Apache HBase Primer

Deepak Vohra

Apress®

Apache HBase Primer

Deepak Vohra

Apress®

Apache HBase Primer

Deepak Vohra
White Rock, British Columbia
Canada

ISBN-13 (pbk): 978-1-4842-2423-6 ISBN-13 (electronic): 978-1-4842-2424-3
DOI 10.1007/978-1-4842-2424-3

Library of Congress Control Number: 2016959189

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

Contents at a Glance

Contents

About the Author

Deepak Vohra is a consultant and a principal member of the NuBean software company. Deepak is a Sun-certified Java programmer and Web component developer. He has worked in the fields of XML, Java programming, and Java EE for over seven years. Deepak is the coauthor of *Pro XML Development with Java Technology* (Apress, 2006). Deepak is also the author of the *JDBC 4.0* and *Oracle JDeveloper for J2EE Development, Processing XML Documents with Oracle JDeveloper 11g, EJB 3.0 Database Persistence with Oracle Fusion Middleware 11g*, and *Java EE Development in Eclipse IDE* (Packt Publishing). He also served as the technical reviewer on *WebLogic: The Definitive Guide* (O'Reilly Media, 2004) and *Ruby Programming for the Absolute Beginner (*Cengage Learning PTR, 2007).

About the Technical Reviewer

Massimo Nardone has more than 22 years of experience in security, web/mobile development, and cloud and IT architecture. His true IT passions are security and Android. He has been programming and teaching how to program with Android, Perl, PHP, Java, VB, Python, C/C++, and MySQL for more than 20 years. Technical skills include security, Android, cloud, Java, MySQL, Drupal, Cobol, Perl, web and mobile development, MongoDB, D3, Joomla, Couchbase, C/C++, WebGL, Python, Pro Rails, Django CMS, Jekyll, Scratch, etc.

He currently works as Chief Information Security Office (CISO) for Cargotec Oyj. He holds four international patents (PKI, SIP, SAML, and Proxy areas). He worked as a visiting lecturer and supervisor for exercises at the Networking Laboratory of the Helsinki University of Technology (Aalto University). He has also worked as a Project Manager, Software Engineer, Research Engineer, Chief Security Architect, Information Security Manager, PCI/SCADA Auditor, and Senior Lead IT Security/Cloud/SCADA Architect for many years. He holds a Master of Science degree in Computing Science from the University of Salerno, Italy.

Massimo has reviewed more than 40 IT books for different publishing companies, and he is the coauthor of *Pro Android Games* (Apress, 2015).

Introduction

Apache HBase is an open source NoSQL database based on the wide-column data store model. HBase was initially released in 2008. While many NoSQL databases are available, Apache HBase is the database for the Apache Hadoop ecosystem.

HBase supports most of the commonly used programming languages such as C, C++, PHP, and Java. The implementation language of HBase is Java. HBase provides access support with Java API, RESTful HTTP API, and Thrift.

Some of the other Apache HBase books have a practical orientation and do not discuss HBase concepts in much detail. In this primer level book, I shall discuss Apache HBase concepts. For practical use of Apache HBase, refer another Apress book: *Practical Hadoop Ecosystem*.

PART I

Core Concepts

CHAPTER 1

Fundamental Characteristics

Apache HBase is the Hadoop database. HBase is open source and its fundamental characteristics are that it is a non-relational, column-oriented, distributed, scalable, big data store. HBase provides schema flexibility. The fundamental characteristics of Apache HBase are as follows.

Distributed

HBase provides two distributed modes. In the *pseudo-distributed* mode, all HBase daemons run on a single node. In the *fully-distributed* mode, the daemons run on multiple nodes across a cluster. Pseudo-distributed mode can run against a local file system or an instance of the Hadoop Distributed File System (HDFS). When run against local file system, durability is not guaranteed. Edits are lost if files are not properly closed. The fully-distributed mode can only run on HDFS. Pseudo-distributed mode is suitable for small-scale testing while fully-distributed mode is suitable for production. Running against HDFS preserves all writes.

HBase supports auto-sharding, which implies that tables are dynamically split and distributed by the database when they become too large.

Big Data Store

HBase is based on Hadoop and HDFS, and it provides low latency, random, real-time, read/write access to big data. HBase supports hosting very large tables with billions of rows and billions/millions of columns. HBase can handle petabytes of data. HBase is designed for queries of massive data sets and is optimized for read performance. Random read access is not a Apache Hadoop feature as with Hadoop the reader can only run batch processing, which implies that the data is accessed only in a sequential way so that it has to search the entire dataset for any jobs needed to perform.

Non-Relational

HBase is a NoSQL database. NoSQL databases are not based on the relational database model. Relational databases such as Oracle database, MySQL database, and DB2 database store data in tables, which have relations between them and make use of

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_1

SQL (Structured Query Language) to access and query the tables. NoSQL databases, in contrast, make use of a storage-and-query mechanism that is predominantly based on a non-relational, non-SQL data model. The data storage model used by NoSQL databases is not some fixed data model; it is a flexible schema data model. The common feature among the NoSQL databases is that the relational and tabular database model of SQL-based databases is not used. Most NoSQL databases make use of no SQL at all, but NoSQL does not imply absolutely no SQL is used, because of which NoSQL is also termed as "not only SQL."

Flexible Data Model

In 2006 the Google Labs team published a paper entitled "BigTable: A Distributed Storage System for Structured Data" (http://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf). Apache HBase is a wide-column data store based on Apache Hadoop and on BigTable concepts. The basic unit of storage in HBase is a *table*. A table consists of one or more *column families*, which further consists of *columns*. Columns are grouped into column families. Data is stored in *rows*. A row is a collection of key/value pairs. Each row is uniquely identified by a row key. The row keys are created when table data is added and the row keys are used to determine the sort order and for data sharding, which is splitting a large table and distributing data across the cluster.

HBase provides a flexible schema model in which columns may be added to a table column family as required without predefining the columns. Only the table and column family/ies are required to be defined in advance. No two rows in a table are required to have the same column/s. All columns in a column family are stored in close proximity.

HBase does not support transactions. HBase is not eventually consistent but is a *strongly consistent* at the record level. Strong consistency implies that the latest data is always served but at the cost of increased latency. In contrast, eventual consistency can return out-of-date data.

HBase does not have the notion of data types, but all data is stored as an array of bytes.

Rows in a table are sorted lexicographically by row key, a design feature that makes it feasible to store related rows (or rows that will be read together) together for optimized scan.

Scalable

The basic unit of horizontal scalability in HBase is a *region*. Rows are shared by regions. A region is a sorted set consisting of a range of adjacent rows stored together. A table's data can be stored in one or more regions. When a region becomes too large, it splits into two at the middle row key into approximately two equal regions. For example, in Figure 1-1, the region has 12 rows, and it splits into two regions with 6 rows each.

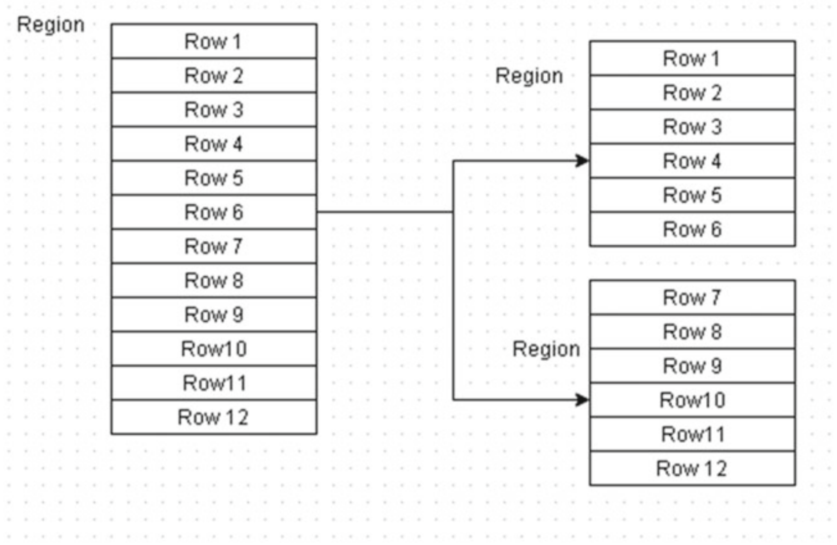


Figure 1-1. *Region split example*

For balancing, data associated with a region can be stored on different nodes in a cluster.

Roles in Hadoop Big Data Ecosystem

HBase is run against HDFS in the fully-distributed mode; it also has the same option available in the pseudo-distributed mode. HBase stores data in StoreFiles on HDFS. HBase does not make use of the MapReduce framework of Hadoop but could serve as the source and/or destination of MapReduce jobs.

Just as Hadoop, HBase is designed to run on commodity hardware with tolerance for individual node failure. HBase is designed for batch processing systems optimized for streamed access to large data sets. While HBase supports random read access, HBase is not designed to optimize random access. HBase incurs a random read latency which may be reduced by enabling the Block Cache and increasing the Heap size.

HBase can be used for real-time analytics in conjunction with MapReduce and other frameworks in the Hadoop ecosystem, such as Apache Hive.

How Is Apache HBase Different from a Traditional RDBMS?

HBase stores data in a table, which has rows and column just as a RDBMS does, but the databases are mostly different, other than the similar terminology. Table 1-1 shows the salient differences.

***Table 1-1.** Salient Differences Between HBase and RDBMS*

Feature	RDBMS	HBase
Schema	Fixed schema	Flexible schema
Data volume	Small to medium (few thousand or a million rows). TB of data.	Big Data (hundreds of millions or billions rows). PB of data.
Primary query language	SQL	Get, Put, Scan shell commands.
Data types	Typed columns	No data types
Relational integrity	Supported	Not supported
Joins	Supports joins, such as equi-joins and outer-joins	Does not provide built-in support for joins. Joins using MapReduce.
Data structure	Highly structured and statically structured. Fixed data model.	Un-structured, semi-structured, and structured. Flexible data model.
Transactions	Supported	Not supported
Advanced query language	Supported	Not supported
Indexes	Primary, secondary, B-Tree, Clustered	Secondary indexes
Consistency	Strongly consistent. CAP theorem consistency.	Strongly consistent. ACID consistency.
Scalability	Provides some level of scalability with vertical scaling in which additional capacity may be added to a server	Highly scalable with horizontal scalability in which additional servers may be added. Scales linearly.
Distributed	Distributed to some extent	Highly distributed
Real-time queries	Supported	Supported
Triggers	Supported	Provides RDBMS-like triggers through coprocessors
Hardware	Specialized hardware and less hardware	Commodity hardware and more hardware
Stored procedures	Supported	Provides stored procedures through coprocessors
Java	Does not require Java	Requires Java
High Availability	Supported	Ultra-high availability

(*continued*)

Table 1-1. (*continued*)

Feature	RDBMS	HBase
Fault tolerant	Fault tolerant to some extent	Highly fault tolerant
Normalization	Required for large data sets	Not required
Object-oriented programming model	Because of the complexity of aggregating data, using JOINS using an object-oriented programming model is not suitable.	The key-value storage makes HBase suitable for object-oriented programming model. Supported with client APIs in object-oriented languages such as PHP, Ruby, and Java.
Administration	More administration	Less administration with auto-sharding, scaling, and rebalancing
Architecture	Monolithic	Distributed
Sharding	Limited support. Manual server sharding. Table partitioning.	Auto-sharding
Write performance	Does not scale well	Scales linearly
Single point of failure	With single point of failure	Without single point of failure
Replication	Asynchronous	Asynchronous
Storage model	Tablespaces	StoreFiles (HFiles) in HDFS
Compression	Built-in with some RDBMS in storage engine/table/index. Various methods available to other RDBMS.	Built-in Gzip compression
Caching	Standard data/metadata cache with query cache	In-memory caching
Primary data object	Table	Table
Read/write throughput	1000s of queries per second	Millions of queries per second
Security	Authentication/Authorization	Authentication/Authorization
Row-/Column- Oriented	Row-oriented	Column-oriented
Sparse tables	Suitable for sparse tables	Not optimized for sparse tables
Wide/narrow tables	Narrow tables	Wide tables
MapReduce integration	Not designed for MR integration	Well integrated

Summary

In this chapter, I discussed the fundamental characteristics of big data store Apache HBase, which are its distributedness, scalability, non-relational, and flexible model. I also discussed HBase's role in the Hadoop big data ecosystem and how is HBase different from traditional RDBMS. In the next chapter, I will discuss how HBase stores its data in a HDFS.

CHAPTER 2

Apache HBase and HDFS

Apache HBase runs on HDFS as the underlying filesysystem and benefits from HDFS features such as data reliability, scalability, and durability. HBase stores data as `Store Files` (`HFiles`) on the HDFS Datanodes. `HFile` is the file format for HBase and `org.apache.hadoop.hbase.io.hfile.HFile` is a Java class. HFile is an HBase-specific file format that is based on the `TFile` binary file format. A Store File is a lightweight wrapper around the HFile. In addition to storing table data HBase also stores the write-ahead logs (WALs), which store data before it is written to HFiles on HDFS. HBase provides a Java API for client access. HBase itself is a HDFS client and makes use of the Java class `DFSClient`. References to DFSClient appear in the HBase client log messages and HBase logs as HBase makes use of the class to connect to NameNode to get block locations for Datanode blocks and add data to the Datanode blocks. HBase leverages the fault tolerance provided by the Hadoop Distributed File System (HDFS). HBase requires some configuration at the client side (HBase) and the server side (HDFS).

Overview

HBase storage in HDFS has the following characteristics:

- Large files
- A lot of random seeks
- Latency sensitive
- Durability guarantees with sync
- Computation generates local data
- Large number of open files

HBase makes use of three types of files:

- WALs or `HLogs`
- Data files (also known as store files or `HFiles`)
- 0 length files: References (symbolic or logical links)

Each file is replicated three times. Data files are stored in the following format in which `"userTable"` and `"table1"` are example tables, and `"098345asde5t6u"` and `"09jhk7y65"` are regions, and `"family"` and `"f1"` are column families, and `"09ojki8dgf6"` and `"90oifgr56"` are the data files:

```
hdfs://localhost:41020/hbase/userTable/098345asde5t6u/family/09ojki8dgf6
hdfs://localhost:41020/hbase/userTable/09drf6745asde5t6u/family/09ojki8ddfre5
hdfs://localhost:41020/hbase/table1/09jhk7y65/f1/90oifgr56
```

Write-ahead logs are stored in the following format in which "server1" is the region server:

```
hdfs://localhost:41020/hbase/.logs/server1,602045,123456090
hdfs://localhost:41020/hbase/.logs/server1,602045,123456090/server1%2C134r5
```

Links are stored in the following format in which "098eerf6" is a region, "family" is a column family and "8uhy67" is the link:

```
hdfs://localhost:41020/hbase/.hbase_snapshot/usertable_snapshot/098eerf6/
family/8uhy67
```

Datanode failure is handled by HDFS using replication. HBase provides real-time, random, read/write access to data in the HDFS. A data consumer reads data from HBase, which reads data from HDFS, as shown in Figure 2-1. A data producer writes data to HBase, which writes data to HDFS. A data producer may also write data to HDFS directly.

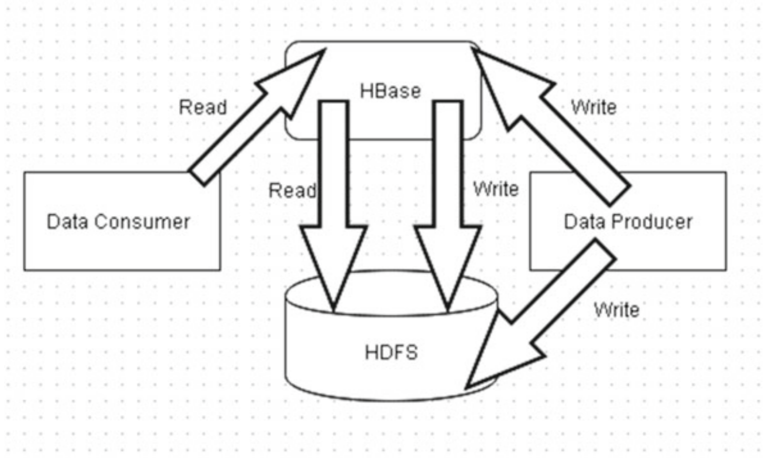


Figure 2-1. *Apache HBase read/write path*

HBase is the most advanced user of HDFS and makes use of the `HFileSystem` (an encapsulation of `FileSystem`) Java API. `HFileSystem` has the provision to use separate filesystem objects for reading and writing WALs and HFiles. The `HFileSystem.create(Path path)` method is used to create the `HFile` data files and `HLog` WALs.

HBase communicates with both the NameNode and the Datanodes using HDFS `Client` classes such as `DFSClient`. HDFS pipelines communications, including data writes, from one Datanode to another as shown in Figure 2-2. HBase write errors, including communication issues between Datanodes, are logged to the HDFS logs and not the HBase logs. Whenever feasible HBase writes are local, which implies that the writes are to a local Datanode. As a result, Region Servers should not get too many write errors. Errors may be logged in both HDFS and HBase but if a Datanode is not able to replicate data blocks, errors are written to HDFS logs and not the HBase logs.

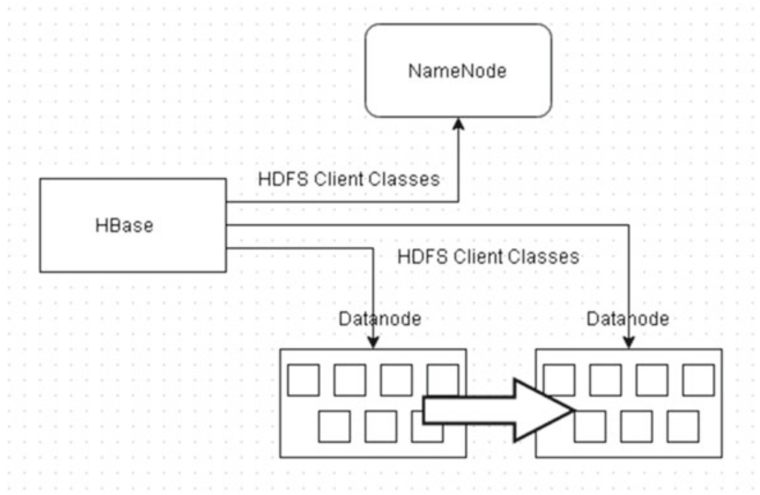


Figure 2-2. *Apache HBase communication with NameNode and Datanode*

The `hdfs-site.xml` configuration file contains information like the following:

- The value of replication factor
- NameNode path
- Datanode path of the local file systems where you want to store the Hadoop infrastructure

A HBase client communicates with the ZooKeeper and the *HRegionServers*. The *HMaster* coordinates the RegionServers. The RegionServers run on the Datanodes. When HBase starts up, the HMaster assigns the regions to each RegionServer, including the regions for the `-ROOT-` and `.META.` (`hbase:meta` catalog table in later version) tables. The HBase table structure comprises of Column Families (or a single Column Family), that group similar column data. The basic element of distribution for an HBase table is a Region, which is further comprised of a Store per column family. A Region is a subset of a table's data. A set of regions is served by a RegionServer and each region is served by only one RegionServer. A Store has an in-memory component called the `MemStore` and a persistent storage component called an HFile or StoreFile. The `DFSClient` is used to store and replicate the HFiles and HLogs in the HDFS datanodes. The storage architecture of HBase is shown in Figure 2-3.

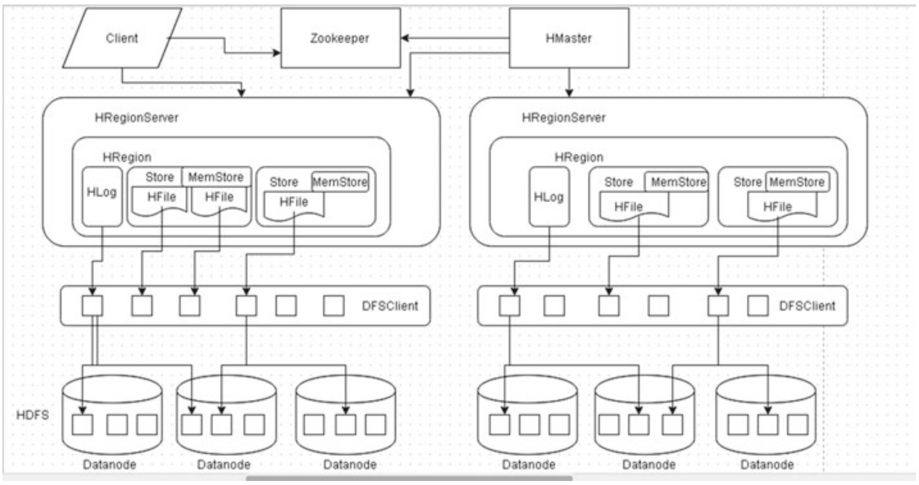


Figure 2-3. *Apache HBase storage architecture*

As shown in Figure 2-3, the HFiles and HLogs are primarily handled by the HRegions but sometimes even the HRegionServers have to perform low-level operations. The following flow sequence is used when a new client contacts the ZooKeeper quorum to find a row key:

1. First, the client gets the server name that hosts the `-ROOT-` region from the ZooKeeper. In later versions, the `-ROOT-` region is not used and the `hbase:meta` table location is stored directly on the ZooKeeper.

2. Next, the client queries the server to get the server that hosts the `.META.` tables. The `-ROOT-` and `.META.` server info is cached and looked up once only.

3. Subsequently, the client queries the `.META.` server to get the server that has the row the client is trying to get.

4. The client caches the information about the HRegion in which the row is located.

5. The client contacts the HRegionServer hosting the region directly. The client does not need to contact the `.META.` server again and again once it finds and caches information about the location of the row/s.

6. The HRegionServer opens the region and creates a HRegion object. A `store` instance is created for each `HColumnFamily` for each table. Each of the store instances can have `StoreFile` instances, which are lightweight wrappers around the HFile storage files. Each HRegion has a `MemStore` and a `HLog` instance.

HBase communicates with HDFS on two different ports:

1. Port 50010 using the `ipc.Client` interface. It is configured in HDFS using the `dfs.datanode.ipc.address` configuration parameter.
2. Port 50020 using the `DataNode` class. It is configured in HDFS using the `dfs.datanode.address` configuration parameter.

Being just another HDFS client, the configuration settings for HDFS also apply to HBase. Table 2-1 shows how the HDFS configuration settings pertain to retries and timeouts.

Table 2-1. *HDFS Configuration Settings*

Configuration setting	Description	Configuration file
`ipc.client.connect.max.retries`	Number of tries a HBase client will make to establish a server connection. Default is 10. For SASL connections, the setting is hard coded at 15 and cannot be reconfigured.	`core-default.xml`
`ipc.client.connect.max.retries.on.timeouts`	Number of tries an HBase client will make on socket timeout to establish a server connection. Default is 45.	`core-default.xml`
`dfs.client.block.write.retries`	Number of retries HBase makes to write to a datanode before signaling a failure. Default is 3. After hitting the failure threshold, the HBase client reconnects with the NameNode to get block locations for another datanode.	`hdfs-default.xml`
`dfs.client.socket-timeout`	The time before which an HBase client trying to establish socket connection or reading times out. Default is 60 secs.	`hdfs-default.xml`
`dfs.datanode.socket.write.timeout`	The time before which a write operation times out. Default is 8 minutes.	`hdfs-default.xml`

Storing Data

Data is written to the actual storage in the following sequence:

1. The client sends an `HTable.put(Put)` request to the HRegionServer. The `org.apache.hadoop.hbase.client.HTable` class is used to communicate with a single HBase table. The class provides the `put(List<Put> puts)` and the `put(Put put)` methods to put some data in a table. The `org.apache.hadoop.hbase.client.Put` class is used to perform `Put` operations for a single row.

