scanning of the .META table to check whether the table actually exists, which makes the operation very costly. Hence, it is not recommended that you use a new HTable instance for each request where the number of concurrent requests are very high.

Whenever there is a requirement of multiple instances of HTable, consider using the HTablePool class. The following constructor can be used to create an instance of HTablePool:

```java
HTablePool()
HTablePool(Configuration config, int maxSize)
HTablePool(Configuration config, int maxSize,
HTableInterfaceFactory tableFactory)
```

The HTablePool class creates a pool with the HTable class instances with the configuration object, and setting the maxSize parameter defining the HTable instances to count a pool. Here is the code for getting the HTable instance from HTablePool:

```java
HTableInterface usersTable = pool.getTable("Costumers");
```

CRUD operations

A rowkey primarily represents each row uniquely in the HBase table, whereas other keys such as column family, timestamp, and so on are used to locate a piece of data in an HBase table. The HBase API provides the following methods to support the CRUD operations:

- Put
- Get
- Delete
- Scan
- Increment

Let's discuss the first three methods in detail and the rest will be covered in the coming chapters.

For More Information:
Writing data

In HBase, when a write request is received, by default, the data is synchronously written into HLog, also referred to as the write ahead log or commit log and to the memstore. Writing data at two places ensures data durability. The memstore is a write buffer that accumulates the data before committing it to the disk permanently in the form of an HFile. Every time the memstore flushes the data to the disk, a new HFile is created. In case of an HBase cluster failure, data that is not committed to an HFile from the memstore, the write buffer is recovered from the HLog file present in the filesystem. This way of writing to HBase is applicable to both row creation and updation.

A Put class instance is used to store data in an HBase table. For storing data in a table, create a Put instance with rowkey using any of the constructors, as follows:

```java
Put(byte[] rowkey)
Put(byte[] rowArray, int rowOffset, int rowLength)
Put(byte[] rowkey, long ts)
Put(byte[] rowArray, int rowOffset, int rowLength, long ts)
Put p = new Put (Bytes.toBytes("John"));
```

HBase stores all the data, including the rowkey, in the form of a byte array and a Java utility class, bytes define various static utility methods for converting Java data types to and from a byte.

Once a Put instance is created using the rowkey component, the next step is to add the data by using either of the following method definitions:

```java
add(byte[] family, byte[] qualifier, byte[] value)
add(byte[] family, byte[] qualifier, long ts, byte[] value)
add (byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)
add (Cell kv)
```

The add() option takes a column family along with an optional timestamp or one single cell as a parameter. In case the timestamp is not defined, the region server sets it for the data inserted. Here is a complete example of how to write data to HBase:

```java
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
```

For More Information:

import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;

public class SinglePutEx {
    public static void main(String[] args) throws IOException {
        // Get instance of Default Configuration
        Configuration conf = HBaseConfiguration.create();

        // Get table instance
        HTable table = new HTable(conf, "tab1");

        // Create Put with rowkey
        Put put = new Put(Bytes.toBytes("row-1"));

        // Add a column with value "Hello", in "cf1:greet", to the // Put.
        put.add(Bytes.toBytes("cf1"), Bytes.toBytes("greet"), Bytes.toBytes("Hello"));

        // Add more column with value "John", in "cf1:person",
        // to the Put.
        put.add(Bytes.toBytes("cf1"), Bytes.toBytes("person"), Bytes.toBytes("John"));

        table.put(put);
        table.close();
    }
}

Data can be inserted into HBase for a single row or as a batch representing multiple rows using the following method of the HTable class:

void put(List<Put> puts)

A list of Put instances can be created, as follows:

List<Put> puts = new ArrayList<Put>();
Put put1 = new Put(Bytes.toBytes("row-1"));
put1.add(Bytes.toBytes("cf1"), Bytes.toBytes("greet"), Bytes.toBytes("Hello"));
puts.add(put1);
...
A few other important methods of the `Put` class can be found in the following table:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get(byte[] family, byte[] qualifier)</code></td>
<td>This returns a list of all the key-value objects with a matching column family and a qualifier</td>
</tr>
<tr>
<td><code>has(byte[] family, byte[] qualifier, byte[] value)</code></td>
<td>This is a convenience method that determines whether the object's family map contains a value assigned to the given family, qualifier, and timestamp</td>
</tr>
</tbody>
</table>