2. The HRegionServer forwards the request to the matching HRegion.

3. Next, it is ascertained if data is to be written to the WAL, which is represented with the `HLog` class. Each RegionServer has a `HLog` object. The `Put` class method `setWriteToWAL(boolean write)`, which is deprecated in HBase 0.94, or the `setDurability(Durability d)` method is used to set if data is to be written to WAL. Durability is an `enum` with values listed in Table 2-2; `Put` implements the `Mutation` abstract class, which is extended by classes implementing put, delete, and append operations.

Table 2-2. *Durability Enums*

Durability Enum	Description
`ASYNC_WAL`	Write the Mutation to the WAL asynchronously as soon as possible.
`FSYNC_WAL`	Write the Mutation to the WAL synchronously and force the entries to disk.
`SKIP_WAL`	Do not write the Mutation to the WAL.
`SYNC_WAL`	Write the Mutation to the WAL synchronously.
`USE_DEFAULT`	Use the column family's default setting to determine durability.

The WAL is a standard sequence file (`SequenceFile`) instance. If data is to be written to WAL, it is written to the WAL.

1. After the data is written (or not) to the WAL, it is put in the `MemStore`. If the `MemStore` is full, data is flushed to the disk.

2. A separate thread in the HRegionServer writes the data to an HFile in HDFS. A last written sequence number is also saved.

The HFile and HLog files are written in subdirectories of the root directory in HDFS for HBase, the `/hbase` directory. The HLog files are written to the `/hbase/.logs` directory. A sub-directory for each HRegionServer is created within the `.logs` directory. Within the `HLog` file, a log is written for each HRegion.

For the data files (HFiles), a subdirectory for each HBase table is created in the /hbase directory. Within the directory (ies) corresponding to the tables, subdirectories for regions are created. Each region name is encoded to create the subdirectory for the region. Within the region's subdirectories, further subdirectories are created for the column families. Within the directories for the column family, the HFile files are stored. As a result, the directory structure in HDFS for data files is as follows:

```
/hbase/<tablename>/<encoded-regionname>/<column-family>/<filename>
```

The root of the region directory has the .regioninfo file containing metadata for the region. HBase regions split automatically if the HFile data file's storage exceeds the limit set by the hbase.hregion.max.filesize setting in the hbase-site.xml/hbase-default.xml configuration file. The default setting for hbase.hregion.max.filesize is 10GB. When the default storage requirement for a region exceeds the hbase.hregion.max.filesize parameter value, the region splits into two and reference files are created in the new regions. The reference files contain information such as the key at which the region was split. The reference files are used to read the original region data files. When compaction is performed, new data files are created in a new region directory and the reference files are removed. The original data files in the original region are also removed.

Each /hbase/table directory also contains a compaction.dir directory, which is used when splitting and compacting regions.

HFile Data files- HFile v1

Up to version 0.20, HBase used the MapFile format. In HBase 0.20, MapFile is replaced by the HFile format. HFile is made of data blocks, which have the actual data, followed by metadata blocks (optional), a fileinfo block, the data block index, the metadata block index, and a fixed size trailer, which records the offsets at which the HFile changes content type.

```
<data blocks><meta blocks><fileinfo><data index><meta index><trailer>
```

Each block has a "magic" at the start. Blocks are in a key/value format. In data blocks, both the key and the value are a byte array. In metadata blocks, the key is a String and the value is a byte array. An empty file structure is as follows:

```
<fileinfo><trailer>
```

The HFile format is based on the TFile binary file format. The HFile format version 1 is shown in Figure 2-4.

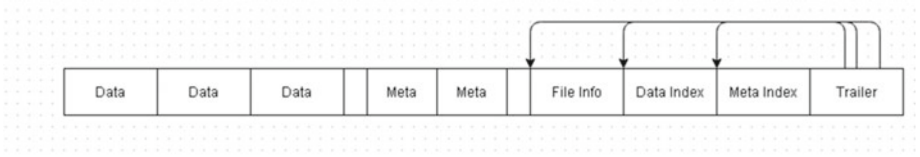


Figure 2-4. *HFile file format*

The different sections of the HFile are discussed in Table 2-3.

Table 2-3. *HFile Sections*

HFile Section	Description
Data	The data blocks in which the HBase table data is stored. The actual data is stored as key/value pairs. It's optional, but most likely an HFile has data. The list of records in a data block are stored in the following format: Key Length: int Value Length: int Key: byte[] Value: byte[]
Meta	The metadata blocks. The meta is the metadata for the data stored in HFile. Optional. Meta blocks are writtern upon file close. RegionServer's StoreFile uses metadata blocks to store a bloom filter, which is used to avoid reading a file if there is no chance that the key is present. A bloom filter just indicates that maybe the key is in the file. The file/s still need to be scanned to find if the key is in the file.
FileInfo	FileInfo is written upon file close. FileInfo is a simple Map with key/value pairs that are both a byte array. RegionServer's StoreFile uses FileInfo to store Max SequenceId, the major compaction key, and timerange info. Max SequenceId is used to avoid reading a file if the file is too old. Timerange is used to avoid reading a file if the file is too new. Meta information about the HFile such as `MAJOR_COMPACTION_KEY`, `MAX_SEQ_ID_KEY`, `hfile.AVG_KEY_LEN`, `hfile.AVG_VALUE_LEN`, `hfile.COMPARATOR`, `hfile.LASTKEY` The format of the FileInfo is as follows: Last Key: byte[] Avg Key Length: int Avg Value Length: int Comparator Class: byte[]
Data Index	Records the offset of the data blocks. The format of the data index is as follows: Block Begin: long Block Size: int Block First Key: byte[]
Meta Index	Records the offset of the meta blocks. The format of the meta index is as follows: Block Begin: long Block Size: int Block First Key: byte[]

(*continued*)

Table 2-3. (*continued*)

HFile Section	Description
Trailer	Has pointers to the other blocks and is written last. The format of the Trailer is as follows: FileInfo Offset: long Data/Meta Index Offset: long Data/Meta Index Count: long Total Bytes: long Entry Count: long Compression Codec: int Version: int

On load, the following sequence is followed:

1. Read the trailer
2. Seek Back to read file info
3. Read the data index
4. Read the meta index

Each data block contains a "magic" header and `KeyValue` pairs of actual data in plain or compressed form, as shown in Figure 2-5.

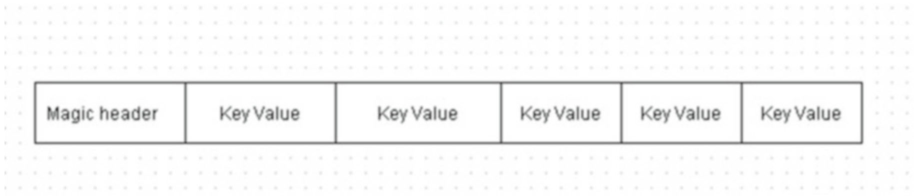


Figure 2-5. *A data block*

HBase Blocks

The default block size for an HFile file is 64KB, which is several times less than the minimum HDFS block size of 64MB. A minimum block size between 8KB and 1MB is recommended for the HFile file. If sequential access is the primary objective, a larger block size is preferred. A larger block size is inefficient for random access because more data has to be decompressed. For random access, small block sizes are more efficient but have the following disadvantages:

1. They require more memory to hold the block index
2. They may be slower to create
3. The smallest feasible block size is limited to 20KB-30KB because the compressor stream must be flushed at the end of each data block

The default minimum block size for HFile may be configured in `hbase-site.xml` with the `hfile.min.blocksize.size` parameter. Blocks are used for different purpose in HDFS and HBase. In HDFS, blocks are the unit of storage on disk. In HBase, blocks are the unit of storage for memory. Many HBase blocks fit into a single HBase file. HBase is designed to maximize efficiency from the HDFS file system and fully utilize the HDFS block size.

Key Value Format

The `org.apache.hadoop.hbase.KeyValue` class represents a HBase key/value. The `KeyValue` format is as follows:

```
<keylength> <valuelength> <key> <value>
```

A `KeyValue` is a low-level byte array structure made of the sections shown in Figure 2-6.

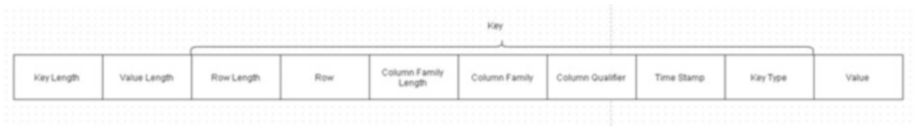


Figure 2-6. *A KeyValue*

The key has the following format:

```
<rowlength> <row> <columnfamilylength> <columnfamily> <columnqualifier>
<timestamp> <keytype>
```

The different sections of a `KeyValue` are discussed in Table 2-4.

Table 2-4. *KeyValue Sections*

KeyValue Section	Description
Key Length	The key length. Using the key length and the value length information, direct access to the value may be made without using the key.
Value Length	The value length. Using the key length and the value length information, direct access to the value may be made without using the key.
Row Length	The row length
Row	The row
Column Family Length	The column family length
Column Family	The column family
Column Qualifier	The column qualifier
Time Stamp	The timestamp
Key Type	The key type
Value	The value

HFile data files are immutable once written. HFiles are generated by flush or compactions (sequential writes). HFiles are read randomly or sequentially. HFiles are big in size with a flush size of tens of GB. Data blocks have a target size represented by `BLOCKSIZE` in the column family descriptor, which is 64KB by default. The target size is an uncompressed and unencoded size. Index blocks (leaf, intermediate, root) also have a target size configured with the `hfile.index.block.max.size` with a default value of 128KB.

Bloom filters may be used to improve read efficiency. Bloom filters may be enabled per column family. Bloom filter blocks have a target size configured with `io.storefile.bloom.block.size` with a default value of 128KB.

HFile v2

To improve performance when large quantities of data are stored, the HFile format has been modified. One of the issues with v1 is that the data and meta indexes and large bloom filters need to be loaded in memory, which slows down the loading process and also uses excessive memory and cache. Starting with HBase 0.92, the *HFile v2* introduces multi-level indexes and a block-level bloom filter for improved speed, memory, and cache usage.

HFile v2 introduces a block-level index as an inline-block. Instead of having a monolithic index and a bloom filter in memory, the index and bloom filter are broken per block, thus reducing the load on the memory. The block-level index is called a *leaf index*. Block-level indexing creates a multi-level index, an index per block. The data block structure is shown in Figure 2-7. The meta and intermediate index blocks are optional.

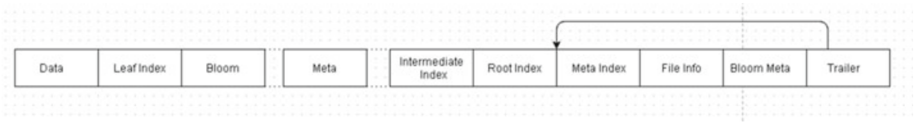


Figure 2-7. *Data block structure*

The last key in each block is kept to create an intermediate index to make the multi-level index B-tree like.

The block header consists of a block type instead of the "magic" in v1. The block type is a description of the block content, such as the data, leaf index, bloom, meta, root index, meta index, file info, bloom meta, and trailer. For fast forward and backward seeks, three new fields have been added for compressed/uncompressed/offset previous block, as shown in Figure 2-8.

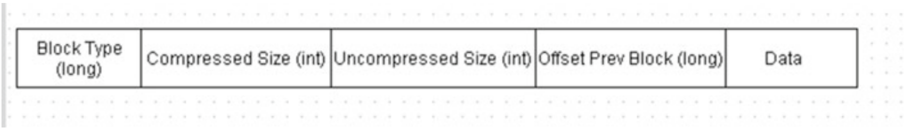


Figure 2-8. *Block header*

Encoding

Data block encoding may be used to improve compression as sorting is used and keys are very similar in the beginning of the keys. Data block encoding also helps by limiting duplication of information in keys by taking advantage of some of the fundamental designs and patterns of HBase: sorted row keys and/or the schema of a given table. The general purpose compression algorithm does not use encoding and the key/value length is stored completely even if a row has a key similar to the preceding key. In HBase 0.94, the *prefix* and *diff* encodings may be chosen. In *prefix* encoding, a new column called `Prefix Length` is added for the common length bytes equal in the previous row. Just the difference from the previous row is stored in each row. The first row has to be stored completely because no previous row exists. The different types of encodings, including the no encoding format, are shown in Figure 2-9.

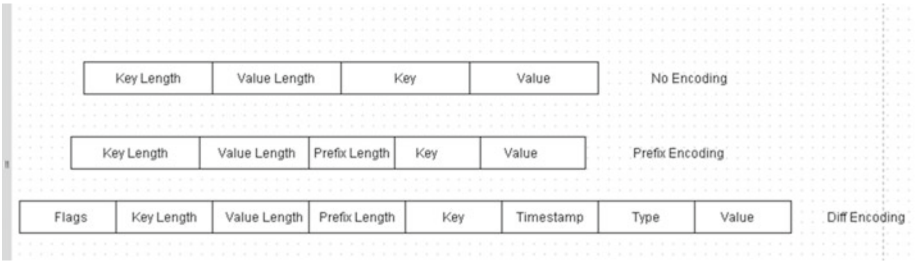


Figure 2-9. *Different types of encodings*

In diff encoding, the key is not considered as a sequence of bytes but the encoder splits each key and compresses each section separately for improved compression. As a result, the column family is stored once only. One byte describes the key layout. Key length, value length, and type may be omitted if equal to the previous row. The timestamp is signed and is stored as a difference from the previous row.

The data block encoding feature is not enabled by default. To enable the feature, `DATA_BLOCK_ENCODING = PREFIX | DIFF | FAST_DIFF` has to be set in the table info.

Compaction

Compaction is the process of creating a larger file by merging smaller files. Compaction can become necessary if HBase has scanned too many files to find a result but is not able to find a result. After the number of files scanned exceeds the limit set in `hbase.hstore.compaction.max`, parameter compaction is performed to merge files to create a larger file. Instead of searching multiple files, only one file has to be searched. Two types of compaction are performed: *minor compaction* and *major compaction*. Minor compaction just merges two or more smaller files into one. Major compaction merges all of the files. In a major compaction, deleted and duplicate key/values are removed. Compaction provides better indexing of data, reducing the number of seeks required to reach a block that could contain the key.

KeyValue Class

The main methods in the `KeyValue` class are discussed in Table 2-5.

Table 2-5. *KeyValue Class Methods*

Method	Description
`getRow()`	Returns the row of the `KeyValue`. To be used on the client side.
`getFamily()`	Returns the column family of the `KeyValue`. To be used on the client side.
`getQualifier()`	Returns the column qualifier of the `KeyValue`. To be used on the client side.
`getTimestamp()`	Returns the timestamp.
`getValue()`	Returns the value of the `KeyValue` as a `byte[]`.To be used on the client side.
`getBuffer()`	Returns the `byte[]` for the `KeyValue`. To be used on the server side.
`getKey()`	Not to be used directly. Used internally for compacting and testing.

When data is added to HBase, the following sequence is used to store the data:

1. The data is first written to a WAL called `HLog`.
2. The data is written to an in-memory `MemStore`.
3. When memory exceeds certain threshold, data is flushed to disk as `HFile` (also called a `StoreFile`).
4. HBase merges smaller HFiles into larger HFiles with a process called compaction.

The HBase architecture in relation to the HDFS is shown in Figure 2-10.

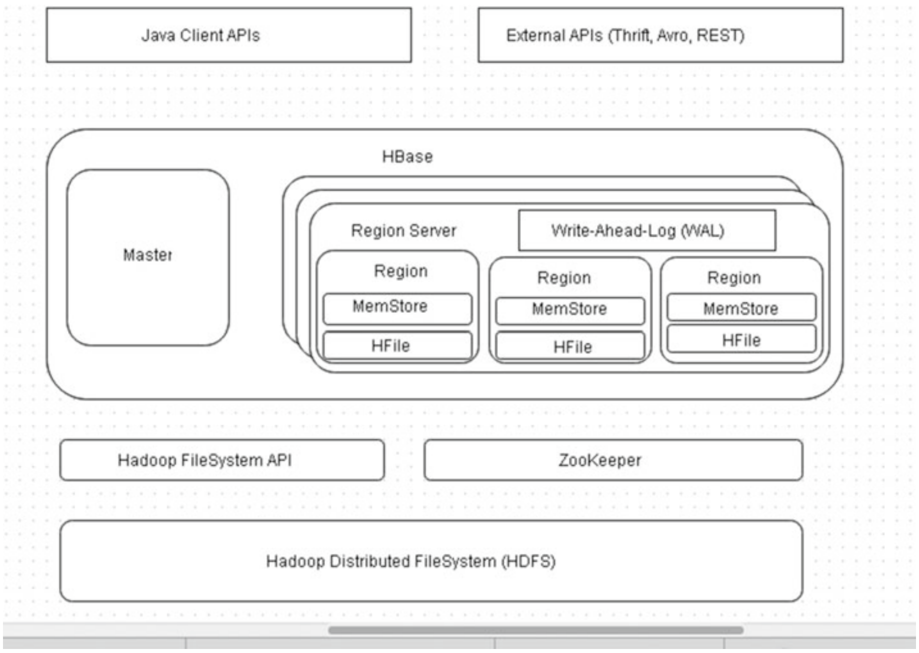


Figure 2-10. *Apache HBase architecture in relation to HDFS*

HBase consists of the following components:

1. Master
2. RegionServers
3. Regions within a RegionServer
4. MemStores and HFiles within a Region

HBase is based on HDFS as the filesystem. The ZooKeeper coordinates the different components of HBase. HBase may be accessed using Java Client APIs, external APIs, and the Hadoop `FileSystem` API. The Master coordinates the RegionServers. A Region is a subset of a table's rows, such as a partition. A RegionServer serves the region's data for reads and writes. The ZooKeeper stores global information about the cluster. The `.META.` tables list all of the regions and their locations. The `-ROOT-` table lists all of the `.META.` tables.

The HBase objects stored in the Datanodes may be browsed from the NameNode web application running at port 50070. The HDFS directory structure for HBase data files is as follows (also shown in Figure 2-11):

```
/hbase/<Table>/<Region>/<ColumnFamily>/<StoreFile>
<Table> is the HBase table.
<Region> is the region.
<ColumnFamily> is the column family.
<StoreFile> is the store file or HFile.
```

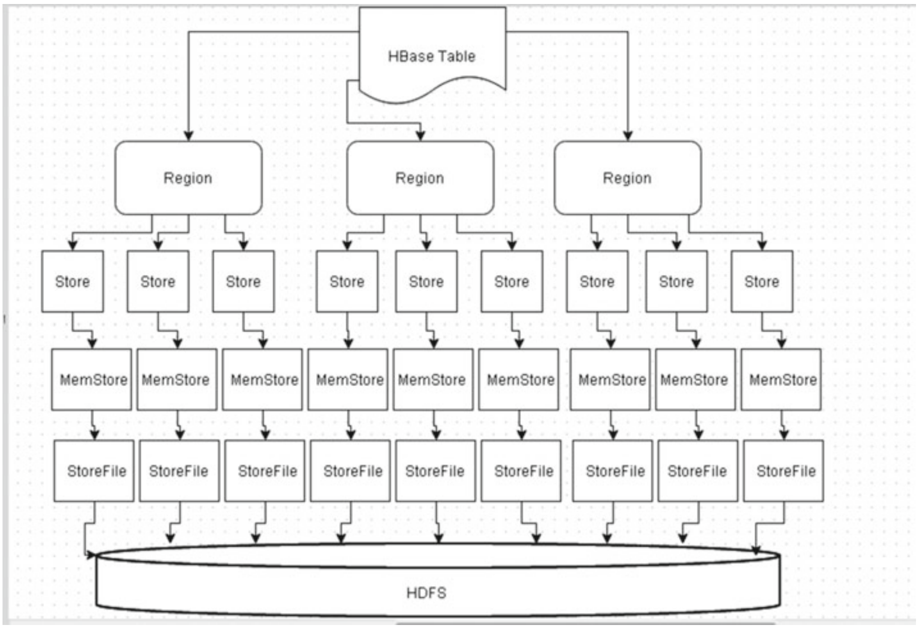


***Figure 2-11.** HDFS directory structure for HBase data files*

The HDFS directory structure for the WAL is as follows:

```
/hbase/.logs/<RegionServer>/<HLog>
```

A StoreFile (HFile) is created every time the MemStore flushes.

As a store corresponds to a column family (CF), the preceding diagram can be redrawn as shown in Figure 2-12.

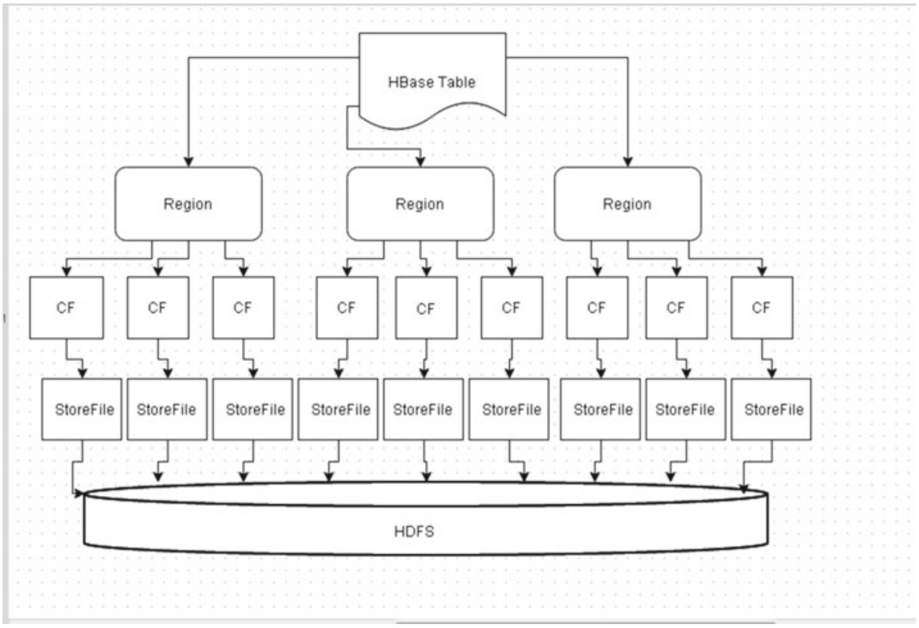


Figure 2-12. *A store is the same as a column family*

Data Locality

For efficient operation, HBase needs data to be available locally, for which it is a best practice to run a HDFS node on each RegionServer. HDFS has to be running on the same cluster as HBase for data locality. Region/RegionServer locality is achieved via HDFS block replication. The following replica placement policy is used by `DFSClient`:

1. The first replica is placed on a local node.
2. The second replica is placed on a different node in the same rack.
3. The third replica is placed on a node in another rack.

The replica placement policy is shown in Figure 2-13.

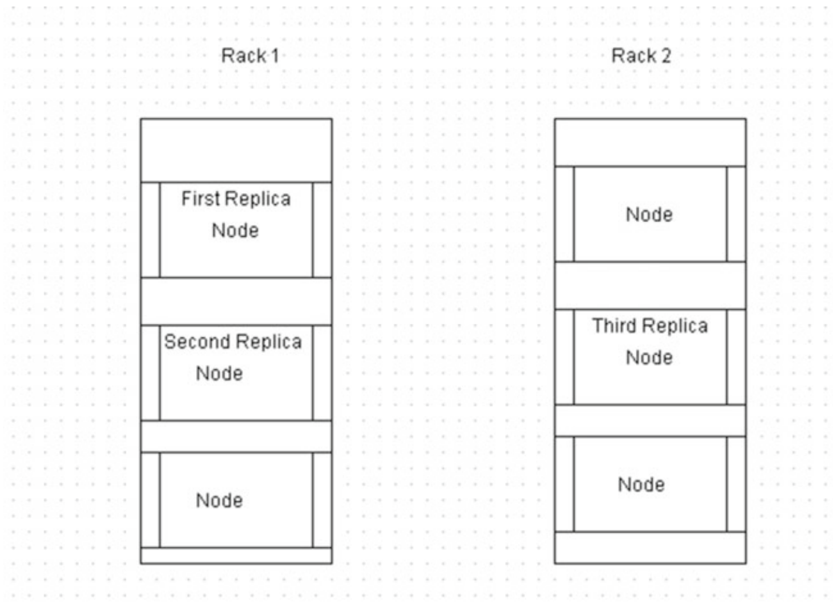


Figure 2-13. *Replica placement policy*

HBase eventually achieves locality for a region after a flush or compaction. In a RegionServer failover, data locality may be lost if a RegionServer is assigned regions with non-local HFiles, resulting in none of the replicas being local. But as new data is written in the region, or the table is compacted and HFiles are rewritten, they will become local to the RegionServer.

Table Format

HBase provides tables with a Key:Column/Value interface with following properties:

- Dynamic columns (qualifiers), no schema required
- “Fixed” column groups (families)
- table[row:family:column]=value

HBase Ecosystem

The HBase ecosystem consists of Apache Hadoop HDFS for data durability and reliability (write-ahead log) and Apache ZooKeeper for distributed coordination, and built-in support for running Apache Hadoop MapReduce jobs, as shown in Figure 2-14.

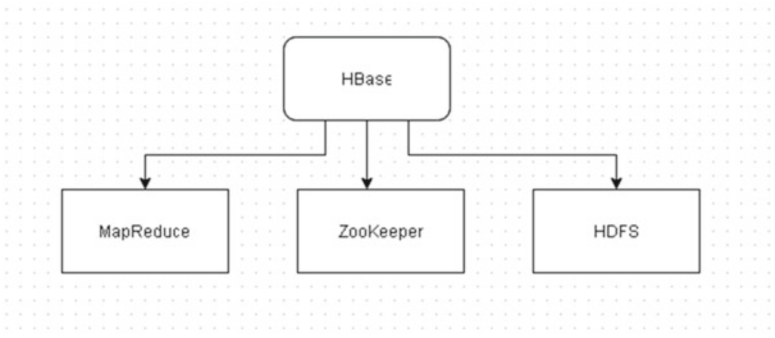


Figure 2-14. *HBase ecosystem*

HBase Services

HBase architecture has two main services: HMaster and HRegionServer, as shown in Figure 2-15.

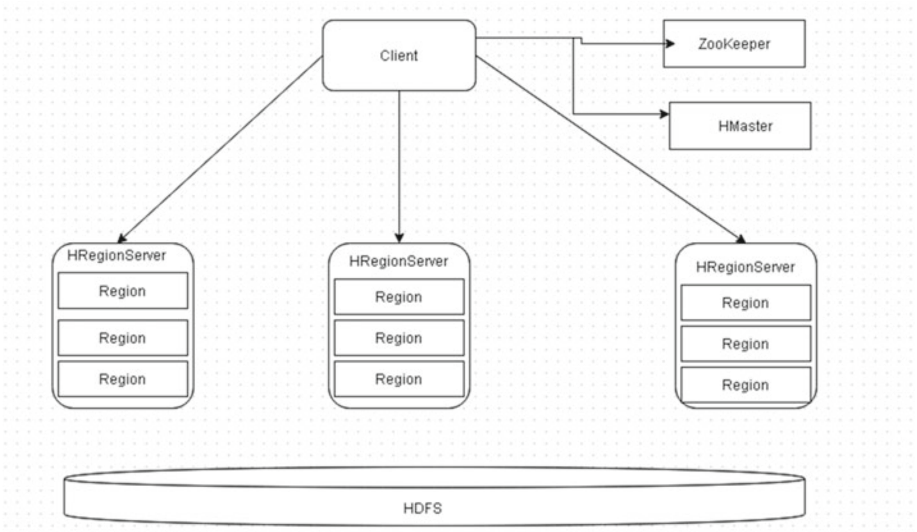


Figure 2-15. *HBase architecture services*

The RegionServer contains a set of Regions and is responsible for handling reads and writes. The region is the basic unit of scalability and contains a subset of a table's data as a contiguous, sorted range of rows stored together. The Master coordinates the HBase cluster, including assigning and balancing the regions. The Master handles all the admin operations including create/delete/modify of a table. The ZooKeeper provides distributed coordination.

Auto-sharding

A region is a subset of a table's data. When a region has too much data, the region splits into two regions. The region ➤ RegionServer association is stored in a System Table called `hbase:meta`. The `.META.` location is stored in the ZooKeeper. An example region split is shown in Figure 2-16.

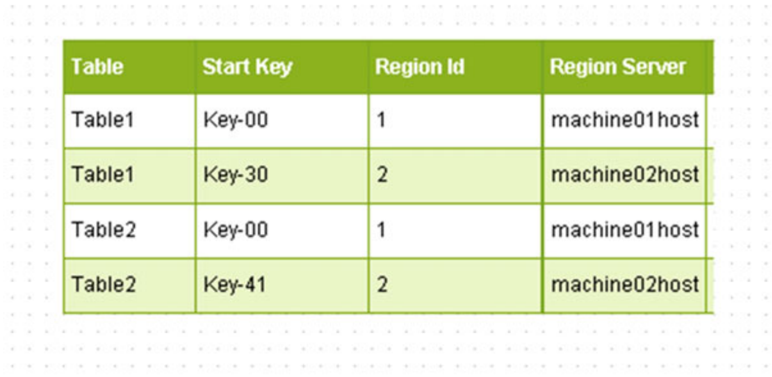


Table	Start Key	Region Id	Region Server
Table1	Key-00	1	machine01host
Table1	Key-30	2	machine02host
Table2	Key-00	1	machine01host
Table2	Key-41	2	machine02host

Figure 2-16. *An example of a region split*

The Write Path to Create a Table

The write path to create a table consists of the following sequence (and is shown in Figure 2-17):

1. The client requests the Master to create a new table with the following HBase shell command, for example:

   ```
   hbase>create 'table1','cf1'
   ```

2. The Master stores the table information, the schema.
3. The Master creates regions based on key-splits if any are provided. If no key-splits are provided, a single region is created by default.
4. The Master assigns the regions to the RegionServers. The region ➤ Region Server assignment is written to a system table called .META.

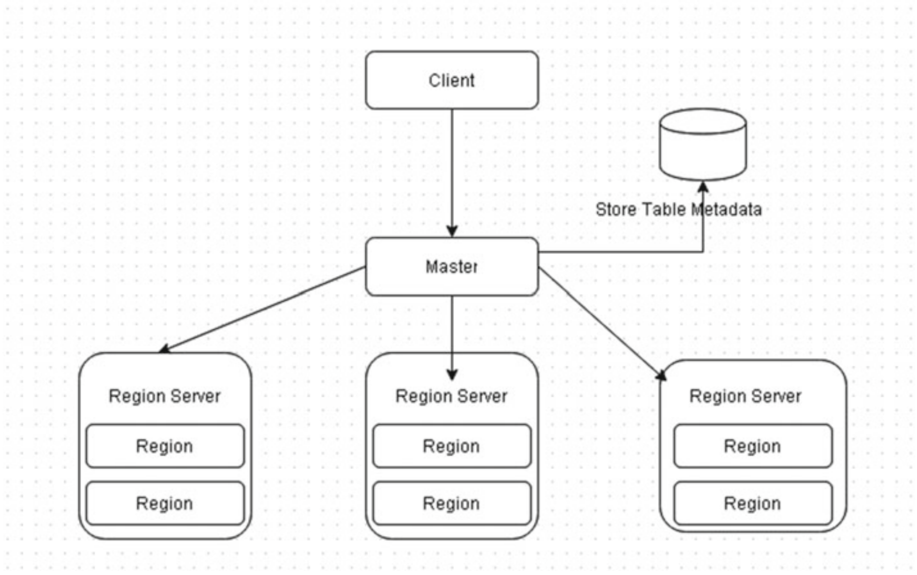


Figure 2-17. *The write path to create a table*

The Write Path to Insert Data

The write path for inserting data is as follows (and is shown in Figure 2-18):

1. Invoke `table.put(row-key:family:column,value)`.
2. The client gets the `.META.` location from the ZooKeeper.
3. The client scans .META. for the RegionServer responsible for handling the key.
4. The client requests the RegionServer to insert/update/delete the specified key/value.
5. The RegionServer processes and dispatches the request to the region responsible for handling the key. The insert/update/delete operation is written to a WAL. The `KeyVaues` are added to the store named MemStore. When the MemStore becomes full, it is flushed to a StoreFile on the disk.

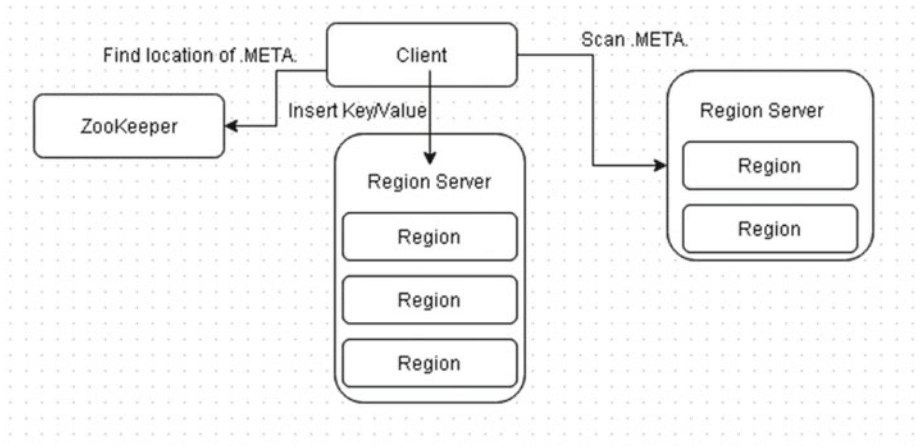


Figure 2-18. *The write path to insert data*

The Write Path to Append-Only R/W

HBase files in HDFS are append-only and immutable once closed. KeyValues are stored in memory (MemStore) and written to disk (StoreFile/HFile) when memory is full. The RegionServer is able to recover from the WAL if a crash occurs. Data is sorted by key before writing to disk. Deletes are inserts but with the “remove flag.” See Figure 2-19.

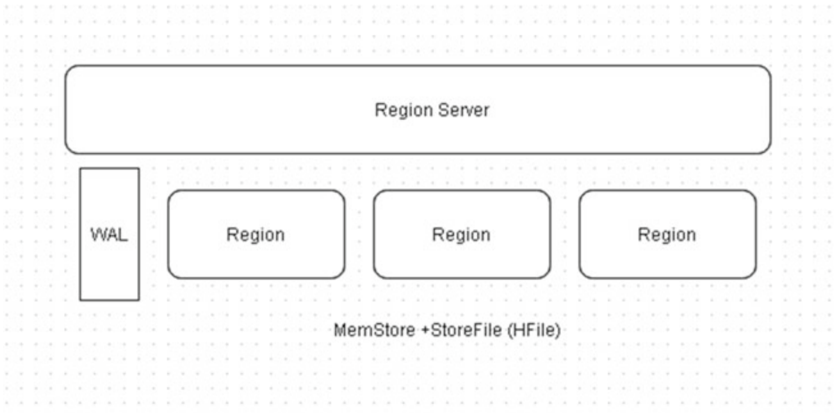


Figure 2-19. *The write path for append-only read/write*

The Read Path for Reading Data

The read path for reading data is as follows (and is shown in Figure 2-20):

1. The client gets the .META. location from the ZooKeeper.
2. The client scans .META. for the RegionServer responsible for handling the key.
3. The clients request the RegionServer to get the specified key/value.
4. The RegionServer processes and dispatches the request to the region responsible for handling the key. MemStore and store files are scanned to find the key. The key, when found, is returned to the client.

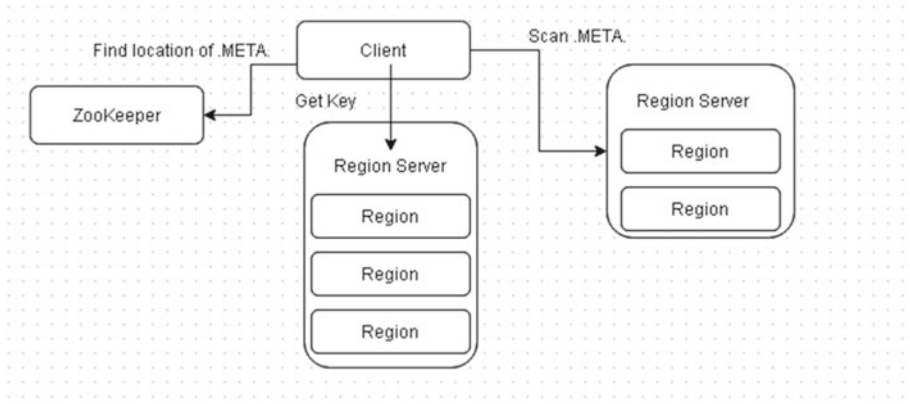


Figure 2-20. *The read path*

The Read Path Append-Only to Random R/W

Each flush of the MemStore creates a new store file. Each file has key/values sorted by a key. Two or more files can contain the same key (updates/deletes). To find a key, scan all of the files with some optimizations. To filter files, the startKey/endKey may be used. A bloom filter may also be used to find the file with the key.

HFile Format

The HFile format is designed for sequential writes with append (k,v) and large sequential reads. Records are grouped into blocks, as shown in Figure 2-21, because they are easy to split, easy to read, easy to cache, easy to index (if records are sorted), and suitable for block compression (snappy, lz4, gz).

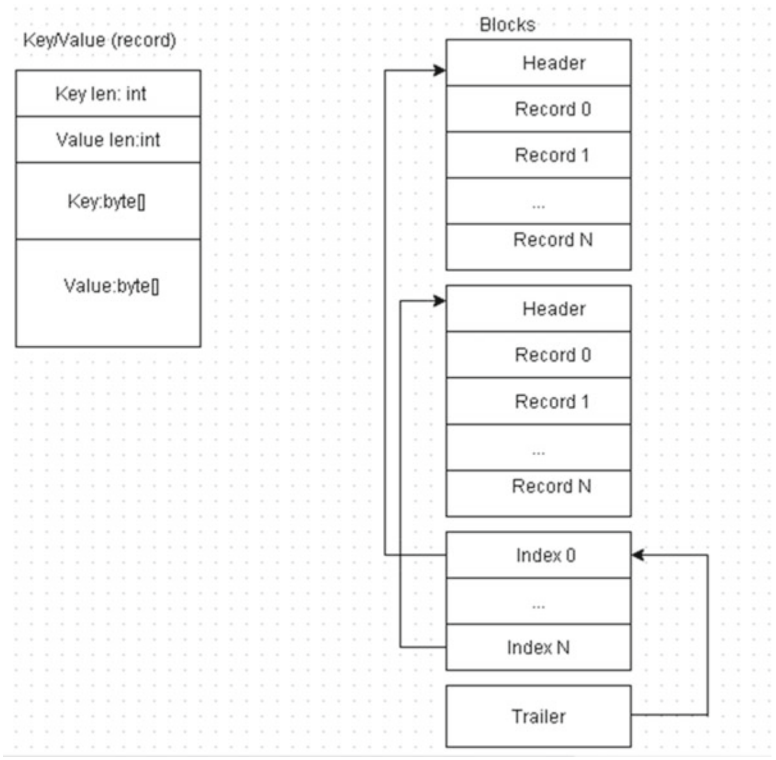


Figure 2-21. *Records are split into blocks*

The Java class `org.apache.hadoop.hbase.io.hfile.HFile` represents the file format for HBase. HFile essentially consists of sorted key/value pairs with both the keys and values being a byte array.

Data Block Encoding

Block encoding (Figure 2-22) makes it feasible to compress the key. Keys are sorted and a similar prefix can be added as a separate column with common key length bytes. Timestamps are similar and only the diff can be stored. The type is “put” most of the time. A file contains keys from one column family only.

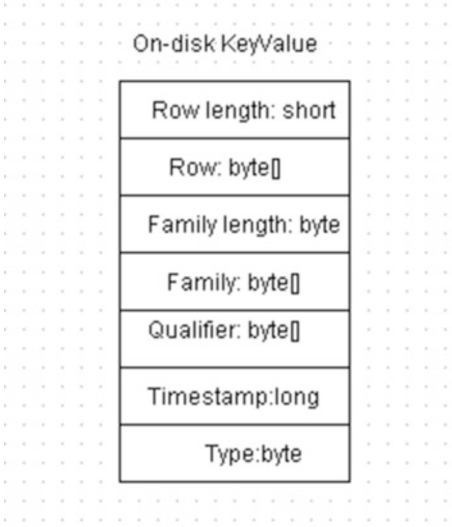


Figure 2-22. *Data block encoding*

Compactions

Compactions reduce the number of files to search during a scan by merging smaller files into one large file. Compactions remove the duplicated keys (updated values). Compactions remove the deleted keys. Old files are removed after merging. Pluggable compactions make use of different algorithms. Compactions are based on statistics (which keys/files are commonly accessed and which are not).

Snapshots

A snapshot is a set of metadata information such as the table "schema" (column families and attributes), the region's information (startKey, endKey); the list of store files, and the list of active WALs. A snapshot is not a copy of the table. Each RegionServer is responsible for taking its snapshot. Each RegionServer sores the metadata information needed for each region and the list of store files, WALs, region startKeys/endKeys. The Master orchestrates the RSs and the communication is done via the ZooKeeper. A two-phase commit like transaction (prepare/commit) is used. A table can be cloned from a snapshot.

```
hbase>clone_snapshot 'snpashotName', 'tableName'
```

Cloning creates a new table with data contained in the snapshot. No data copies are involved. HFiles are immutable and shared between tables and snapshots. Data may be inserted/updated/removed from the new table without affecting the snapshot, original tables, and the cloned tabled. On compaction or table deletion, files are removed from disk and if files are referenced by a snapshot or a cloned table, the file is moved to an "archive" directory and deleted later when no references to the file exist.

The HFileSystem Class

The `org.apache.hadoop.hbase.fs.HFileSystem` class is an encapsulation for the `org.apache.hadoop.fs.FilterFileSystem` object that HBase uses to access data. The class adds the flexibility of using separate filesystem objects for reading and writing HFiles and WALs.

Scaling

The auto-sharding feature of HBase dynamically redistributes the tables when they become too large. The smallest data storage unit is a region, which has a subset of a table's data. A region contains a contiguous, sorted range of rows that are stored together. Starting with one region, when the region becomes too large, it is split into two at the middle key, creating approximately two equal halves, as shown in Figure 2-23.

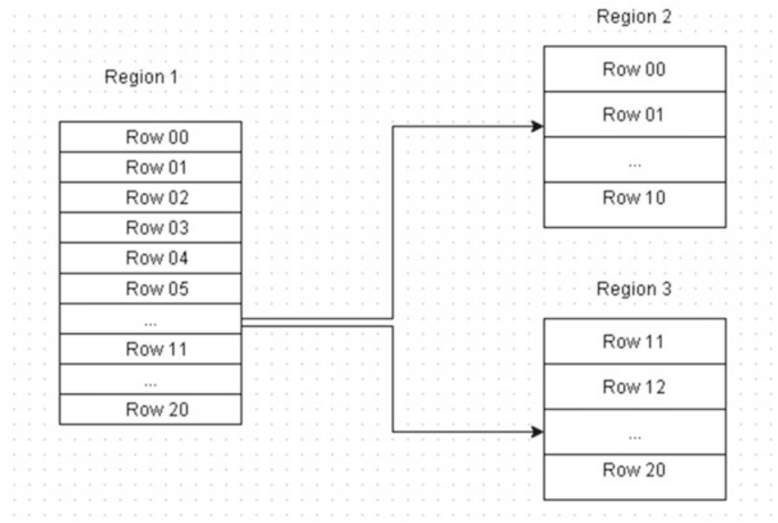


Figure 2-23. *Region splitting*

HBase has a master/slave architecture. The slaves are called RegionServers, with each RegionServer being responsible for a set of regions, and with each region being served by one RegionServer only. The HBase Master coordinates the HBase cluster's administrative operations. At startup, each region is assigned to a RegionServer and the Master may move a region from one RegionServer to the other for load balancing.

The Master also handles RegionServer failures by assigning the regions handled by the failed RegionServer to another RegionServer. The Region ➤ RegionServer mapping is kept in a system table called `.META.`. From the `.META.` table, it can be found which region is responsible for which key. In read/write operations, the Master is not involved at all and the client goes directly to the RegionServer responsible to serve the requested data.

For `Put` and `Get` operations, clients don't have to contact the Master and can directly contact the RegionServer responsible for handling the specified row. For a client scan, the client can directly contact the RegionServers responsible for handling the specified set of keys. The client queries the `.META.` table to identify a RegionServer. The `.META.` table is a system table used to track the regions. The `.META.` table contains the RegionServer names, region identifiers (Ids), the table names, and startKey for each region, as shown in Figure 2-24. By finding the startKey and the next region's startKey, clients are able to identify the range of rows in a particular region. The client contacts the Master only for creating a table and for modifications and deletions. The cluster can keep serving data even if the Master goes down.

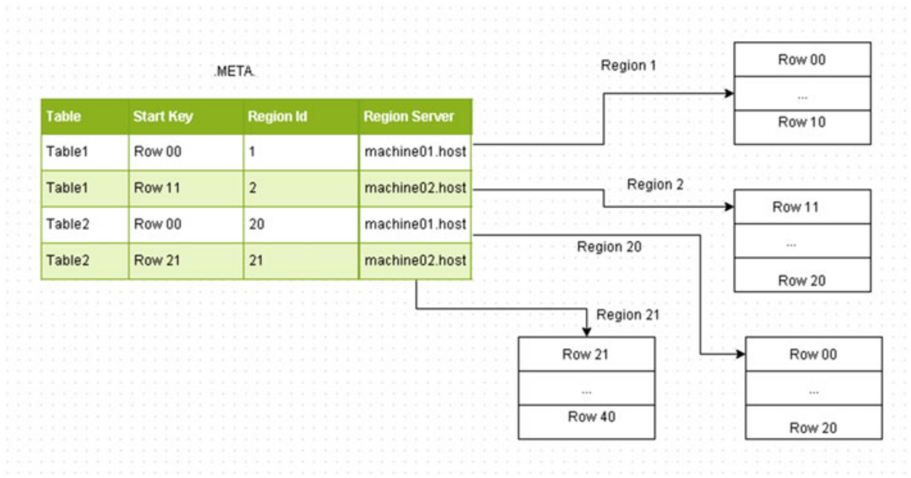


Table	Start Key	Region Id	Region Server
Table1	Row 00	1	machine01.host
Table1	Row 11	2	machine02.host
Table2	Row 00	20	machine01.host
Table2	Row 21	21	machine02.host

Figure 2-24. *The .META. table*

To avoid having to get the region location again and again, the client keeps a cache of region locations. The cache is refreshed when a region is split or moved to another RegionServer due to balancing or assignment policies. The client receives an exception when the cache is outdated and cache is refreshed by getting updated information from the `.META.` table.