### Reading data

HBase uses an LRU cache for reads, which is also called the block cache. This block cache keeps the frequently accessed data from the HFiles in the memory to avoid frequent disk reads, and every column family has its own block cache. Whenever a read request arrives, the block cache is first checked for the relevant row. If it is not found, the HFiles on the disk are then checked for the same. Similar to the `Put` class, the `Get` class instance is used to read the data back from the HBase table. The HBase table defines the following method for reading the data and takes the `Get` class instance as an argument:

```java
Result get(Get getInst)
```

This method extracts certain cells from a given row. Here, the `Get` class instance can be created by either of the class constructors:

```java
Get(byte[] rowkey)
```

This constructor creates a `Get` operation for the specified row identified by the rowkey. For narrowing down the data search to a specific cell, additional methods are provided in the following table:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addFamily(byte[] family)</code></td>
<td>Get all columns from the specified family</td>
</tr>
<tr>
<td><code>addColumn(byte[] family, byte[] qualifier)</code></td>
<td>Get the column from the specific family with the specified qualifier</td>
</tr>
<tr>
<td><code>setTimeRange(long minStamp, long maxStamp)</code></td>
<td>Get versions of columns only within the specified timestamp range (minStamp, maxStamp)</td>
</tr>
<tr>
<td><code>setTimeStamp(long timestamp)</code></td>
<td>Get versions of columns with the specified timestamp</td>
</tr>
<tr>
<td><code>setMaxVersions(int max versions)</code></td>
<td>Get up to the specified number of versions of each column. The default value of the maximum version returned is 1 which is the latest cell value.</td>
</tr>
</tbody>
</table>

For More Information:

The following is a complete example of how to read data to HBase:

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

public class SingleGetEx {
    public static void main(String[] args) throws IOException {
        // Get instance of Default Configuration
        Configuration conf = HBaseConfiguration.create();
        // Get table instance
        HTable table = new HTable(conf, "tab1");
        // Create Get with rowkey
        Get get = new Get(Bytes.toBytes("row-1"));
        // Add a column with value "Hello", in "cf1:greet", to the // Put.
        get.addColumn(Bytes.toBytes("cf1"), Bytes.toBytes("greet"));
        Result result = table.get(get);
        byte[] val = result.getValue(Bytes.toBytes("cf1"), Bytes.toBytes("greet"));
        System.out.println("Cell Value: " + Bytes.toString(val));
        table.close();
    }
}
```

Data reading in HBase can take place for a single row or in the form of a batch representing multiple rows using the following method of the HTable class:

```java
Results[] get(List<Get> gets)
```

Here, List can be defined as follows:

```java
List<Get> gets = new ArrayList<Get>();
Get get1 = new Get(Bytes.toBytes("row-1"));
get1.add(Bytes.toBytes("cf1"), Bytes.toBytes("greet"));
gets.add(get1);
...```
Some of the other important methods defined in the Get class are stated in the following table:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>getFamilyMap()</td>
<td>Method for retrieving the get method's family map</td>
</tr>
<tr>
<td>getMaxResultsPerColumnFamily()</td>
<td>Method for retrieving the get method's maximum number of values to return per column family</td>
</tr>
<tr>
<td>getCacheBlocks()</td>
<td>Gets whether blocks should be cached for this method</td>
</tr>
</tbody>
</table>

**Updating data**

Data updation in HBase is done in a manner that is similar to writing it. The new data is updated in the table using a Put instance. The following is the sample code for updating data in HBase:

```java
// Get instance of Default Configuration
Configuration conf = HBaseConfiguration.create();

// Get table instance
HTable table = new HTable(conf, "tab1");

// Create Put with rowkey
Put put = new Put(Bytes.toBytes("row-1"));

// Update a column with value "Hello", in "cf1:greet", to the Put.
put.add(Bytes.toBytes("cf1"), Bytes.toBytes("greet"), Bytes.toBytes("GoodMorning"));

// Update more column with value "David", in "cf1:person", to the Put.
put.add(Bytes.toBytes("cf1"), Bytes.toBytes("person"), Bytes.toBytes("David"));

table.put(put);
table.close();
```

For More Information:  
Defining the Schema

Deleting data
The Delete command only marks the cell for deletion rather than deleting the data immediately. The actual deletion is performed when the compaction of HFiles is done to reconcile these marked records and to free the space occupied by the deleted data.