The `.META.` table is also a table like the other tables and client has to find from ZooKeeper on which RegionServer the `.META.` table is located. Prior to HBase 0.96, HBase design was based on a table that contained the META locations, a table called `-ROOT-`. With HBase 0.96, the `-ROOT-` table has been removed and the META locations are stored in the ZooKeeper, as shown in Figure 2-25.

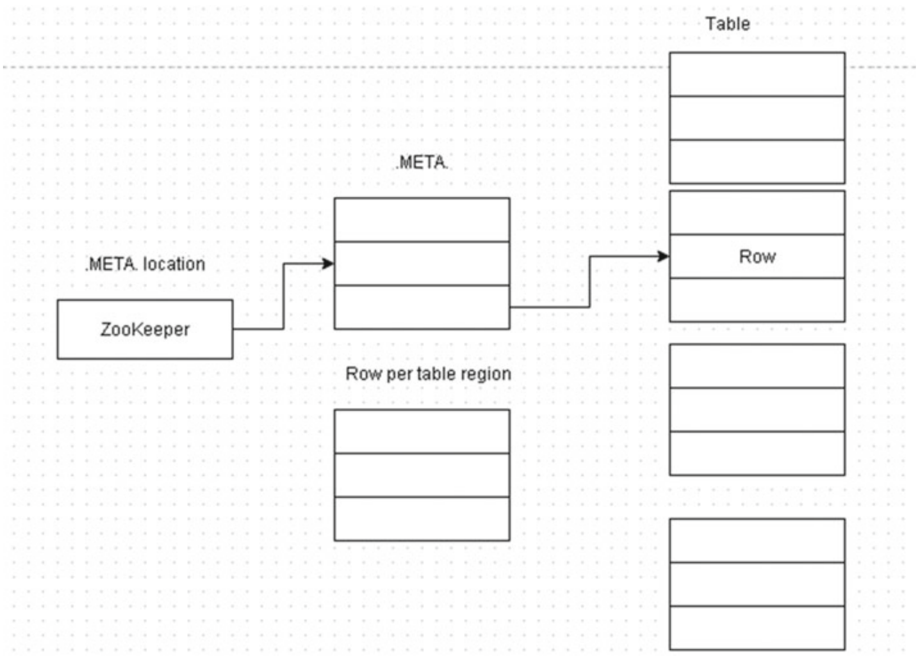


Figure 2-25. *The .META. table locations are stored in the ZooKeeper*

HBase Java Client API

The HBase Java client API is used mainly for the CRUD (Create/Retrieve/Update/Delete) operations. The HBase Java Client API provides two main interfaces, as discussed in Table 2-6.

Table 2-6. *HBase Java Client API Interfaces*

Interface	Description
`org.apache.hadoop.hbase.client.HBaseAdmin`	The HBaseAdmin is used to manage HBase database table metadata and for general administrative functions such as create, drop, list, enable, and disable tables. HBaseAdmin is also used to add and drop table column families.
`org.apache.hadoop.hbase.client.HTable`	HTable is used to communicate with a single HBase table. The class is not thread-safe for reads and writes. In a multi-threaded environment, HTablePool should be used. The interface communicates directly with the RegionServers for handling the requested set of keys. HTable is used by a client for get/put/delete and all other data operations.

Random Access

For random access on a table that is much larger than memory, HBase cache does not provide any advantage. HBase does not need to retrieve the entire file block from HDFS into memory for the data requested. Data is indexed by key and retrieved efficiently. To maximize throughput, keys are designed such that data is distributed across the servers in clusters equally. HBase blocks are the unit of indexing (also caching and compression) designed for fast random access. HDFS blocks are the unit of filesystem distribution. Tuning HDFS block size compared to HBase parameters has performance impacts.

HBase stores data in large files with sizes in the order of magnitude of 100s of MB to a GB. When HBase wants to read, it first checks the MemStore for data in memory from a recent update or insertion. If it's not in memory, HBase finds HFiles with a range of keys that could contain the data. If compactions have been run only one HFile. An HFile contains a large number of data blocks that are kept small for fast random access. At the end of the HFile an index references these blocks and keeps the range of keys in each block and offset of the block in the HFile. When an HFile is first read, the index is loaded into memory and kept in memory for future accesses.

1. HBase performs a binary search in the index, first in memory, to locate the block that could potentially contain the key.
2. When the block is located, a single disk seek is performed to load the block that is to be subsequently checked for the key.
3. The loaded 64k HBase block is searched for the key and, if found, the key-value is returned.

Small block sizes provide efficient disk usage when performing random accesses, but this increases the index size and memory requirements.

Data Files (HFile)

Data files are immutable once written. Data files are generated by flush or compactions (sequential writes). Data files are read randomly (*preads*) or sequentially. Big in size, the flushsize could be tens of GBs. All data is in blocks. The target size of data blocks is 64KB by default and is set in the `BLOCKSIZE` column family descriptor. The target size is the uncompressed and unencoded size. Index blocks (leaf, intermediate, root) also have a target size, which is set in `hfile.index.block.max.size` and is 128KB by default. Bloom filter blocks have a target size set with `io.storefile.bloom.block.size` and is 128KB by default.

The data file format for HFile v2 is shown in Figure 2-26.

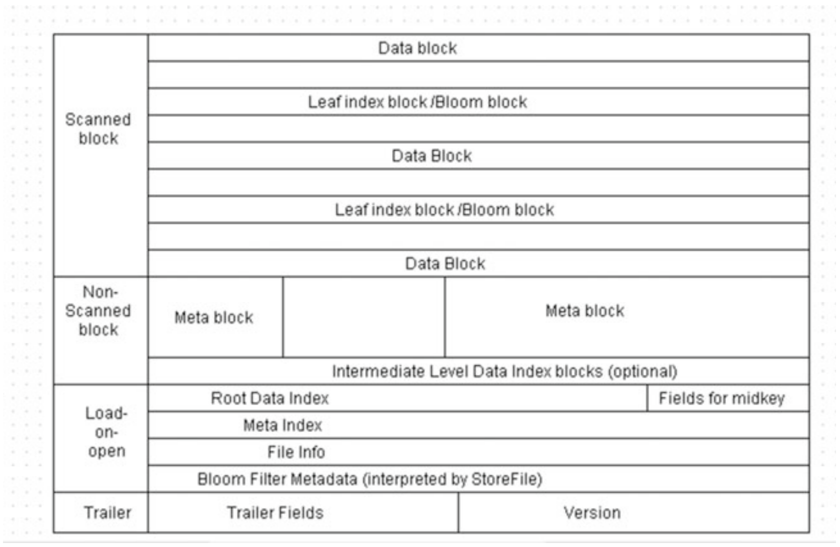
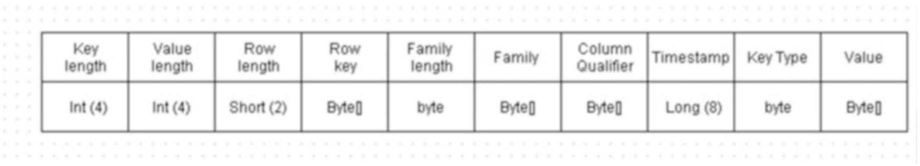


Figure 2-26. *Data file format for HFile v2*

I/O happens at block boundaries. A random read consists of disk seek plus reading a whole block sequentially. Read blocks are put into block cache so that they do not have to be read again. Leaf index blocks and bloom filter blocks also are cached. Smaller block sizes are used for faster random access. Smaller block sizes provide smaller read and faster in-block search. But smaller blocks lead to a larger block index and more memory consumption. For faster scans, use larger block sizes. The number of key-value pairs that fit an average block may also be determined. The block format is shown in Figure 2-27.



Key length	Value length	Row length	Row key	Family length	Family	Column Qualifier	Timestamp	Key Type	Value
Int (4)	Int (4)	Short (2)	Byte[]	byte	Byte[]	Byte[]	Long (8)	byte	Byte[]

Figure 2-27. *Block format*

Compression and data block encoding (PREFIX, DIFF, FAST_DIFF, PREFIX_TREE) minimizes file sizes and on-disk block sizes.

Reference Files/Links

When a region is split at a `splitkey` reference, files are created referring to the top or bottom section of the store file that is split. HBase archives data/WAL files but archives them such as `/hbase/.oldlogs` and `/hbase/.archive. HFileLink` is a kind of application-specific hard/soft link. HBase snapshots are logical links to files with backrefs.

Write-Ahead Logs

One logical WAL is created per region. One physical WAL is created per RegionServer. WALs are rolled frequently using the following settings:

- `hbase.regionserver.logroll.multiplier` with a default of 0.95
- `hbase.regionserver.hlog.blocksize` with the default the same as file system block size

WALs are chronologically ordered set of files and only the last one is open for writing. If the `hbase.regionserver.maxlogs` with a default of 32 is exceeded, a force flush is caused. Old log files are deleted as a whole. Every edit is appended. Sequential writes from WAL that sync very frequently at the rate of hundreds of time per sec. Only sequential reads from replication and crash recovery. One log file per RegionServer limits the write throughput per RegionServer.

Data Locality

HDFS local reads are called short-circuit reads.

HDFS local reads (Figure 2-28) bypass the datanode layer and directly go to the OS files. Hadoop 1.x implementation is as follows:

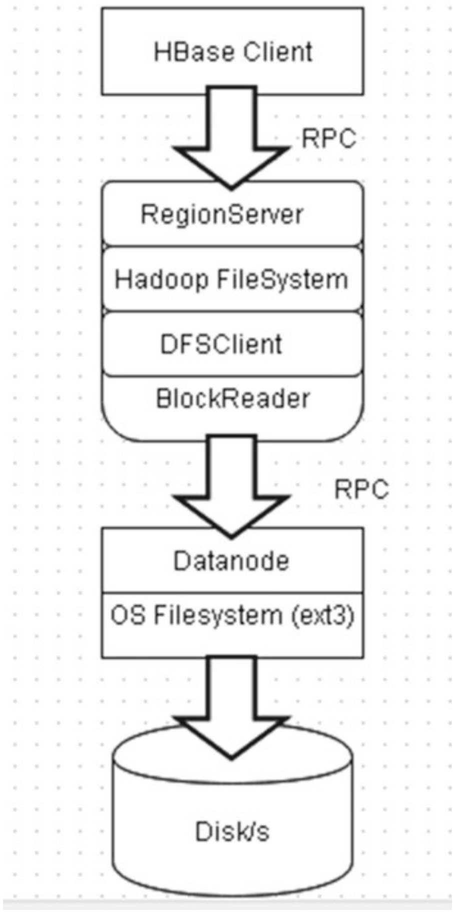


Figure 2-28. *Local reads*

`DFSClient` asks the local datanode for local paths for a block. Datanode verifies that the user has permission. The client gets the path for the block and opens the file with `FileInputStream`.

The `hdfs-site.xml` settings for a local read are as follows:

```
dfs.block.local-path-access.user=hbase
dfs.datanode.data.dir.perm = 750
```

The `hbase-site.xml` settings for a local read are as follows:

```
dfs.client.read.shortcircuit=true
```

The Hadoop 2-0 implementation of HDFS local reads includes the following settings:

- Keep the legacy implementation
- Use Unix Domain sockets to pass the File Descriptor (FD)
- The datanode opens the block file and passes FD to the `BlockReaderLocal` running in Regionserver process
- More secure than the 1.0 implementation
- Windows also supports domain sockets, needed to implement native APIs
- Local buffer size is set with `dfs.client.read.shortcircuit.buffer.size`
- `BlockReaderLocal` fills the whole buffer every time HBase tries to read an HfileBlock
- `dfs.client.read.shortcircuit.buffer.size` = 1MB vs. 64KB HFile block size
- SSR buffer is direct buffer in Hadoop 2, but not in Hadoop 1
- Local buffer size = Number of regions x Number of stores x number of avg store files x number of avg blocks per file x SSR buffer size

For example, 10 regions x 2 x 4 x 1GB/64MB x 1MB = 1.28GB of non-heap memory usage.

Checksums

HDFS checksums are not inlined. They are two files per block, one for data and one for checksums, as shown in Figure 2-29.

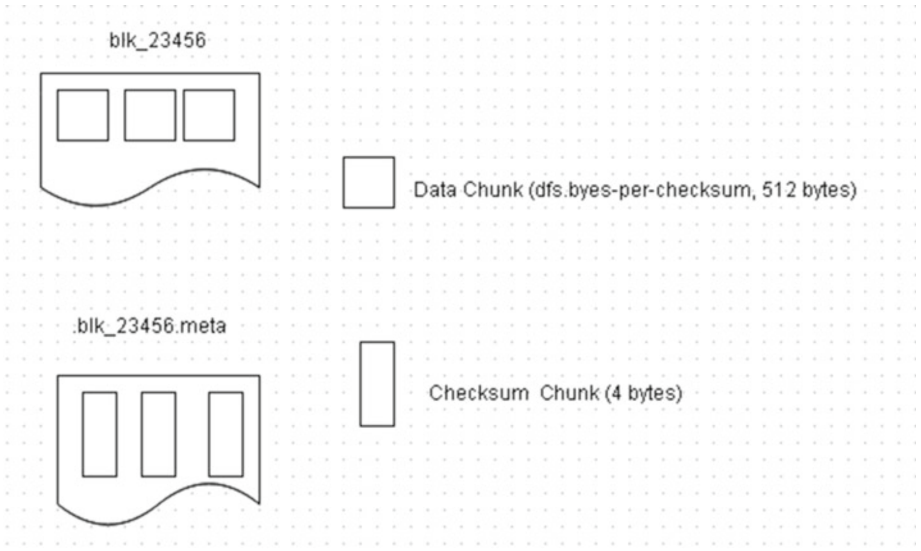


Figure 2-29. *Two files per block*

Random positioned read causes two seeks. HBase checksums are included with 0.94. HFile v 2-1 writes checksums per HFile block. The HFile data block chunk and the Checksum chunk are shown in Figure 2-30. HDFS checksum verification is bypassed on block read as checksum verification is done by HBase. If the HBase checksum fails, revert to checksum verification from HDFS for some time. Use the following settings:

```
hbase.regionserver.checksum.verify = true
hbase.hstore.bytes.per.checksum =16384
hbase.hstore.checksum.algorithm=CRC32C
```

Do not set

```
dfs.client.read.shortcircuit.skip.checksum = false
```

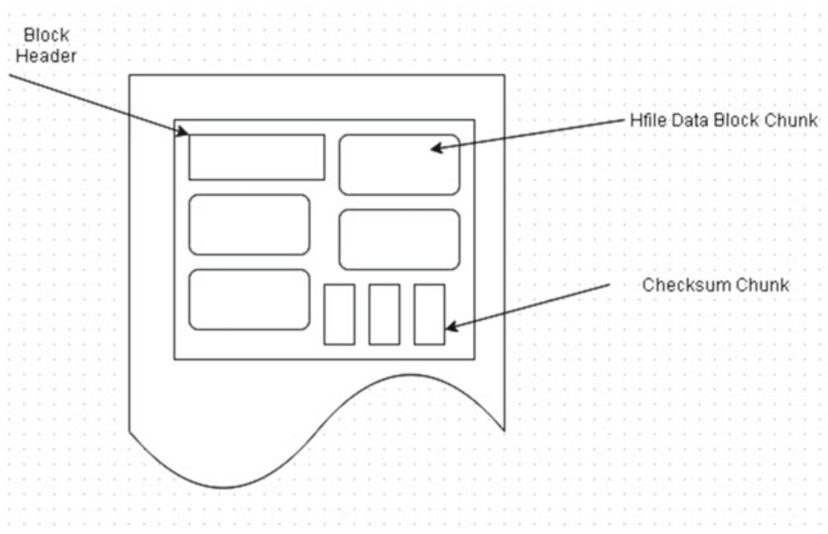


Figure 2-30. *The HFile data block chunk and the Checksum chunk*

Data Locality for HBase

Data locality is low when a region is moved as a result of load balancing or region server crash and failover. Most of the data is not local unless the files are compacted. When writing a data file, provide hints to the NameNode for locations for block replicas. The load balancer should assign a region to one of the affiliated nodes upon server crash to keep data locality and SSR. Data locality reduces data loss probability.

MemStore

When a RegionServer receives a write request, it directs the request to a specific region, with each region storing a set of rows. Row data can be separated in multiple column families. Data for a particular column family is stored in HStore, which is comprised of MemStore and a set of HFiles. The MemStore is kept in the RegionServer's main memory and HFiles are written to HDFS. Initially, a write request is written to the MemStore and when a certain memory threshold is exceeded, the MemStore data gets flushed to an HFile. The MemStore is used because data is stored on HDFS by row key.

As HDFS is designed for sequential reads/writes with no file modifications. HBase is not able to efficiently write data to disk as it is received because the written data is not sorted and not optimized for future retrieval. HBase buffers last received data in memory (MemStore), sorts it before flushing, and writes to HDFS using sequential writes. Some of the other benefits of MemStore are as follows:

1. MemStore is an in-memory cache for recently added data, which is useful when last-written data is accessed more frequently than older data
2. Certain optimizations are done on rows/cells in memory before writing to persistent storage

Every MemStore flush creates one HFile per column family. When reading HBase, first check if the requested data is in the MemStore and, if not found, go to the HFile to get the requested data. Frequent MemStore flushes can affect reading performance and bring additional load to the system. Every flush creates an HFile, so frequent flushes will create several HFiles, which will affect the read performance because HBase must read several HFiles. HFiles compaction alleviates the read performance issue by compacting multiple smaller files into a larger file. But compaction is usually performed in parallel with other requests and it could block writes on region server.

Summary

In this chapter, I discussed how HBase stores data in HDFS, including the read-write path and how HBase communicates with the NameNode and the Datanode. I discussed the HBase storage architecture and the HFile format used for storing data. Data encoding, compactions, and replica placement policy were also discussed. In the next chapter, I will discuss the characteristics that make an application suitable for Apache HBase.

CHAPTER 3

Application Characteristics

Apache HBase is designed to be used for random, real-time, relatively low latency, read/write access to big data. HBase's goal is to store very large tables with billions/millions of rows and billions/millions of columns on clusters installed on commodity hardware.

The following characteristics make an application suitable for HBase:

- Large quantities of data in the scale of 100s of GBs to TBs and PBs. Not suitable for small-scale data
- Fast, random access to data
- Variable, flexible schema. Each row is or could be different
- Key-based access to data when storing, loading, searching, retrieving, serving, and querying
- Data stored in collections. For example, some metadata, message data, or binary data is all keyed into the same value
- High throughput in the scale of 1000s of records per second
- Horizontally scalable cache capacity. Capacity may be increased by just adding nodes
- The data layout is designed for key lookup with no overhead for sparse columns
- Data-centric model rather than a relationship-centric model. Not suitable for an ERD (entity relationship diagram) model
- Strong consistency and high availability are requirements. Consistency is favored over availability
- Lots of insertion, lookup, and deletion of records
- Write-heavy applications
- Append-style writing (inserting and overwriting) rather than heavy read-modify-write

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_3

Some use-cases for HBase are as follows:

- Audit logging systems
- Tracking user actions
- Answering queries such as
    - What are the last 10 actions made by the user?
    - Which users logged into the system on a particular day?
- Real-time analytics
    - Real-time counters
    - Interactive reports showing trends and breakdowns
    - Time series databases
- Monitoring system
- Message-centered systems (Twitter-like messages and statuses)
- Content management systems serving content out of HBase
- Canonical use-cases such as storing web pages during crawling of the Web

HBase is not suitable/optimized for

- Classical transactional applications or relational analytics
- Batch MapReduce (not a substitute for HDFS)
- Cross-record transactions and joins

HBase is not a replacement for RDBMS or HDFS. HBase is suitable for

- Large datasets
- Sparse datasets
- Loosely coupled (denormalized) records
- Several concurrent clients

HBase is not suitable for

- Small datasets (unless many of them)
- Highly relational records
- Schema designs requiring transactions

Summary

In this chapter, I discussed the characteristics that make an application suitable for Apache HBase. The characteristics include fast, random access to large quantities of data with high throughput. Application characteristics not suitable were also discussed. In the next chapter, I will discuss the physical storage in HBase.

PART II

Data Model

CHAPTER 4

Physical Storage

The filesystem used by Apache HBase is HDFS, as discussed in Chapter 2. HDFS is an abstract filesystem that stores data on the underlying disk filesystem. HBase indexes data into HFiles and stores the data on the HDFS Datanodes. HDFS is not a general purpose filesystem and does not provide fast record lookups in *files*. HBase is built on top of HDFS and provides fast record lookups and updates for large *tables*. HBase stores table data as key/value pairs in indexed HFiles for fast lookup. HFile, the file format for HBase, is based on the TFile binary file format. HFile is made of blocks, with the block size configured per column family. The block size is 64k. If the key/value exceeds 64k of data, it's not split across blocks but the key/value is read as a coherent block. The HFile is replicated three times for durability, high availability, and data locality. Data files are stored in the format discussed in Chapter 2 and the `HFileSystem.create(Path path)` method is used to create the HFile data files.

A HBase client communicates with the ZooKeeper and the HRegionServers. The HMaster coordinates the RegionServers. The RegionServers run on the datanodes. Each RegionServer is collocated with a datanode, as shown in Figure 4-1.

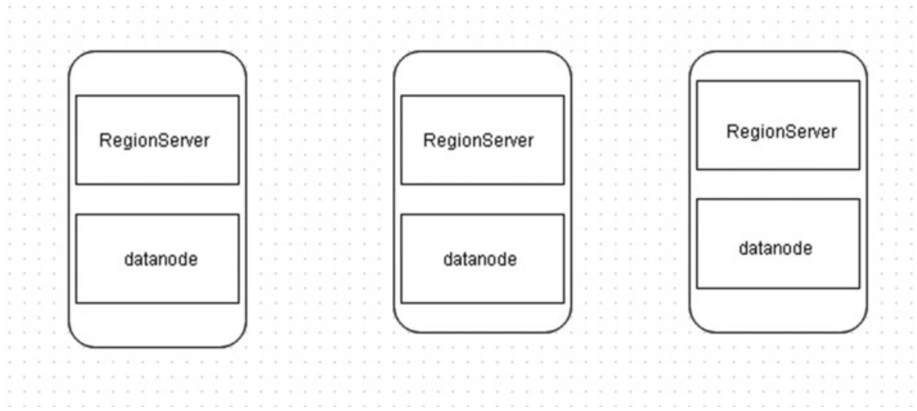


Figure 4-1. *RegionServer collocation with a datanode*

The storage architecture of HBase is discussed in more detail in Chapter 2 and shown in Figure 2-3.

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_4

Summary

In this chapter, I discussed how HBase physically stores data in HDFS, with each RegionServer being collocated with a datanode. In the next chapter, I will discuss the column family and the column qualifier.

CHAPTER 5

Column Family and Column Qualifier

Column qualifiers are the column names, also known as column keys. For example, in Figure 5-1, Column A and Column B are the column qualifiers. A table value is stored at the intersection of a column and a row. A row is identified by a row key. Row keys with the same user ID are adjacent to each other. The row keys form the primary index and the column qualifiers form the per row secondary index. Both the row keys and the column keys are sorted in ascending lexicographical order.

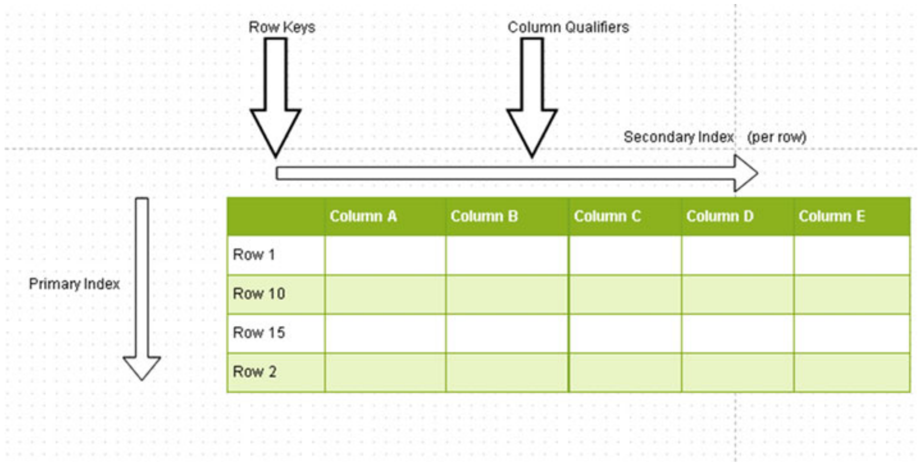


Figure 5-1. *Column qualtifiers*

Each row can have different column qualifiers, as shown in Figure 5-2. HBase stores the column qualifier with a certain value which is part of the row key. Apache HBase doesn't limit the number of column qualifiers, which means that the creation of long column qualifiers can require a lot of storage.

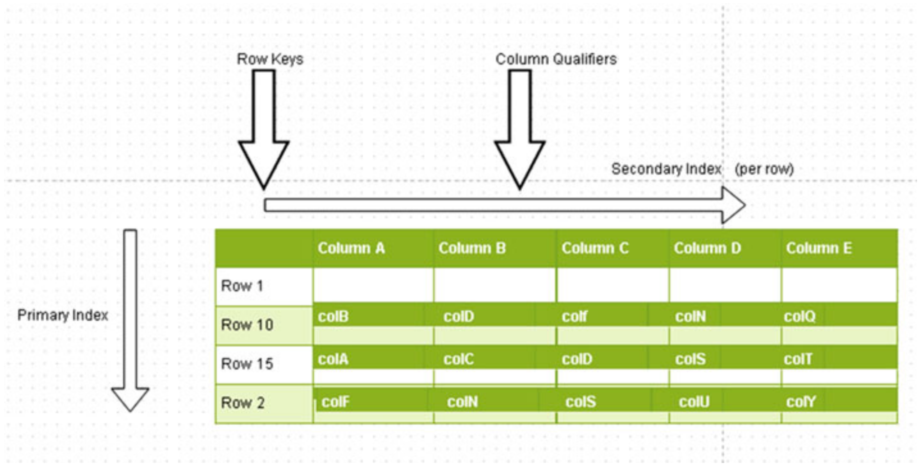


***Figure 5-2.** Different column qualifiers*

The HBase data model consists of a table, which consists of multiple rows. A row consists of a row key and one or more columns, and the columns have values associated with them. Rows are sorted lexicographically and stored. To store related rows adjacent or near each other, a common row key pattern is usually used. A column, which stores a value, consists of a column family name and a column qualifier delimited by a : (colon). For example, column family `cf1` could consist of column qualifiers (or column keys) `c1`, `c2,` and `c3`. A column family consists of a collocated set of columns. Each column family has a set of storage properties such as whether the column values are to be cached in memory, how the row keys are encoded, or how data is compressed. A column's family is used for performance reasons. Each row in a table has the same column family/ies, although a row does not have to store a value in each column family. A column qualifier is the actual column name to provide an index for a column. A column qualifier is added to a column family with the two separated by a : (colon) to make a column. Though each row in a table has the same column families, the column qualifiers associated with each column family can be different. Column qualifiers are the actual column names, or column keys. For example, the HBase table in Figure 5-3 consists of column families `cf1`, `cf2,` and `cf3`. And column family `cf1` consists of column qualifiers `c1`, `c2`, and `c3` while column family `cf2` consists of column qualifiers `c2`, `c4`, and `c5`, and column family `cf3` consists if column qualifiers `c4`, `c6`, and `c7`. Column families are fixed when a table is created; column qualifiers are not fixed when a table is created and are mutable and can vary from row to row. For example, in Figure 5-3, the table has three column families (`cf1`, `cf2`, and `cf3`) and each row has different column qualifiers associated with each column family and each column. Some rows do not store data in some of the column families while other rows have data in each of the column families. For example, `Row-1` has data stored in each of the column families while `Row-10` has only two of the column families and `Row-15` has only one column family.

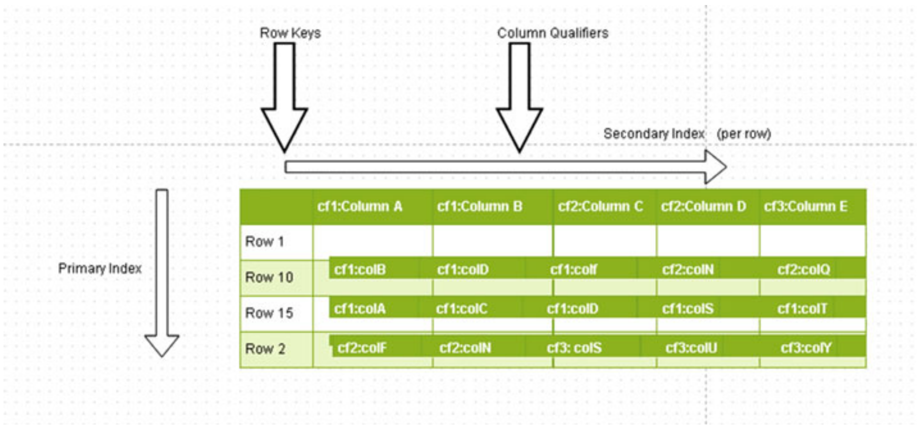


Figure 5-3. *Using column qualifiers*

What makes a HBase table sparse is that each row does not have to include all the column families. Each column family is stored in its own data file. As a result, some data files may not include data for some of the rows if the rows do not store data in those column families.

A `KeyValue` consists of the key and a value, with the `key` being comprised of the row key plus the column family plus the column qualifier plus the timestamp, as shown in Figure 5-4. The value is the data identified by the key. The `timestamp` represents a particular version.

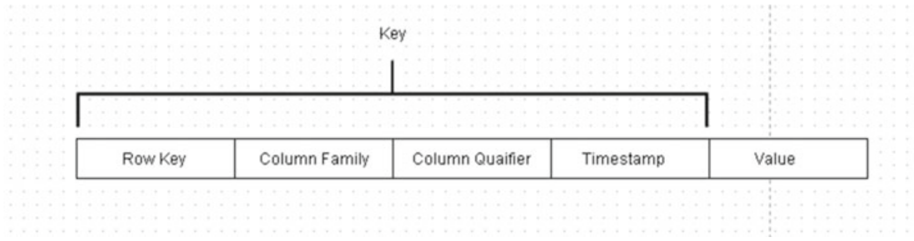


Figure 5-4. *Key and value*

As another representation, a row is shown to have a row key and two column families. `Column Family 1` has three column qualifiers associated with it. `ColQ1` column qualifier has three versions, and each version is associated with a different value, as shown in Figure 5-5.

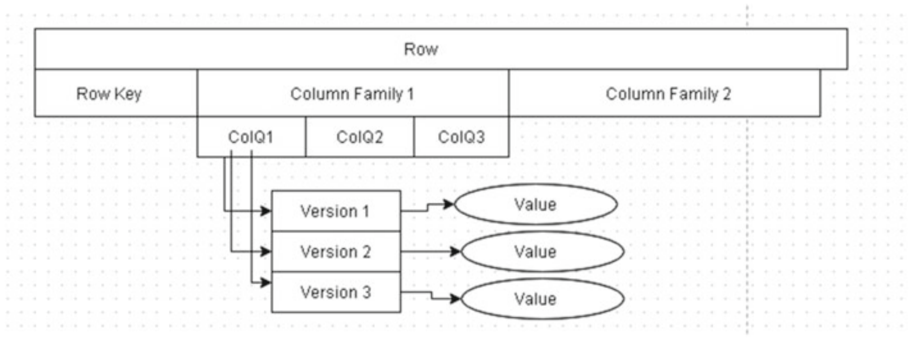


Figure 5-5. *Relationship between column qualifier, version, and value*

The essential differences between a column family and a column qualifier are listed in Table 5-1.

Table 5-1. *Differences Between a Column Family and a Column Qualifier*

	Column Family	**Column Qualifier**
Mutability	Not mutable	Mutable
Schema	Each row has same column families.	Each row can have different column qualifiers within a column family.
Column notation	Column family is the prefix, for example `cf1:col1`. All column members of a column family have the same prefix.	Column qualifier is the suffix, for example `cf2:col2`.
Empty	A column family must not be empty when identifying a column.	A column qualifier could be empty; for example, `cf1:` is a column in column family `cf1` with an empty column qualifier.
Number	Any number, storage space permitting.	Any number, storage space permitting.
Storage unit	Data stored per column family in a separate data file called a HFile.	Data is not stored per column qualifier. A HFile could have several column qualifiers associated with it.

It is recommended to use a few column families because each column family is stored in its own data file and too many column families can cause many data files to be open. Compactions may be required with several column families.

The following is an example of a data file HFile for column family `cf1`:

```
123 cf1  col1 val1  @ ts1
123 cf1  col2 val2  @ ts1
235 cf1  col1 val3  @ ts1
235 cf1  col2 val4  @ ts1
235 cf1  col2 val5  @ ts2
```

The HFile has two row keys, `123` and `235`. Row key `123` has two column qualifiers associated with it: `col1` and `col2`. Each of the column qualifiers has a value associated with it and a timestamp. Row key `235` has two column qualifiers, also `col1` and `col2`, but `col2` has two versions or timestamps associated with it (`ts1` and `ts2`).

Summary

In this chapter, I introduced the column family and the column qualifier and the relationship between the two. In the next chapter, I will discuss row versioning.

CHAPTER 6

Row Versioning

When data is stored in HBase, a *version* (also called a *timestamp*) is required for each value stored in a cell. The timestamp is created automatically by the RegionServer or may be supplied explicitly. By default, the timestamp is the time at the RegionServer when the data was written. Alternatively, the timestamp may be set explicitly. Timestamps are stored in descending order in an HFile, which implies the most recent timestamp is stored first. The timestamp identifies a version and must be unique for each cell. A `{row, column, version}` tuple specifies a cell in a table. A *`KeyValue`* consists of the key and a value with the key being comprised of the *row key* + *column Family* + *column qualifier* + *timestamp*, as shown in Figure 5-4. The value is the data identified by the key. The timestamp represents a particular version.

Each column consists of any number of versions, which implies that any number of tuples in which the row and column are the same and only the version is different could be created. Typically, the version is the timestamp. The version applies to the actual data stored in a cell, the intersection of a row key with a column key. Coordinates for a cell are row key ➤ column key ➤ version. As an example, a row is shown to have a row key and two column families. Column `Family 1` has three column qualifiers associated with it. The `ColQ1` column qualifier has three versions, and each version has a different value, as shown in Figure 6-1.

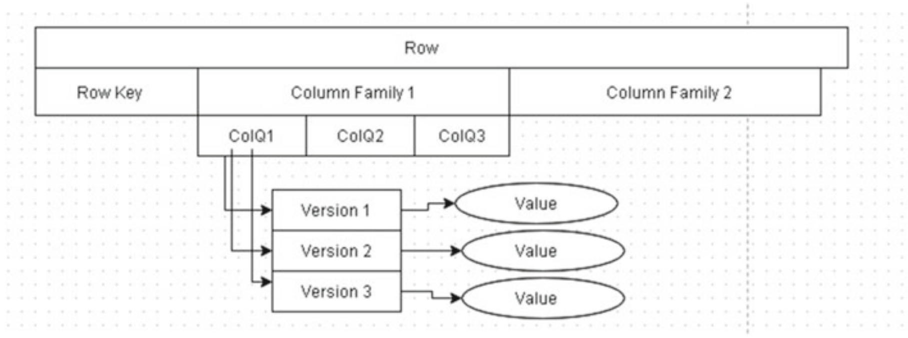


Figure 6-1. *An example of a column qualifier with three versions*

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_6

Physical coordinates for a cell are region directory ➤ column family directory ➤ row key ➤ column family name ➤ column qualifier ➤ version.

The row and column keys are stored as bytes and the versions as `long` integers. The versions are stored in decreasing order so that when reading a `StoreFile` the most recent version is found first.

Doing a `Put` on a table always creates a new version of a cell identified by a timestamp. By default, `currentTimeMillis` is used to create the timestamp. The version may be specified explicitly on a per column basis. The `long` value of a version can be a time in the past or the future or a non-time long value. An existing version may be overwritten by doing a `Put` at exactly the same *`{row, column, version}`* with a different or same value.

The `org.apache.hadoop.hbase.client.Put` class is used to perform `Put` operations for a single row. To perform a `Put,` first instantiate a `Put` object for which the constructors listed in Table 6-1 are provided.

Table 6-1. *Put Class Constructors*

Put Constructor	Description
`Put(byte[] row)`	Creates a `Put` object for the specified row byte array.
`Put(byte[] rowArray, int rowOffset, int rowLength)`	Creates a `Put` object from the specified row array using the given offset and row length.
`Put(byte[] rowArray, int rowOffset, int rowLength, long ts)`	Creates a `Put` object from the specified row array using the given offset, row length, and timestamp.
`Put(byte[] row, long ts)`	Creates a `Put` object for the specified row byte array and timesamp.
`Put(ByteBuffer row)`	Creates a `Put` object for the specified row byte buffer.
`Put(ByteBuffer row, long ts)`	Creates a `Put` object for the specified row byte buffer and timestamp.
`Put(Put putToCopy)`	Copies a `Put` object.

After a `Put` object has been created, columns may be added to it using one of the overloaded `addColumn` methods (shown in Table 6-2), each of which returns a `Put` object.

Table 6-2. Overloaded addColumn Method

Method	Description
`addColumn(byte[] family, byte[] qualifier, byte[] value)`	Adds a column using the specified column family, column qualifier, and value, each of type `byte[]`. The version or timestamp is created implicitly.
`addColumn(byte[] family, byte[] qualifier, long ts, byte[] value)`	Adds a column using the specified column family, column qualifier, timestamp, and value, each of type `byte[]` except the timestamp, which is of type `long`. The version or timestamp is created explicitly.
`addColumn(byte[] family, ByteBuffer qualifier, long ts, ByteBuffer value)`	Adds a column using the specified column family of type `byte[]`, column qualifier of type ByteBuffer, timestamp of type `long`, and value of type `ByteBuffer`. The version or timestamp is created explicitly.

The versions are configurable for a column family. In CDH 5 (>=0.96), the maximum number of versions is 1 by default. In earlier CDH (<0.96), the maximum number of versions defaults to 3. The default setting may be configured with `hbase.column.max.version` in `hbase-site.xml`. The maximum number of versions may be altered using the `alter` command with `HColumnDescriptor.DEFAULT_VERSIONS`. For example, the following command sets the maximum number of versions to 5 for column family `cf1` in table `table1`:

```
alter ‘table1', NAME => ‘cf1', VERSIONS => 5
```

Excess versions are removed during major compactions. The minimum number of versions may also be set and defaults to 0, which implies the feature is disabled and no minimum is configured. For example, the following `alter` command sets the minimum number of versions to 1 in column family `cf1` in table `table1`:

```
alter ‘table1', NAME => ‘cf1', MIN_VERSIONS => 1
```

Versions Sorting

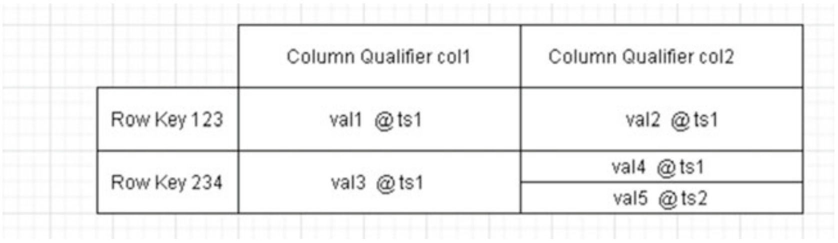
Versions are sorted from newest to oldest by sorting the timestamps lexicographically. When a version needs to be deleted because of reaching a threshold, for example, the “oldest” version lexicographically is deleted even if the version is the most recent added.

If multiple writes to a cell are made using the same timestamp, only one of those versions is kept and it is undefined which one. If multiple writes to a cell are made with out-of-order timestamps and the number of versions exceeds the maximum versions setting, only the highest timestamps versions are kept. Existing versions may be updated. And versions may be added out of order, implying that a more recent timestamp may be added before an older timestamp. When doing a `Get` with no explicit version specified, the most recent version is returned.

The following is an example of a data file HFile for column family `cf2`:

```
123 cf2  col1 val1  @ ts1
123 cf2  col2 val2  @ ts1
235 cf2  col1 val3  @ ts1
235 cf2  col2 val4  @ ts1
235 cf2  col2 val5  @ ts2
```

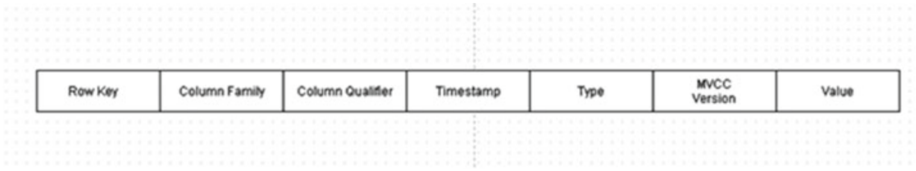
The HFile has two keys: `123` and `235`. Row key `123` has two column qualifiers associated with it: `col1` and `col2`. Each of the column qualifiers has a value associated with it and a timestamp. Row key `235` has two column qualifiers, also `col1` and `col2`, but `col2` has two versions or timestamps associated with it, `ts1` and `ts2`, as shown in Figure 6-2.



	Column Qualifier col1	Column Qualifier col2
Row Key 123	val1 @ts1	val2 @ts1
Row Key 234	val3 @ts1	val4 @ts1 val5 @ts2

Figure 6-2. *Row key 234 ➤ column qualifier col2 is associated with two versions.*

The unit of storage in HBase consists of the following fields: row key, column family, column qualifier, timestamp, type, MVCC (multiversion concurrency control) version, and value, as shown in Figure 6-3.



Row Key	Column Family	Column Qualifier	Timestamp	Type	MVCC Version	Value

Figure 6-3. *Row key 234 ➤ column qualifier col2 is associated with two versions.*

Uniqueness is determined by row key, column family, column qualifier, timestamp, and type. Cells with a lower timestamp are sorted first.

Summary

In this chapter, I discuss how a row key ➤ column qualifier (a cell) can be associated with multiple versions indicated by timestamps. Each version or timestamp represents a value, and when a value is requested, the latest value stored is returned. In the next chapter, I will discuss logical storage.

CHAPTER 7

Logical Storage

A cell is the logical storage unit for the Apache HBase data model. Cells are stored individually, and empty cells are not stored at all, which makes HBase storage sparse. Values are stored as an array of bytes. A *{row, column, version}* tuple specifies a cell in a table. Cells store data as uninterpreted bytes. The *timestamp* identifies a version and must be unique for each cell.

Three coordinates define each cell. Coordinates for a cell are row key ➤ column key ➤ version. The column key includes the column family. At a fine-grained level, including the table, a cell's coordinates are table ➤ row key ➤ column family ➤ column key ➤ version. Cells are sorted lexicographically by their row key. Row keys form the primary index. The coordinates are accompanied by a cell value as the cell value is transferred through the system. Typically, the version is the timestamp, which implies that the coordinates for a cell could be described as row key ➤ column key ➤ timestamp. The timestamp applies to the actual data stored in a cell, the intersection of a row key with a column key. Timestamps are stored in decreasing order with the most recent first. Cell versions may be constrained by predicate deletions such as store-only values from the previous day. The timestamp by default identifies the time at which a cell is created. By default, `currentTimeMillis` is used to create the timestamp. Timestamps may be specified explicitly on a per column basis.

Multiple versions of the same cell are stored consecutively including the timestamp. Cells are sorted in descending order of timestamp, newest value first. The entire cell including all the structural information is a `KeyValue` object and includes {row key, column key {column family:column qualifier}, timestamp, and value}. A KeyValue object is sorted by row key first (primary index) and by column key next (secondary index). A KeyValue could also be called a cell.

All row keys with the same `row key` are collocated on the same RegionServer, which makes ACID (Atomicity, Consistency, Isolation, Durability) guarantees for updates with the same row key feasible without complex and slow two-phase commit or paxos.

Data in a cell is not updated on a table update. Every update creates a new cell.

Using the same example as introduced in Chapter 5 of a row having a row key and two column families, column `Family 1` has three column qualifiers associated with it. The `ColQ1` column qualifier has three versions and each version has a different value. The value is stored in a cell. Physical coordinates for a cell are region directory ➤ column family directory ➤ row key ➤ column family name ➤ column qualifier ➤ version.

In Figure 7-1, the intersection of row `10` and column `A` has three cells, not one. Each cell is identified with a version. Each cell stores a value.

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_7

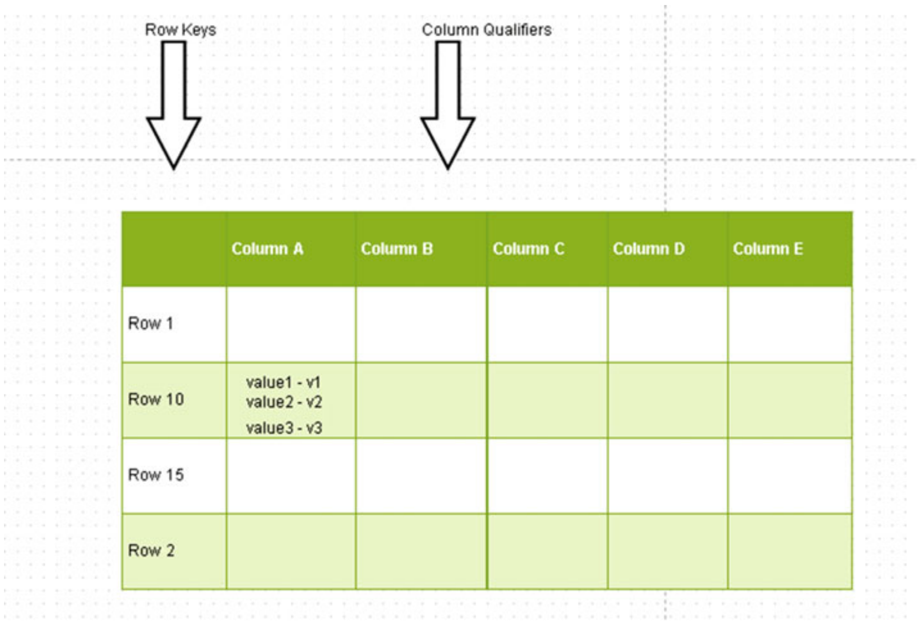


Figure 7-1. *An example of a column qualifier with three versions*

As discussed before, the unit of storage in HBase consists of the fields row key, column family, column qualifier, timestamp, type, MVCC (multiversion concurrency control) version. Value and uniqueness are determined by the row key, column family, column qualifier, timestamp, and key type, as shown in Figure 7-2. Cells with a lower timestamp are sorted first.