Compaction is the process of choosing HFiles from a region and combining them. In a major compaction process, it picks all the HFiles and writes back the key-values to the output HFile that are not marked as deleted. Whereas, in a minor compaction, it only takes a few files placed together and combines them into one. Also, minor compaction does not filter the deleted files. The compaction process takes care of the versions and uses the ExploringCompactionPolicy algorithms internally.

Similar to the Put and Get classes, the Delete class instance is used to delete the data from the HBase table. The HBase table defines the following method for deleting the data, which takes the Delete class instance as an argument:

```
void delete(Delete deleteInst)
```

This method deletes the latest cells from a given row. Here, the Delete class instance can be created using either of the class constructors:

```
Delete(byte[] row)
Delete(byte[] rowArray, int rowOffset, int rowLength)
Delete(byte[] rowArray, int rowOffset, int rowLength, long ts)
Delete(byte[] row, long timestamp)
Delete(Delete d)
```

This constructor creates a Delete operation for the specified row identified by the rowkey. For narrowing down the data search to a specific cell, additional methods provided within the Delete class are as follows:

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteColumn(byte[] family, byte[] qualifier)</td>
<td>This deletes the latest version of the specified column based on the timestamp</td>
</tr>
<tr>
<td>deleteColumn(byte[] family, byte[] qualifier, long timestamp)</td>
<td></td>
</tr>
<tr>
<td>deleteColumns(byte[] family, byte[] qualifier)</td>
<td>This deletes all the versions of the specified column</td>
</tr>
<tr>
<td>deleteFamily(byte[] family)</td>
<td>This deletes all the versions of all columns of the specified family</td>
</tr>
</tbody>
</table>

For More Information:
Here is a complete code example for deleting data from HBase:

```java
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;

public class SingleGetEx {
    public static void main(String[] args) throws IOException {
        // Get instance of Default Configuration
        Configuration conf = HBaseConfiguration.create();

        // Get table instance
        HTable table = new HTable(conf, "tab1");

        // Create Delete with rowkey
        Delete delete = new Delete(Bytes.toBytes("row-1"));

        // Add a column with value "Hello", in "cf1:greet", to the Put.
        delete.deleteColumns(Bytes.toBytes("cf1"), Bytes.toBytes("greet"));

        table.delete(delete);
        table.close();
    }
}
```

Data deletion in HBase can happen for a single row or in the form of a batch representing multiple rows using the following method of the HTable class:

```java
void delete(List<Delete> deletes)
```

<table>
<thead>
<tr>
<th>Method name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteFamily(byte[] family, long timestamp)</td>
<td>This deletes all the columns of the specified family with a timestamp less than or equal to the specified timestamp</td>
</tr>
<tr>
<td>deleteFamilyVersion(byte[] family, long timestamp)</td>
<td>This deletes all the columns of the specified family with a timestamp equal to the specified timestamp</td>
</tr>
</tbody>
</table>

For More Information:
Defining the Schema

Here, List can be defined as follows:

```java
List<Delete> deletes = new ArrayList<Delete>();
Delete delete1 = new Delete(Bytes.toBytes("row-1"));
delete1.deleteColumn(Bytes.toBytes("cf1"), Bytes.toBytes("greet"));
deletes.add(delete1);
```

Another important method defined in the Get class is given as follows:

<table>
<thead>
<tr>
<th>Methods name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>setTimestamp(long timestamp)</td>
<td>This sets the timestamp of delete</td>
</tr>
</tbody>
</table>

As discussed earlier, HBase maintains versions of data for each cell; this principle is applicable to all the CRUD operations. That is, `deleteColumn(…)` deletes the specific version based on parameters, and `deleteColumns(…)` deletes all the versions for a specified cell. Similarly, the data reading process reads the version of data based on the parameter values provided.

Summary

In this chapter, we learned the basics of modeling data and some strategies to consider when designing a table in HBase. We also learned how to perform basic CRUD operations on the table created using various APIs provided by HBase. In the next chapter, we will look into HBase table keys, table scan, and some other advanced features such as filters.

For More Information:

Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

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