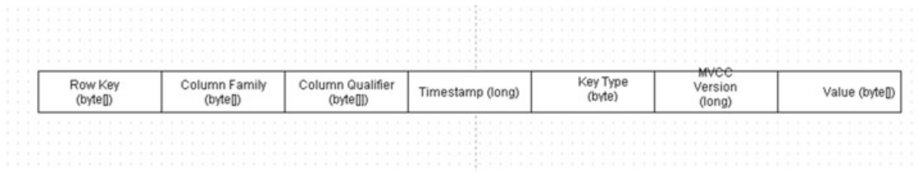


Figure 7-2. *Unit of storage*

The value in a cell is stored as a `byte[]`. HBase does not make use of any typing for key or value. All key/values for the same column family are stored in the same HFile. Using the same example of a data file, the HFile for column family `cf1`, data for row key `235` has two column qualifiers (`col1` and `col2`) and {row key `235`, `col2`} has two different versions or timestamps associated with it.

```
123 cf1  col1 val1  @ ts1
123 cf1  col2 val2  @ ts1
235 cf1  col1 val3  @ ts1
235 cf1  col2 val4  @ ts1
235 cf1  col2 val5  @ ts2
```

Data stored in cells is distributed across hundreds or thousands of machines.

Summary

In this chapter, I discussed the logical storage of HBase as comprising of a {row, column, version} tuple, which constitutes a table cell. I also discussed the unit of storage and the data type associated with each field. In the next chapter, I will discuss the major components of an HBase cluster.

PART III

■ ■ ■

Architecture

CHAPTER 8

Major Components of a Cluster

An HBase cluster can consist of one or more nodes with its components distributed across the cluster. The major components of HBase cluster are as follows:

1. Master
2. RegionServers
3. ZooKeeper

HBase runs in two modes: *standalone* and *distributed*. On a distributed cluster, the Master is typically on the same node as the HDFS NameNode, and the RegionServers are on the same node as a HDFS Datanode, with each RegionServer being collocated with a datanode. For a small cluster, a ZooKeeper may be collocated with the NameNode (not the datanode), but for a large cluster, the ZooKeeper should run on a separate node. The major components, including the subcomponents, are shown in Figure 8-1.

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_8

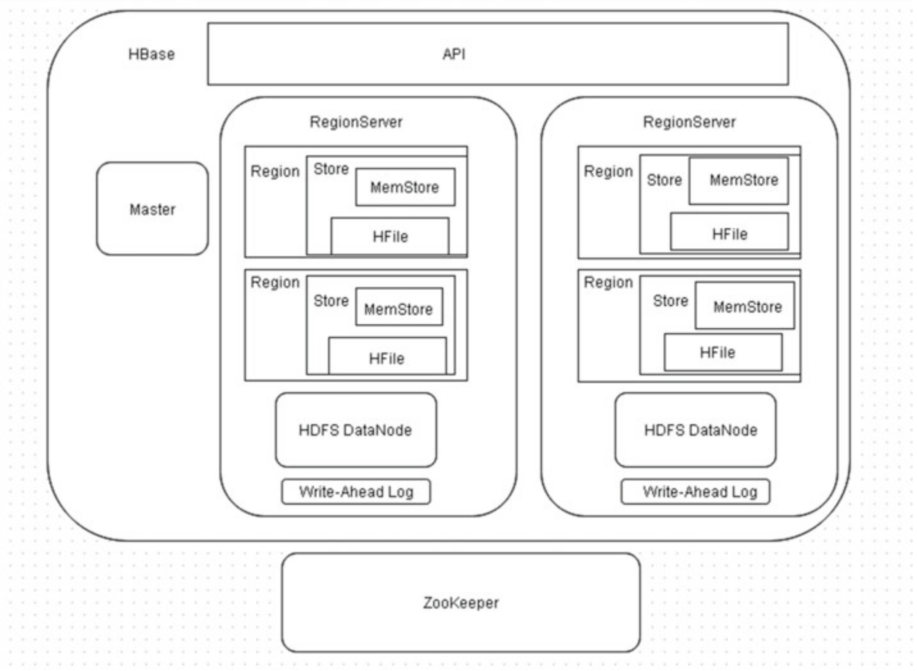


Figure 8-1. *Major components of Apache HBase*

Master

The Master manages the cluster. The Master assigns Regions to RegionServers on startup and failover, and performs load balancing. The Master is not a component in the data storage or retrieval path. HBase handles the DDL operations such as Create and Delete table. The Master server monitors all RegionServers in the cluster, and all metadata changes are made via the Master. The Master runs on the NameNode on HDFS while the RegionServers are collocated with the HDFS Datanodes. A multi-Master environment is supported, in which one Master is active and is registered with the ZooKeepers, and if the active Master fails, another Master becomes the active Master. HMaster is the Java interface for the Master server. The Master runs several background threads including the load balancer. The load balancer balances a cluster's load by moving regions around.

RegionServers

RegionServers manage data. A table's row keys are distributed across the cluster stored on different RegionServers. For example, if a table has row keys A, H, N, S, V, Y, Z, row keys [D-G] and [V-G] could be on RegionServer 1, row keys [A-C] and [R-U] could be on RegionServer 2, and row keys [H-M] and [N-Q] could be on RegionServer 3, as shown in Figure 8-2.

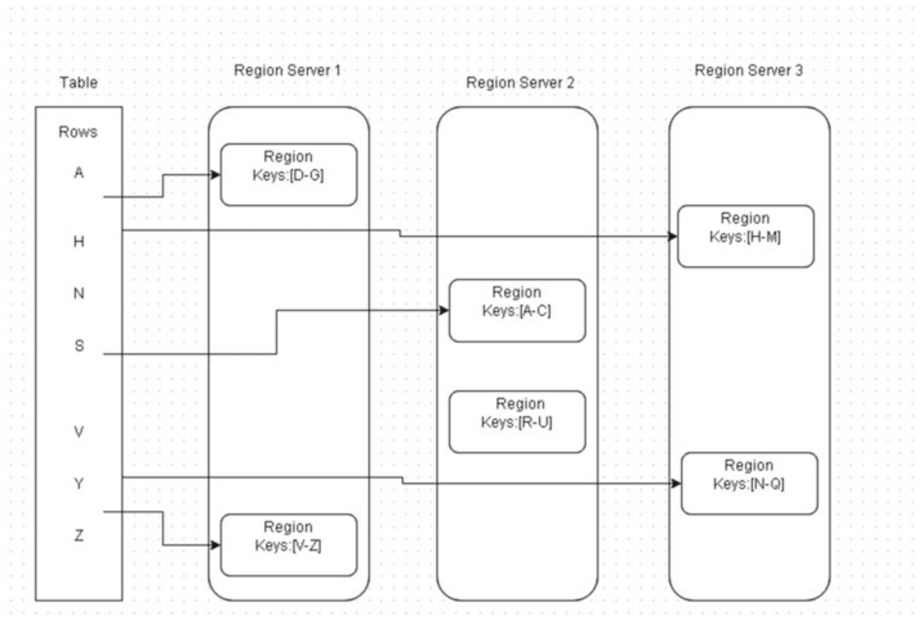


Figure 8-2. *An example of a table's row keys being distributed across three region servers*

A cluster can have one or more RegionServers. RegionServers serve and manage Regions. The RegionServers handle read/write requests from clients. A RegionServer runs on an HDFS datanode. HRegionServer is the Java interface for RegionServer. The RegionServer is responsible for region operations such as region splitting and region compaction. RegionServer also manages data-oriented operations such as `Get`, `Put`, and `Delete`. The RegionServer runs several background threads including those for minor/major compactions, MemStore flush to StoreFile, and WAL. RegionServers are collocated with the datanodes, providing data locality.

ZooKeeper

The ZooKeeper bootstraps and coordinates the cluster. The ZooKeeper provides shared state information for components of the distributed system. ZooKeeper also provides server failure notifications so that a Master can failover to another RegionServer. The ZooKeeper can be a single node or an ensemble of nodes.

The ZooKeeper is also used to store metadata for operations such as master address and recovery state. The `hbase:meta` table (previously `.META.`) stores a list of all regions in the system. The `hbase:meta` is stored in the ZooKeeper. The active Master has a lease in the ZooKeeper. HBase manages a ZooKeeper ensemble by default, but the ZooKeeper may also be run separately. An odd number (3, 5, 7) of nodes in a ZooKeeper ensemble is recommended because it will tolerate more node failures. An ensemble of 2n+1 nodes tolerates n failures. For example, an ensemble of 7 nodes will tolerate 3 failures while an ensemble of 6 nodes will tolerate 2 failures. To start an HBase cluster, a ZooKeeper should be started first, followed by a Master, followed by RegionServers.

Regions

In HBase, data is stored in a table. A table is sorted by row key lexicographically. A table is made up of one or more regions. A region is the unit of scalability in HBase. A region has a startKey and a endKey and contains a sorted, contiguous range of rows. A complete table is not necessarily stored on the same region or even the same RegionServer. Each region can be on a different node and may consist of several HDFS files and blocks, each of which is replicated. Regions are spread randomly across RegionServer/s. Regions can be moved around for load balancing. Regions split automatically or manually with growing data. Capacity is a factor only of cluster nodes vs. regions per node. Regions are the basic units of data distribution and availability in tables. A region is made up of a store per column family. The hierarchy of components including the region is table ➤ region ➤ store ➤ MemStore and store file ➤ block.

When HBase starts, the Master assigns regions to RegionServers. If required for load balancing, the Master also reassigns regions across the RegionServers.

Write-Ahead Log

Inserts are done in the write-ahead log (WAL) first. The WAL records all data changes (`Puts` and `Deletes`) to file-based storage. The WAL is a backup for when a RegionServer crashes. Under normal operation, data stored in a WAL is not used because data is first stored in MemStore and subsequently flushed to an HFile. But, if a RegionServer crashes, the data changes are replayed from the WAL.

The WAL is essential for a data change to complete successfully. If a write to a WAL fails, the whole data write fails. Usually one instance of WAL runs on each RegionServer. In older versions of HBase (<=0.94), HLog was used for a WAL. WAL stores data on HDFS just as the HFile data files. WAL stores in the `/hbase/WALs/` directory for HBase 0.94 and later and in `/hbase/.logs` directory for HBase prior to 0.94. Subdirectories per region are created in WAL.

Store

HBase is based on log-structured merge-trees (LSM trees). A store is created per column family per region. A store stores data and is made up of a single MemStore and 0 or more StoreFiles. A StoreFile is a façade on an HFile, which is stored in HDFS on disk. StoreFiles are made of blocks. The block size is configured per column family. HBase supports compression and encoding, which occur at the StoreFile block level. Data modifications are first stored in memory (MemStore) and flushed to disk on regular intervals, or if a memory threshold is exceeded, or explicitly with a shell command. Each flush generates an HFile. Small flushes are merged in the background by a process called compaction to keep the number of files small. Having a small number of files provides the advantage of faster lookup. HBase has two in-memory structures: MemStore and block cache. While MemStore is for the Write path, the block cache is for the Read path. Reads read block cache first, and if the requested data is not found, the HFile on the disk is read. A block cache is provided for frequently read data and reduces the read latency. Regardless of the number of columns, atomicity is provided at row level.

HDFS

HDFS is the storage layer of HBase. Each RegionServer is collocated with an HDFS Datanode, which is usually on the same node as the RegionServer.

Clients

Clients include native Java API, Gateway for REST, Thrift, and Avro.

Summary

In this chapter, I discussed the major components of an Apache HBase cluster, which are Master, RegionServers, and ZooKeeper. In the next chapter, I will discuss regions.

CHAPTER 9

Regions

In Apache HBase data is stored in a table but a table is not the fundamental unit. HBase is designed for big data and a single table would be unwieldy to store big data. A table is sorted by a row key lexicographically. To store data of scale a table is made up of one or more regions; in fact, it's typically several regions spread across several RegionServers. A region is a subset of a table's data. When a table is created, it consists of only one region by default. The whole capacity of the cluster could be said to be underutilized when data is first loaded into an empty table because the data is sent to a single RegionServer.

A region is the unit of horizontal scalability in HBase. A region has a *startKey* and an *endKey* and contains a sorted, contiguous range of rows. Regions are non-overlapping; the same row key is not stored on multiple regions. HBase guarantees strong consistency within a single row by hosting a region on only a single RegionServer at a time. A complete table is not necessarily stored on the same region or even the same RegionServer. Each region can be on a different node and may consist of several HDFS files and blocks, each of which is replicated. Regions are spread randomly across RegionServer/s. Regions are made available to clients by RegionServers. Regions are the physical mechanism used to distribute the write and query load across the RegionServers. Regions can be moved around for load balancing and failover. Regions split automatically or manually with growing data. Capacity is a factor only of cluster nodes vs. regions per node.

Regions are the basic units of data distribution and availability in tables. A region is made up of a *store* per column family. The data for each column family is stored and accessed separately. The hierarchy of components including the region is table ➤ region ➤ store ➤ MemStore and store file ➤ block.

When HBase starts, the Master assigns regions to RegionServers. If required for load balancing, the Master also reassigns regions across the RegionServers.

RegionServers manage data stored in regions. A table's row keys are distributed across the cluster stored on different regions on different RegionServers, as shown in Figure 8-2 in Chapter 8. Figure 8-2 shows regions only for one table, but a RegionServer typically holds regions from several tables.

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_9

How Many Regions?

It is recommended to use a small number (20-200) of medium-large sized (5-20GB) regions per region server. The optimal number of regions is 100. The following are some of the factors to consider:

1. The available heap space is a limiting factor in selecting the number of regions. Approximately 2MB is required per MemStore, a MemStore being per column family per region. With 100 regions and 3 column families per region, the MemStore heap space requirement is 600MB. Having fewer regions reduces the MemStore heap requirement.
2. A large number of regions generates a large number of tiny flushes; with each flush generating a StoreFile, a large number of StoreFiles are generated, which in turn require more compactions. Also, the `MemStore` and the `StoreFile` index require more heap space.
3. A large number of regions put a load on the Master because the Master has to assign/reassign regions to RegionServers. Also, the Master has to move the regions around for load balancing.

Compactions

When the number of StoreFiles in a store gets too large, the RegionServer performs a compaction to merge the StoreFiles into a smaller number of StoreFiles.

Region Assignment

The regions are assigned on the HBase start by the Master as follows:

1. The Master invokes the assignment manager
2. The assignment manager refers to the existing assignment in `hbase:meta` metadata
3. If the RegionServer is still available, the assignment is kept
4. If the RegionServer is not online, the load balancer is invoked to assign the region to another RegionServer
5. The `hbase:meta` metadata is updated with the new assignment

Failover

With multiple RegionServers serving data, a RegionServer could fail and the regions on the RegionServer could become unavailable. The ZooKeeper detects the RegionServer failure. But, because data is replicated across the cluster, the Master performs a failover to another RegionServer hosting the same set of row keys in a region. Again, regions provide a fundamental unit of the logical data model. Regions are assigned similarly as on startup.

Region Locality

Locality is the closeness of a region to a RegionServer. Region locality is achieved with HDFS block replication across the cluster. A client contacts a RegionServer, and if a region is closer to the RegionServer, less network transfer is required for client operations. Region/RegionServer locality is achieved with block replication. The replica placement policy is the replica placement policy of HDFS, which is as follows:

1. The first replica is on the local node
2. The second replica is on a random node on another rack
3. The third replica is on the same rack as the second but on a different node
4. Subsequent replicas are on random nodes in the cluster

On failover, region locality may be compromised temporarily as RegionServer/s are reassigned regions with non-local StoreFiles, but as data is added to a region or on compaction, the StoreFiles are rewritten and become local to the RegionServer.

The benefits of regions are distributed datastores, partitioning, auto sharding and scalability, and region splitting.

Distributed Datastore

The HBase design of using multiple regions for a table is inline with the design of a distributed datastore. Instead of storing one large table (even if replicated) over one or a few nodes, replicas of a table's regions are distributed evenly across a cluster of nodes. Replication of regions provides high availability.

Partitioning

Regions provide partitioning of data. Multiple clients accessing a table can use different regions of the table and as a result won't overload a partition (region) and RegionServer. Having multiple regions reduces the number of disk seeks required to find a row, and data is returned faster (low latency) to a client request.

Auto Sharding and Scalability

When the number of row keys in a region becomes too large, the region splits into approximately two equal halves, a process called *auto-sharding*. The basic unit of horizontal scalability in HBase is a region. Rows are shared by regions. A region is a sorted set consisting of a range of adjacent rows stored together. A table's data can be stored in one or more regions. When a region becomes too large, it splits at the middle row key into approximately two equal regions. For example, in Figure 9-1, the Region has 12 rows and it splits into two Regions with 6 rows each.

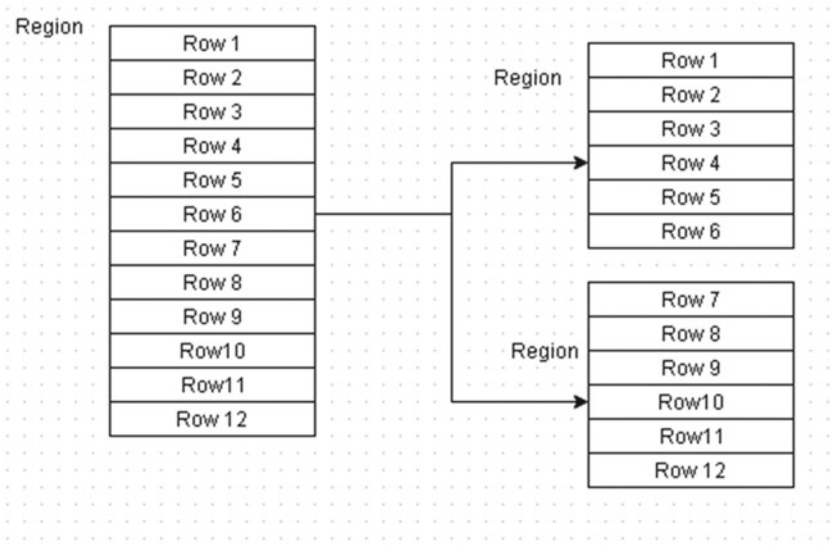


Figure 9-1. *A region split into two regions*

Region Splitting

Regions split when a threshold is exceeded. Splits are handled by the RegionServer, which splits a region and offlines the split region. Subsequently, the two split regions are added to `hbase:meta` and opened on the RegionServer and reported to the Master. Region splitting is automatic by default but may be run manually also. The HBase region splitting policy is configured in `hbase.regionserver.region.split.policy`. The default split policy before 0.94 is `org.apache.hadoop.hbase.regionserver.ConstantSizeRegionSplitPolicy`, which is based on the maximum configurable size. The maximum configurable size is set in `hbase.hregion.max.filesize`. When the sum of the sizes of a Region's StoreFiles exceeds the `hbase.hregion.max.filesize` setting, the region is split. The default value of `hbase.hregion.max.filesize` is 10GB. Since 0.94, the default policy is `IncreasingToUpperBoundRegionSplitPolicy`, which is based on the algorithm of "the split size is the number of regions that are on this server that all are of the same table, cubed, times twice the region flush size OR the maximum region split size, whichever is smaller." The `hbase.regionserver.regionSplitLimit` setting limits the number of regions splits, which is a guideline and not a hard limit. The default is 1,000 regions.

Manual Splitting

By default, automatic splitting is configured and recommended but manual splitting may be used by first disabling automatic splitting by setting `hbase.hregion.max.filesize` to a very large value such as 100GB, which is unlikely to be reached. Manual splitting may be performed with pre-splitting or later. The following are some of the reasons to perform manual splitting:

1. Data is sorted such that new data is sorted at the end of a table as in a timeseries, which makes one of the RegionServers disproportionately overloaded
2. One of the regions has developed a hotspot due to an unusual or unexpected load
3. New RegionServers are added and manual splitting is required to balance the load quickly
4. A bulk load that could cause the uneven load across regions

All of these issues would also be handled by automatic load balancing when found but even the best of row key designs may not get the same result as manual splitting, and load distribution is fixed faster with manual splitting. The RegionSplitter may be used for manual splitting.

Pre-Splitting

In a new table, the row key range is not known and the row key to split a region is undeterminable. As a result, only a single region is created by default in a new table. But a table can be pre-split into a specified number of regions using pre-splitting. An optimal number of pre-split regions is 10. The `org.apache.hadoop.hbase.util.RegionSplitter` utility is used to create a table with a specific number of pre-splits. Note that pre-splitting could cause heterogeneous load distribution, which degrades performance.

Load Balancing

As regions are located on RegionServers, the data could become unevenly distributed across the cluster, with some RegionServers hosting more regions and data than other RegionServers. The Master performs load balancing by moving regions across the cluster. Regions serve as the fundamental unit of data for transfer. It is easier to move a smaller unit of data than the complete table. The load balancer runs periodically as configured in `hbase.balancer.period` with a default of 30,000 (5 minutes).

Preventing Hotspots

Disproportional traffic to a single region ➤ RegionServer can cause hotspots. Manual region splitting can be used to alleviate hot spots.

Summary

In this chapter, I discussed regions. In the next chapter, I will discuss how a client finds a row in an HBase table.

CHAPTER 10

Finding a Row in a Table

In this chapter I discuss how a row in a Apache HBase table is found. The sequence used to find a row/s is as follows:

1. HBase identifies the file/s (HFiles) that store/s the row/s requested using the metadata
2. Each HFile keeps a block index that identifies the block in the HFile where a row is found
3. After finding the block that could contain a row, HBase scans the block to retrieve all key/value pairs requested, and copy of the key/value pairs is stored in the block cache in memory
4. The requested row is returned to the client
5. Subsequent requests for the same row/s get served from the block cache. Data in the block cache gets dropped using the LRU algorithm when the cache is filled

In subsequent sections, the details of each of these steps will be discussed, answering questions such as how the metadata is stored and how the metadata file is found. Finding a row is a DML operation because it does not involve creating a table. Regions are made available to clients by RegionServers. Clients contact a RegionServer directly for DML operations such as reading and writing data. The RegionServers handle read/write requests from clients. The Master is not involved in finding a row in an HBase table. But, how does a client find which RegionServer stores the row/s of data? The Region ➤ Region Server mapping is stored in the `hbase:meta` catalog table (also known as the `META` table). As `hbase:meta` is also a table, just like any other HBase table, it is stored on a RegionServer. The location of hbase:meta is kept in the ZooKeeper on assignment by the Master. When HBase starts up, the Master assigns the regions to each RegionServer, including the regions for the `hbase:meta` table.

A HBase client communicates with the ZooKeeper and the HRegionServers. The HMaster coordinates the RegionServers. The RegionServers run on the datanodes. The read path for reading data and the write path for inserting data were discussed in Chapter 2.

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_10

Block Cache

As `hbase:meta` is required to access RegionServers, it is kept in the block cache for as long as is feasible. The following flow sequence is used when a new client contacts the ZooKeeper quorum to find a row key:

1. First, the client gets the RegionServer name that hosts the `hbase:meta` table region from the ZooKeeper.
2. The `hbase:meta` RegionServer info is cached and looked up once only.
3. Subsequently, the client queries the `hbase:meta` RegionServer to get the RegionServer that contains the row the client needs.
4. The client caches the information about the region in which the row is located.
5. The client contacts the RegionServer hosting the region directly. The client does not need to contact the `hbase:meta` region server again and again once it finds and caches information about the location of the row/s.
6. The `HRegionServer` class opens the region and creates a `HRegion` object. `HRegionServer` makes a set of `HRegions` available to clients, and a single HBase deployment has several HRegionServers, each of which checks in with the `HMaster`. A store instance is created for each `HColumnFamily` for each table and each of the store instances can have `StoreFile` instances, which are lightweight wrappers around the HFile storage files. Each `HRegion` has a `MemStore` and a WAL instance.

The block cache caches the following information, which is used in finding a row of data:

- Row data: Each `Get` or `Scan` that yields data that is not already in the block cache is added to the block cache.
- Row keys: When a value (key/value) is loaded into block cache, its key is also cached. It is advantageous to make the keys small so that they occupy less space in the cache.
- The `hbase:meta` table, which keeps track of the RegionServer ➤ region mappings, is given in-memory priority and is kept in memory for as long as feasible. The `hbase:meta` table could consume several MB of cache if a large number of regions are defined.
- Block indexes of HFiles are stored in the block cache. Using an index, a client is able to find a row of data without having to open the entire HFile. Index size is a factor of the size of the row keys, block size, and amount of data stored in an HFile.

The hbase:meta Table

The `hbase:meta` table list all the regions and their locations. The region ➤ Region Server association is stored in `hbase:meta`. For example, the `hbase:meta` table shown in Figure 10-1 lists the region ➤ Region Server mappings for `Table1` and `Table2. Table1 startKey Key-00` row is stored in a region with a Region Id of 1, which is mapped to RegionServer `machine01host. Table1` startKey `Key-30` row is stored in a region with a Region Id of 2, which is mapped to RegionServer `machine02host`. Similarly, `Table2` startKey `Key-00` row is stored in a region with a Region Id of 1, which is mapped to RegionServer `machine01host. Table2` startKey `Key-41` row is stored in a region with a Region Id of 2, which is mapped to RegionServer `machine02host`. The table ➤ startKey ➤ RegionId ➤ RegionServer mappings are shown in Figure 10-1.

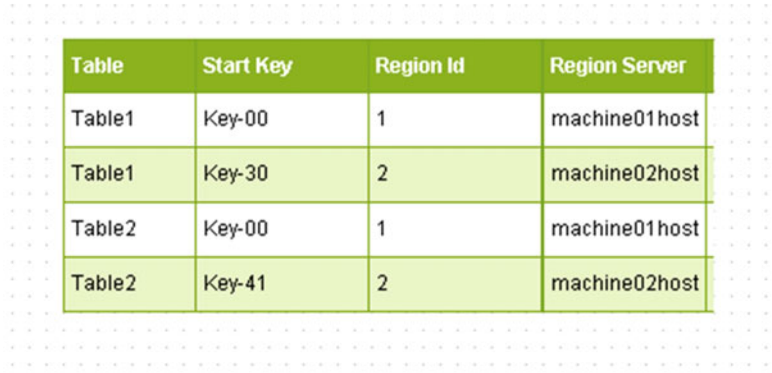


Table	Start Key	Region Id	Region Server
Table1	Key-00	1	machine01host
Table1	Key-30	2	machine02host
Table2	Key-00	1	machine01host
Table2	Key-41	2	machine02host

Figure 10-1. *Row startKey ➤ Region Id ➤ RegionServer host mappings*

It can be found which region is responsible for which key. In read/write operations, the Master is not involved at all and the client goes directly to the RegionServer responsible for serving the requested data.

For `Put` and `Get` operations, clients don't have to contact the Master and can directly contact the RegionServer responsible for handling the specified row. For a client scan, the client can directly contact the RegionServers responsible for handling the specified set of keys. The client queries the `hbase:meta` table to identify a RegionServer, `hbase:meta` being a system table used for tracking the regions. The `hbase:meta` table contains the RegionServer names, region identifiers (Ids), table names, and startKey for each region. For the `hbase:meta` table shown in Figure 10-1, the RegionServer ➤ Region mappings are shown in Figure 10-2. RegionServer `machine01.host` is mapped to regions 1 and 20, and Region Server `machine02.host` is mapped to regions 2 and 21. By finding the startKey and the next region's startKey, clients can identify the range of rows in a particular region. The client contacts the Master only for creating a table and for modifications and deletions. The cluster can keep serving data even if the Master goes down.

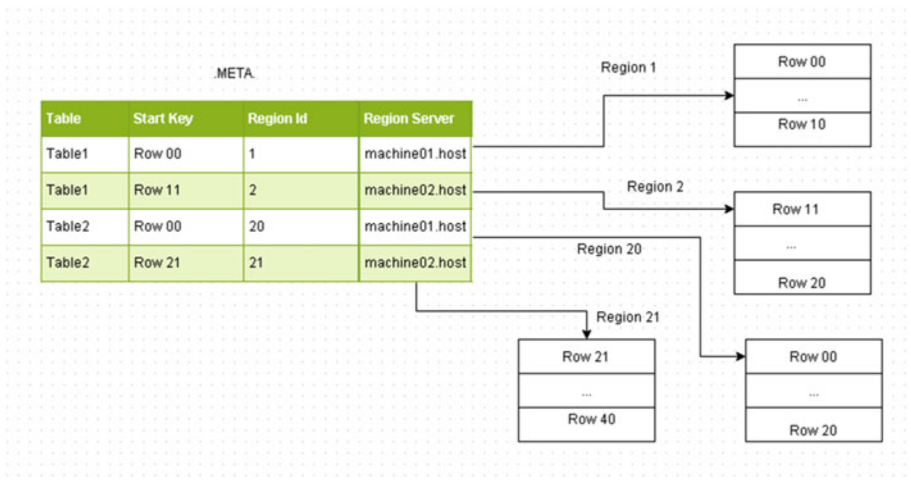


Table	Start Key	Region Id	Region Server
Table1	Row 00	1	machine01.host
Table1	Row 11	2	machine02.host
Table2	Row 00	20	machine01.host
Table2	Row 21	21	machine02.host

Figure 10-2. *RegionServer ➤ region mappings*

To avoid having to get the region location again and again, the client keeps a cache of region locations. The cache is refreshed when a region is split or moved to another RegionServer due to balancing or assignment policies. The client receives an exception when the cache is outdated and cache is refreshed by getting updated information from the `hbase:meta` table.

The `hbase:meta` is also a table like the other HBase tables and the client has to find from the ZooKeeper on which RegionServer the `hbase:meta` itself is located. Prior to HBase 0.96, the META locations were stored in a table called `-ROOT-`. With HBase 0.96, `-ROOT-` has been removed and the meta locations are stored in ZooKeeper, as shown in Figure 10-3.

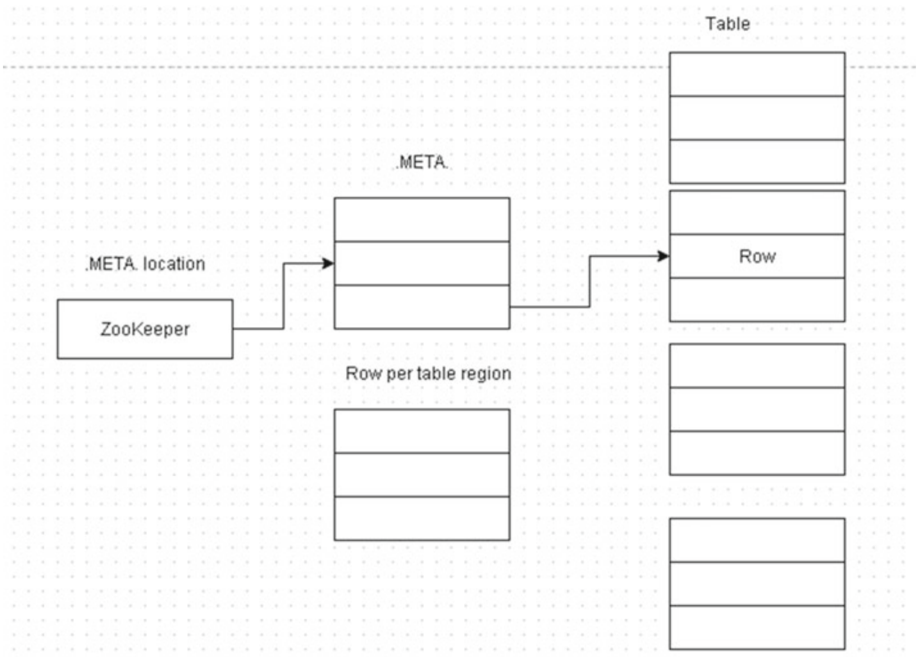


Figure 10-3. *The hbase:meta location is in the ZooKeeper*

The HBase Java Client API provides the two main interfaces discussed in Table 2-6 in Chapter 2.

Summary

In this chapter, I discussed how a client finds a row in a table. In the next chapter, I will discuss compactions.

CHAPTER 11

Compactions

Compactions were introduced in Chapter 2. In this chapter, I will discuss compactions in more detail. Each flush of the `MemStore` generates a `StoreFile`, which is façade on an `HFile`. The MemStore size at which a flush is performed is set in `hbase.hregion.memstore.flush.size`, which is 128MB by default. The MemStore size is checked at a frequency set in `hbase.server.thread.wakefrequency`, which has a default of 10,000 ms. If the number of StoreFiles in a store exceed some limits/thresholds, the files are compacted into larger StoreFiles. Compaction is the process of creating a larger StoreFile (HFile) file by merging smaller `StoreFile` files. Compaction asynchronously reads the smaller StoreFiles and rewrites the StoreFiles into a single StoreFile. Compactions do not merge regions. Old files are removed after merging. Compaction is performed on a per-region basis. Compactions are performed to improve read performance. Compaction could become necessary if HBase has scanned too many StoreFile files to find a result but is not able to find a result. After the number of files scanned exceeds the limit set in the `hbase.hstore.compaction.max` parameter, compaction is performed to merge files to create a larger StoreFile file. Instead of searching multiple files, only one StoreFile file has to be searched.

Two types of compactions are performed: *minor compaction* and *major compaction*. Major compaction merges all the files. In a major compaction, deleted and duplicate key/values are removed. When the schema updates to tables or column families are made, such as changes to region size or block size, the updates take effect on the next major compaction when the StoreFile gets rewritten.

Minor Compactions

Minor compaction just merges two or more smaller StoreFile files into one larger StoreFile file. As a result, a store has a fewer number of StoreFiles. Minor compactions select a small number of adjacent StoreFiles for compactions. Configuration properties (discussed in Table 11-1) `hbase.hstore.compaction.min`, `hbase.hstore.compaction.max`, `hbase.hstore.compaction.min.size`, `hbase.hstore.compaction.ratio` effect minor compactions. Minor compactions do not drop deletes or excess or expired versions.

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_11

Major Compactions

Major compaction merges all the StoreFile files in a store into a single StoreFile. In a major compaction, deleted and duplicate key/values are removed. Major compactions run automatically at a frequency set in a configuration property called `hbase.hregion.majorcompaction`, which has a default value of 7 days (>=0.96). Before 0.96, frequency for major compactions was once a day. Setting `hbase.hregion.majorcompaction` to 0 disables major compactions. It is not recommended to disable major compactions. An explicit major compaction may also be performed by a user with Admin permissions. Major compactions may be requested through the HBase shell and with the `Admin.majorCompact` API.

A major compaction could degrade the read performance temporarily during the major compaction because the underlying filesystem is in flux. As a result, a client may experience high latency and even request timeouts.

Compaction Policy

How is it determined whether a compaction is to be a minor compaction or a major compaction and which StoreFiles to select for compaction? It is determined by a *compaction policy*. Prior to HBase 0.96, the `RatioBasedCompactionPolicy` was used, in which the first set of StoreFiles for the selection criteria are compacted. Since 0.96, the default compaction policy is the `ExploringCompactionPolicy`, which tries to select the best possible set of StoreFiles with the least amount of work. For example, the `ExploringCompactionPolicy` may determine that a minor compaction is more beneficial than a major compaction. The StoreFile selection is based on several configuration parameters.

The objective of the `ExploringCompactionPolicy` is to find the best compaction set. The `ExploringCompactionPolicy` algorithm is as follows:

1. A list of all existing StoreFiles in a store is made
2. For a user-requested compaction, an attempt is made to perform the compaction. But it may not be feasible to perform the requested compaction. Not all the StoreFiles may be available to compact or a store could have too many StoreFiles
3. The number of StoreFiles is reduced. StoreFiles selected for exclusion include those larger than `hbase.hstore.compaction.max.size` and StoreFiles created with bulk load and excluded with `hbase.mapreduce.hfileoutputformat.compaction.exclude`
4. After excluding some StoreFiles, restart from step 1 and make a list of potential sets of StoreFiles to compact. A potential set is made from `hbase.hstore.compaction.min` number of contiguous StoreFiles in the list. A set with number of StoreFiles fewer than `hbase.hstore.compaction.min` or more than `hbase.hstore.compaction.max` would not be a potential set for compaction

5. Determine if the size of a set of StoreFiles is the smallest feasible compaction size and, if so, store the set as a "fall-back" set that could be used for compaction if the compaction algorithm gets set and can't find any other set suitable for compaction

6. If not selected as a fall-back, perform some more validation to find if the StoreFiles in the set are suitable for compaction. If a StoreFile is larger than `hbase.hstore.compaction.max.size` or `hbase.hstore.compaction.min.size x hbase.hstore.compaction.ratio`, exclude the StoreFile from the set

7. A set is still in contention for being selected the best compaction set if it has only one StoreFile or if its `size x hbase.hstore.compaction.ratio` (or `hbase.hstore.compaction.ratio.offpeak` for offpeak compaction) is less than the sum of all the other StoreFiles in the set. Compare the best compaction with the previously selected best compaction and select the new best compaction set. After the entire list of potential compactions has been processed, perform compaction on the best compaction set. If none is found, perform compaction on the fall-back set

Function and Purpose

The function of compactions is to reduce the number of StoreFiles in a store. Compaction has the following purposes or benefits:

1. Compaction provides better indexing of data, reducing the number of seeks required to reach a block that could contain the key

2. Compactions reduce the number of files to search during a scan by merging smaller files into one large file

3. Read latency is reduced as fewer StoreFiles have to be searched. HBase is designed for fast random access

4. Compactions remove the duplicated keys (updated values)

5. Compactions remove the deleted keys

6. Excess versions are removed on a major compaction

7. StoreFiles with only expired rows are removed on a minor compaction

8. Encryption of HFiles

9. Rotation of the data key

Compactions do have some disadvantages:

1. HFiles compaction alleviates the read performance issue by compacting multiple smaller files into a larger file. But compaction is usually performed in parallel with other requests and it could block writes on a RegionServer. The read performance gain is at a loss of write performance, which could be a factor in a write-intensive workload.
2. The index size and as a result the memory required to store the index increases. The index size is proportional to the size of a StoreFile.

Versions and Compactions

The maximum number of cell versions may be configured on a column family, with a default of three. When the number of versions exceeds the maximum setting, the excess versions are not included when the StoreFiles are rewritten on a major compaction. The excess version/s could affect query results until a major compaction is run to remove the excess version/s. For example, the number of versions is 3: v1, v2, and v3, with v3 being the latest and the maximum number of versions is two. Before a major compaction has been run, one of the latter versions is deleted; for example, v3 or v2 is deleted. With two versions, version v1 gets returned in a query.

Delete Markers and Compactions

All delete markers including those for future timestamps are purged on a major compaction unless `KEEP_DELETED_CELLS` is set to true. If the `hbase.hstore.time.to.purge.deletes` (ms) configuration property is set to a non-default value, other than 0, the delete markers are kept until their timestamp and the `hbase.hstore.time.to.purge.deletes`.

Expired Rows and Compactions

A StoreFile could contain only expired rows. Expired rows get deleted on a minor compaction. Setting `hbase.store.delete.expired.storefile` to false or setting minimum or versions to other than 0 disables the feature of deleting StoreFiles with only expired rows.

Region Splitting and Compactions

HBase regions split automatically if the HFile data files storage exceeds the limit set by the `hbase.hregion.max.filesize` setting in the `hbase-site.xml/hbase-default.xml` configuration file. The default setting for `hbase.hregion.max.filesize` is 10GB. When the default storage requirement for a region exceeds the `hbase.hregion.max.filesize` parameter value, the region splits into two and reference files are created in the new regions. The reference files contain information such as the key where the region was split. The reference files are used to read the original region data files. When compaction is performed, the new data files are created in a new region directory and the reference files are removed. The original data files in the original region are also removed. The `HConstants.MAJOR_COMPACTION_PERIOD` setting determines how often a region performs major compaction with a default value of 7 days. If regions are split into too many large regions, increase the value of `HConstants.MAJOR_COMPACTION_PERIOD`. Each /hbase/table directory also contains a `compaction.dir` directory, which is used when splitting and compacting regions.

Number of Regions and Compactions

Region count should be kept low (20-200 of 5-20GB). Too many regions cause tiny flushes, generating a large number of StoreFiles, causing compactions.

Data Locality and Compactions

HBase eventually achieves locality for a region after a flush or compaction. In a RegionServer failover, data locality may be lost if a RegionServer is assigned regions with non-local HFiles, resulting in none of the replicas being local. But as new data is written in the region or a table is compacted and HFiles are rewritten, they will become local to the RegionServer.

Write Throughput and Compactions

Frequent compactions could affect the write throughput and therefore for write-intensive workloads less-frequent compactions are recommended. Less-frequent compactions would result in more store files per region. Setting `hbase.hstore.compaction.min` to a higher value would reduce compaction frequency by making store files eligible for compaction at a greater size and resulting in more store files. So that updates are not blocked when the number of store files becomes too large, increase the size of the `hbase.hstore.blockingStoreFiles` setting and reduce the `hbase.hstore.blockingStoreFiles` setting.

Encryption and Compactions

HBase supports encryption of HFiles and WALs using the AES encryption algorithm, and encryption may be configured in the schema per column family. Any HFile written after configuring encryption is encrypted. To encrypt all HFiles, perform an explicit major compaction after configuring column family/ies for encryption. A major compaction causes rewriting of HFiles, and the rewritten HFiles are in an encrypted form. Similarly, a data key may be rotated by configuring the column family with a new data key and subsequently causing a major compaction, which causes HFiles to be written with the new data key.

Configuration Properties

Optimal compaction settings are based on latency requirements and read/write volumes. Most of the compaction configuration properties are discussed in Table 11-1.

Table 11-1. *Compaction Configuration Properties*

Configuration Property	Description	Default
`hbase.hregion.memstore.flush.size`	The MemStore size above which it is flushed to disk as a StoreFile. Default is 128 MB.	134217728 bytes
`hbase.hregion.majorcompaction`	Time between major compactions in ms. Setting to 0 disables time-based major compactions. User-requested and size-based major compactions are not affected if set to 0. Value is multiplied by `hbase.hregion.majorcompaction.jitter` to cause the compaction to start within a window of time.	604800000 (7 days)
`hbase.hregion.majorcompaction.jitter`	Multiplier factor to affect the `hbase.hregion.majorcompaction` setting. Creates a window of time in which the compaction runs.	0.50
`hbase.hstore.compactionThreshold`	The number of StoreFiles in a store after which a compaction is run to combine all StoreFiles into one StoreFile.	3
`hbase.hstore.flusher.count`	Number of flush threads run simultaneously. More threads cause more StoreFiles, which could start a compaction sooner.	2

(*continued*)

Table 11-1. (*continued*)

Configuration Property	Description	Default
hbase.hstore.blockingStoreFiles	Number of StoreFiles in a store after which updates to the store are blocked until a compaction is performed or until `hbase.hstore.blockingWaitTime` is exceeded. A blocked situation could arise in which the `hbase.hstore.blockingStoreFiles` does not permit an update because the number of StoreFiles exceeds the setting and compaction must be performed and the `hbase.hregion.memstore.flush.size` has also been exceeded in MemStore, requiring a flush. Such a block is alleviated by the compaction policy.	10
`hbase.hstore.blockingWaitTime`	The time (ms) for which updates to a store are blocked on reaching the `hbase.hstore.blockingStoreFiles` limit of StoreFiles even if the compaction started due to reaching the `hbase.hstore.blockingStoreFiles` limit has not completed.	90000
`hbase.hstore.compaction.min`	The minimum number of StoreFiles that must be eligible for minor compaction before a compaction is run. The objective of tuning the setting is to avoid causing several tiny StoreFiles to be generated before a compaction is run and avoid running a minor compaction every two files. Default setting should suffice.	3

(*continued*)

Table 11-1. (*continued*)

Configuration Property	Description	Default
`hbase.hstore.compaction.max`	The maximum number of StoreFiles selected for a single minor compaction regardless of the number of StoreFiles eligible for compaction. Indirectly it controls how long a minor compaction runs.	10
`hbase.hstore.compaction.min.size`	The minimum size for a StoreFile, below which it is always eligible for minor compaction. For StoreFiles of the size exactly or larger, the `hbase.hstore.compaction.ratio` is used to determine if they are eligible. If too many relatively small (1-2MB) StoreFiles are being generated in a write-heavy environment, the default setting may be too large as multiple compactions would be required to bring a StoreFile size above the minimum size. Reducing the setting would make the minimum size close to the size of StoreFiles being generated and multiple compactions would not be required for a StoreFile. Mostly the default setting should be fine.	134217728 bytes (128 MB)
`hbase.hstore.compaction.max.size`	The maximum StoreFile size for a file to be eligible for compaction, and StoreFiles larger than the setting are excluded from compaction. StoreFiles could be getting excluded from compaction at a relatively lower size and the benefit of compactions might not be being realized. If compactions are occurring too often without much benefit, consider raising the value as it would exclude StoreFiles from compaction at a larger size, resulting in a few, large sized StoreFiles that do not get compacted often.	LONG.MAX_VALUE 9223 Petabyte

(*continued*)

Table 11-1. (*continued*)

Configuration Property	Description	Default
hbase.hstore.compaction.ratio	For StoreFiles of the size exactly or larger than the hbase.hstore.compaction.min.size setting, the hbase.hstore.compaction.ratio is used to determine if they are eligible for a minor compaction. Setting the value too large, such as 10, causes large StoreFiles that were generated using minor compactions because the StoreFiles of size about 10 times the hbase.hstore.compaction.min.size value were still eligible for minor compaction. Setting the value too low would exclude StoreFiles from minor compaction at a relatively small size, producing more relatively small StoreFiles. Raising the value causes compaction of large StoreFiles, which increases the write cost as more data has to be compacted but reduces the read cost as fewer StoreFiles have to be read. Default value of 1.2 should be fine for most purposes.	1.2
hbase.hstore.compaction.ratio.offpeak	Same as hbase.hstore.compaction.ratio, but applies only to off-peak hours.	5.0
hbase.hstore.time.to.purge.deletes	Applies to delete markers with future timestamps. The delete markers are kept through a major compaction that is at a timestamp less than a delete marker's timestamp and get deleted on major compaction that occurs after the delete marker's timestamp plus hbase.hstore.time.to.purge.deletes.	0

(*continued*)

Table 11-1. (*continued*)

Configuration Property	Description	Default
`hbase.regionserver.thread.compaction.throttle`	Two thread pools are created for compactions: one for small compactions and another for large compactions. Separate thread pools are created so that compactions of small tables such as the `hbase:meta` is not slowed due to larger compactions taking up most of the threads. The value of the setting is the threshold for a compaction to be considered a large compaction.	2684354560 (2.5 GB)
`hbase.hstore.compaction.kv.max`	The maximum number of key/value pairs read and write in a batch in a compaction. Lower the value if KeyValues are big, causing `OutOfMemoryExceptions`. Increase the value if rows are wide and small.	10
`hbase.server.compactchecker.interval.multiplier`	By default, events such as MemStore flush determine when a scan is made to check if a compaction is necessary. But, due to infrequent writes it may be necessary to perform a check if a compaction is required at a specified interval. The `hbase.server.thread.wakefrequency` setting multiplied by `hbase.server.thread.wakefrequency` determines the frequency at which a check is performed for compaction.	1000

Pluggable compactions, which make use of different algorithms, may also be used, such as compactions based on statistics (which keys/files are commonly accessed and which are not).

Summary

In this chapter, I discussed compactions. The topics covered including the different kinds of compactions (minor compactions and major compactions), compaction policy, the function of compactions, and versions, delete markers, expired rows, and region splitting in relation to compactions. I also discussed regions, data locality, write throughput, and encryption in relation to compactions. Configuration properties for compactions were also discussed. In the next chapter, I will discuss RegionServer failover.

CHAPTER 12

Region Failover

Apache HBase provides automatic failover on RegionServer crashes. When a RegionServer crashes, the HBase cluster and the data remain available. When a RegionServer crashes, all the regions on the RegionServer migrate to another RegionServer. The Master handles RegionServer failures by assigning the regions handled by the failed RegionServer to another RegionServer.

A RegionServer crash is different from an administrator stopping a RegionServer, which allows for the RegionServer to close the regions and shut down properly and for the Master to reassign the closed regions.

MTTR (Mean Time to Recover) is the average time required to recover from a failed RegionServer. The objective of MTTR for HBase regions metric is to detect failure of a RegionServer and restore access to the failed regions as soon as possible.

The Role of the ZooKeeper

The ZooKeeper has the all-important role of detecting RegionServer crashes and notifying the Master so that the Master may perform the failover to another RegionServer. If no RegionServers are failing, there is no actual value to track in the logs of the ZooKeeper. However, since RegionServers do fail, the ZooKeeper is highly available and it is useful for managing the transfer of the queues in the event of a failure. The ZooKeeper coordinates, communicates, and shares state between the Master/s and the RegionServer. The ZooKeeper is a client/server system for distributed coordination and it provides an interface similar to a filesystem, consisting of nodes called znodes, which may contain transient data. When a RegionServer starts, it creates a sub-znode for describing its online state. For example, a sub-znode could be `/hbase/rs/host1`. The active Master registers in the `/hbase/master` znode. A sub-znode used in region assignment/reassignment is the znode for unassigned regions, `/hbase/unassigned/<region name>`.

HBase Resilience

HBase is resilient to failures while being consistent. HBase implements consistency by having a single RegionServer responsible for a region, which is a subset of data. The resilience to failure is implemented in the HDFS, a distributed filesystem.

1. HBase puts table data in HFiles, which are stored in HDFS. HDFS replicates the blocks of the HFiles, three times by default.
2. HBase keeps a commit log called a write-ahead log (WAL), also stored in the HDFS and also replicated three times by default.

Rebuilding a certain RegionServer can take approximatively 10-15 minutes or even more, so even the latest improvements of HBase can only provide timeline-consistent read access using standby RegionServers. This can be a serious problem for sensitive or critical apps.

Phases of Failover

Before I discuss region server failover, I will discuss the write path to HBase, also shown in Figure 12-1. The client contacts the RegionServer directly for a write. The RegionServer is collocated with a datanode. The HBase table data is written to the local datanode and subsequently replicated to other datanodes with three replicas by default. The ZooKeeper keeps a watch on all of the RegionServers.

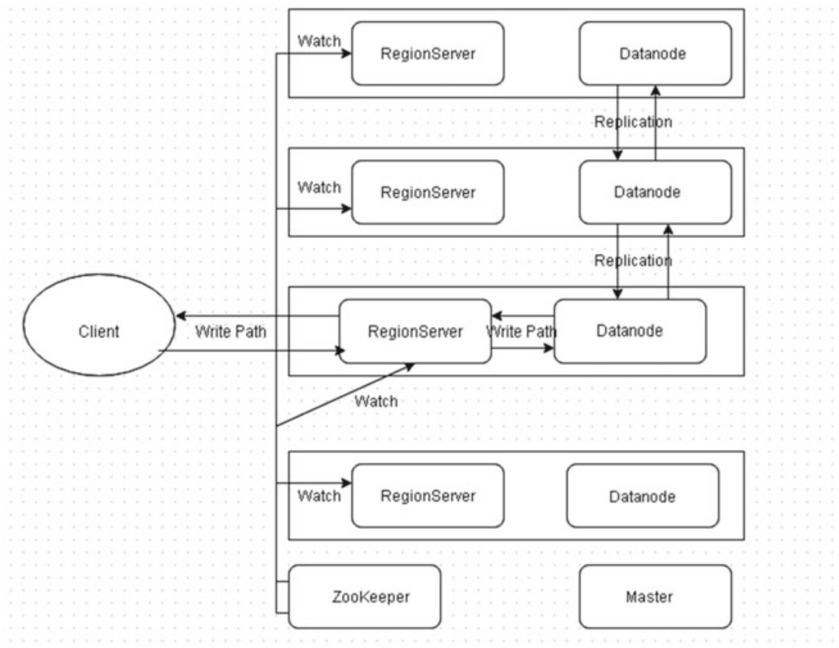


Figure 12-1. *The write path to HBase*

The phases in failover involve failure detection and the recovery process, as follows:

1. Failure Detection: Detect the RegionServer crash.
2. Data Recovery: Recover the writes in progress, which involves reading the WAL and recovering the edits that were not flushed.
3. Regions Reassignment: Reassign/reallocate the regions offlined due to failure to other RegionServer/s.

The three phases (crash detection, data recovery, and region reassignment) are shown in Figure 12-2. The ZooKeeper is shown detecting the RegionServer crash. The ZooKeeper notifies the Master about the RegionServer crash. The Master performs data recovery using the WAL logs. The Master also performs the region reassignment to other RegionServers. The Master notifies the client about the RegionServer failure and the client disconnects from the failed RegionServer. The failover process is shown in Figure 12-2.

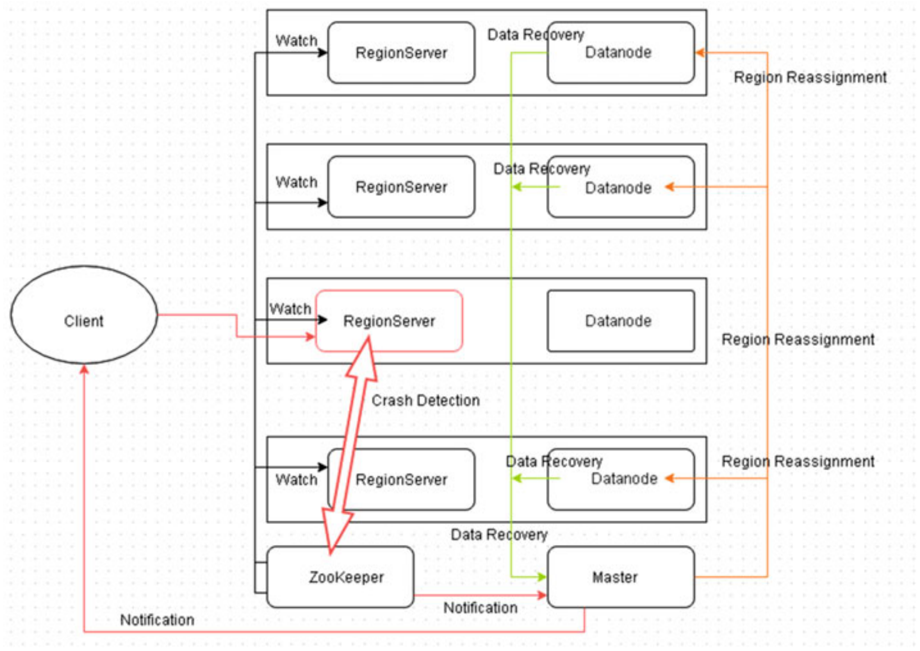


Figure 12-2. *Failover process*

Next, I will discuss these phases in slightly more detail.

Failure Detection

Detecting RegionServer failure due to a crash is performed by the ZooKeeper. Each RegionServer is connected to the ZooKeeper and the Master monitors these connections. When a ZooKeeper detects that a RegionServer has crashed, the ZooKeeper ends the RegionServer's session and notifies the Master about the RegionServer. The Master declares the RegionServer as unavailable by notifying the client. The Master starts the data recovery process and subsequent region reassignment.

The MTTR is influenced by the `zookeeper.session.timeout` setting (default is 90000 ms) in `hbase-default.xml/hbase-site.xml`. If the RegionServer crashes, it could be 90 secs before the ZooKeeper finds out about the crash and times-out the session. But, the ZooKeeper can find out sooner than the configured timeout. The Master finds about the crash from the ZooKeeper and starts the failover to another RegionServer. The Master performs data recovery using the edits stored in the WAL and performs region reassignment. The `zookeeper.session.timeout` may be lowered to reduce the MTTR. A smaller timeout could lead to false positives.

Data Recovery

Data recovery makes use of the edits stored in the WALs. A single WAL consisting of multiple files for all the user regions in a RegionServer is kept. One logical WAL is created per region. One physical WAL is created per RegionServer. WALs are chronologically ordered sets of files and only the last one is open for writing. Every edit is appended to a WAL and sequential writes that sync very well are made to a WAL. Sequential reads for replication and crash recovery are performed.

The Master is able to recover writes in progress from the WAL if a RegionServer crash occurs. The Master reads the edits from the WAL and replays (rewrites) them on another region server, to which region/s have been reassigned. When a RegionServer crashes, the recovery of WAL is started. The recovery is performed in parallel and random RegionServers pick up the WAL logs and split them by edits-per-region into separate files on the HDFS. Subsequently, the regions are reassigned to random RegionServers (not necessarily the same that picked up the WALs) and each RegionServer reads the edits from the respective edits-per-region log split files to recover the correct region state. New data may be written to HBase during WAL replay.

The recovery process is slowed down if it is not just a RegionServer crash, but also the node (machine) on which the RegionServer is running has crashed. As WAL logs are replicated three times, with one of the replicas being on HDFS datanode on the same node (machine) as the RegionServer, 1/3 (33%) of replicas have become unavailable. During data recovery, 33% of reads go the failed datanode first and are redirected to a non-failed datanode. The recovery process has access to only two of the three replicas, which could slow down the recovery process. Having more than the default replicas could alleviate the slow recovery due to machine crash. Also, HBase writes are also written to the crashed datanode and the NameNode has to re-replicate the lost data to bring the replica count to the configured.

Regions Reassignment

The objective is to reassign the regions as fast as possible. The ZooKeeper has an important role in the reassignment. Reassignment is performed by the ZooKeeper and requires synchronization between the Master and the RegionServers through the ZooKeeper.

From these phases, the failure detection takes about 30-90 seconds. Data recovery is about 10 secs and region reassignment is 10 seconds.

After the RegionServer failover is complete, the client connects on a RegionServer to which the data has been recovered and the regions reallocated.

Failover and Data Locality

HBase eventually achieves locality for a region after a flush or compaction. In a RegionServer failover, data locality may be lost if a RegionServer is assigned regions with non-local HFiles, resulting in none of the replicas being local. But as new data written in the region or table is compacted and HFiles are rewritten, they will become local to the RegionServer.

Data locality is low when a region is moved as a result of load balancing or a RegionServer crash and failover. Most of the data is not local unless the files are compacted. When writing a data file, provide hints to the NameNode for locations for block replicas. The load balancer should assign a region to one of the affiliated nodes on a server crash to keep data locality and SSR. Data locality reduces data loss probability.

Configuration Properties

The configuration properties affecting region failover are shown in Table 12-1.

Table 12-1. *Properties Affecting Region Failover*

Configuration Property	Description	Default Value
`zookeeper.session.timeout`	ZooKeeper Session timeout. Increasing the zookeeper session timeout can be a fast first fix, for instance, for garbage collection pauses.	90000
`hbase.regionserver.msginterval`	Interval between messages from RegionServer to Master	3000

Summary

In this chapter, I discussed region failover including the role of the ZooKeeper. HBase is designed to be resilient to failures while being consistent at the same time. The phases of failover, which include failure detection and recovery process, are discussed. In the next chapter, I will discuss creating column families.

CHAPTER 13

Creating a Column Family

Columns, including the column values, are grouped into column families for performance reasons. A column family is both the logical and physical grouping of columns. A column consists of a column family and a column qualifier. A fully qualified column name consists of a prefix, which is the column family name, followed by a : (colon) and the column qualifier. For example, if a table has a single column family `cf1`, which has column qualifiers `col1`, `col2,` and `col3`, the columns in the table would be `cf1:col1`, `cf1:col2`, and `cf1:col3`. The column family name must be composed of printable characters while the column qualifier can be any bytes. Column families must be declared when a table is created, but the column qualifiers may be created on an as-needed basis dynamically. Each row in a table has the same column families even though a column family may not store any data. A table can be defined as a sparse set of rows stored in column families. The maximum number of row versions is configured per column family. All column family members are stored together on disk. Empty cells in a table are not stored at all, not even as null values. As a store is defined per column family, each StoreFile (HFile) stored on disk is per column family, which implies that the data for a column family is stored separately on disk. The storage characteristics of a column family include the following:

1. Are the values cached in memory?
2. How is the data compressed?
3. Are the row keys encoded?

All columns within a column family share the same characteristics such as versioning and compression.

Cardinality

The cardinality of a column family is the number of rows in the column family. If a column family's data is spread across several regions and region servers, the mass scans of the column family become less efficient.

Number of Column Families

The number of column families should be kept low, 2 or 3. HBase presently doesn't perform well on column families above 2 or 3. In general, selecting fewer column families reduces the amount of data to be scanned. Presently, compactions and flushing are performed on a per-region basis, which needlessly flushes and compacts column families that do not need it and introduces unnecessary I/O network load. It is recommended to have one column family; if the second or third column family is added, the query should run on one column family at a time. Having more than a few column families causes several files to be open per region. Several column families also incur class overhead per column family. Several column families could cause compaction storms because StoreFiles are created per column family. Several column families would generate several column families.

Column Family Compression

Column family compression is a best practice and deflates data on disk. But in memory (MemStore), or while being transferred between RegionServer and client, the data is inflated. Therefore, compression does not eliminate the effect of oversized column family, oversized keys, or oversized column names, which are recommended to be kept short. The compression could be `BLOCK` or `RECORD`. The type of compression to use depends on the data used. For example, if table has a single column that stores a blob of text data and only one version is required to be kept, `BLOCK` compression is recommended because it spans multiple rows for the best compression ratio. If a table has variable number of rows containing text data and multiple versions are used, `RECORD` compression is recommended because the compression is applied per record or row. Compression ratios are generally better for `BLOCK` compression, therefore it's recommended for use with blobs of text data. Access times are better for `RECORD` compression because a single row is fetched at a time.

Column Family Block Size

The block size is configurable per column family and is 64k by default. If the cell values are expected to be large, make the block size large. The StoreFile index size is reduced if the block size is large; as a result, a StoreFile index requires less memory.

Bloom Filters

A bloom filter is used to ascertain if a given column exists in a given row. A bloom filter adds an extra index, which incurs a storage overhead in memory and an updata overhead in time. The purpose of a bloom filter is to reduce the lookup time, which makes them especially suitable if a column family has a large number of variably named columns with each cell having a small amount of data. Inserting new items and checking for existing items is speeded up with a bloom filter. Deletion is slowed as it requires rebuilding the index.

IN_MEMORY

The `IN_MEMORY` characteristic of a column family makes cell values to be kept in memory more preferably than normal. `IN_MEMORY` speeds up certain kinds of read/write patterns. The disadvantage is that it consumes more RAM and may interfere with HDFS backups because data is or might be written to disk less frequently. An example of a `blocksize` command when describing table is as follows:

```
BLOCKSIZE => '12345', IN_MEMORY => 'false', BLOCKCACHE => 'true'}]}
```

MAX_LENGTH and MAX_VERSIONS

MAX_LENGHT and MAX_VERSIONS affect the function of a column family. `MAX_LENGTH` is how many bytes can be stored in each cell, with a default of max size of a 32-bit signed integer. If the data to be stored per cell is large, use a higher value. `MAX_VERSIONS` is the maximum number of supported versions with a default of 3.

The main factors to be considered when creating column families are as follows:

1. The access pattern and size characteristics of all members of a column family should be the same because they are stored together.
2. The number of column families should be kept low, preferable 1 and at the most 2 or 3.
3. The column family name must be printable because it is used as directory name in the filesystem; the column qualifier can be any arbitrary bytes.
4. The maximum number of row versions should be set to a very high level, such as hundreds or more.
5. Sorting per column family can be used to convey application logic or access pattern.
6. The column family name and column qualifier name must be kept short because cell coordinates, which are `{rowkey, columnfamily:columnqualifier,timestamp}`, accompany a cell value through the system. The column family and column qualifier names should be in the range of 1-3 characters each.

Summary

In this chapter, I discussed how to create a column family. In the next chapter, I will discuss RegionServer splits.

PART IV

Schema Design

CHAPTER 14

Region Splitting

A table's data is stored in regions. A single table's data can be stored in one or more regions. A region is a sorted set consisting of a range of adjacent rows stored together. A table's data can be stored in one or more regions depending on how many rows are stored in a region. RegionServers manage data stored in regions. When HBase starts, the Master assigns regions to RegionServers. If required for load balancing, the Master also reassigns regions across the RegionServers. As discussed in Chapter 9, when the number of row keys in a region becomes too large, the region splits into approximately two equal halves, and this is called *auto-sharding*. Regions split automatically or manually with growing data as a region becomes too large. A RegionServer does not compact and splits in parallel. For example, a table's row keys are not stored in the same region; a table's row keys are distributed across the cluster stored on different regions on different RegionServers.

The basic unit of horizontal scalability in HBase is a region. Rows are shared by regions. When a region becomes too large, it splits at the middle row key into approximately two equal regions. For example, in Figure 14-1, the region has 12 rows and it splits into two regions with 6 rows each.

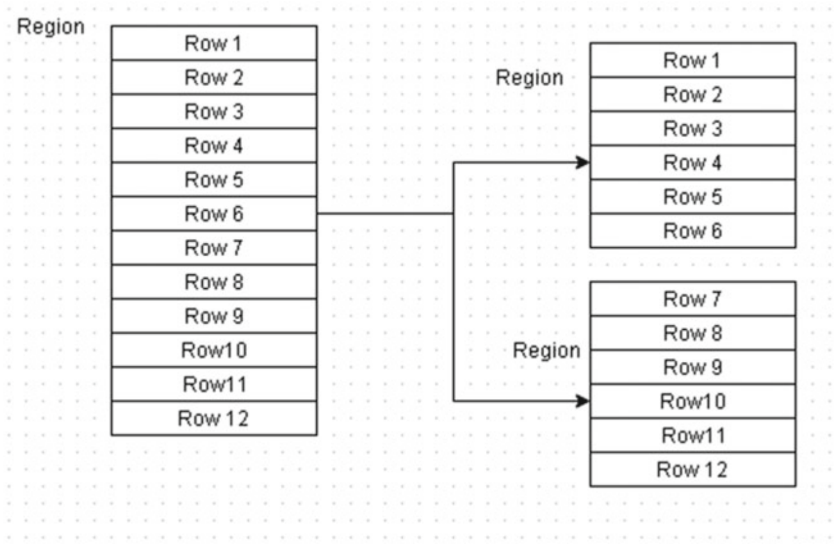


Figure 14-1. *A region splits into two regions at the middle row key*

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_14

Region splitting was discussed in detail in Chapter 9. Compactions and region splits was discussed in Chapter 11.

Managed Splitting

Managed or manual splitting was also introduced in Chapter 9. Managed splitting is recommended only for workloads. By default, automatic splitting is configured and recommended. To use managed splitting, first disable automatic splitting by setting `hbase.hregion.max.filesize` to a very large value that is unlikely to be reached, such as 100GB. Managed splitting may be performed with pre-splitting or later as a rolling split of all regions in an existing table, both with the `org.apache.hadoop.hbase.util.RegionSplitter` utility. Certain workload characteristics benefit from manual splitting.

1. Data (~ 1k) that would grow instead of being replaced.
2. Data growth is roughly uniform across all regions.
3. OLTP workload in which data loss cannot be tolerated.

The following are some of the reasons to perform managed splitting, in addition to the ones discussed in Chapter 9:

1. Data splits are needed with growing amounts of data debugging and profiling, and this is easier with manual splitting. Issues such as data offlining bugs, unknown number of regions, automatically named and constantly renamed regions are some of the issues with automatic regions.
2. The compaction algorithm may be finely tuned. Staggered time-based major compactions can be used to spread out the network I/O load and prevent split/compaction storms when regions reach the same data size at the same time.
3. Region boundaries are known and invariant.
4. Mitigates region creation and movement under load.
5. All of these issues would also be handled by automatic load balancing when found but even the best of row key designs may not get the same result as manual splitting, and load distribution is fixed and faster with manual splitting.

Pre-Splitting

Pre-splitting, introduced in Chapter 9, is the process of creating a table with the specified number of pre-split regions. Ordinarily, in a new table, the row key range is not known and the row key to split a region is undeterminable. As a result, only a single region is created by default in a new table. But a table could be created with a pre-specified number of splits. To prevent compaction storms that occur due to the uniform data growth in a large series of regions resulting in same-sized regions, the optimal number of pre-splits depends on the largest StoreFile in a region. The largest region should be just big enough so that it is compacted only during timed major compaction and an optimal number of pre-split regions is 10. When selecting the number of pre-splits, it is better to select fewer and perform rolling splits later.

Configuration Properties

The configuration properties for region splitting are discussed in Table 14-1.

Table 14-1. *Configuration Properties for Region Splitting*

Configuration Property	Description	Default
`hbase.hregion.max.filesize`	Specifies the maximum cumulative HFile size. If the sum of the sizes of the HFiles in a region exceeds the value, the region is split into two.	10737418240 (10GB)
`hbase.regionserver.region.split.policy`	Specifies a split policy that determines when a region is split. Some of the supported split policies include ConstantSizeRegionSplitPolicy, DisabledRegionSplitPolicy, DelimitedKeyPrefixRegionSplitPolicy, and KeyPrefixRegionSplitPolicy.	`org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy`
`hbase.regionserver.regionSplitLimit`	A guideline and not a hard limit to limit the number of regions after which regions are not split further.	1000
`hbase.client.keyvalue.maxsize`	Specifies the maximum allowable size of a KeyValue instance. An upper size limit for a single entry in a HFile. As a single KeyValue entry cannot be split, the region cannot be split further because the data is too large for a split. Recommended to set to a fraction of the maximum region size. Setting to 0 disables the check.	10485760 (10MB)

The sequence used in region splitting is as follows:

1. The client sends write requests to the RegionServer.
2. The write requests accumulate in memory.
3. When the MemStore is filled and reaches a threshold, the data stored in memory is written to HFiles on disk with a process called memstore flush.
4. As store files accumulate, the region server compacts them to fewer, larger files using a compaction policy.
5. The amount of data in a region grows.
6. The RegionServer consults a region split policy to determine if the region should be split.
7. A region split request is added to a queue.

But how is a region actually split? Is it split at a certain row key? What happens to the split regions? What happens to the region split? The following sequence is used by the RegionServer in a split:

1. When the RegionServer decides to split a region, it starts a split transaction. The RegionServer acquires a shared lock on the table to prevent schema modifications during the split. Next, it creates a znode in ZooKeeper in `/hbase/region-in-transition/region-name` and sets the znode state to SPLITTING. The Master finds about the split process having started as it monitors the `/hbase/region-in-transition` znode.
2. The RegionServer creates a subdirectory called `.splits` in the region directory in HDFS. The splitting region is taken offline. Any request a client sends to a splitting region gets the `NotServingRegionException`.
3. The RegionServer creates subdirectories in the `.splits` directory for the regions to be generated and also creates the data structures.
4. The RegionServer splits the store files to create two reference files per store file in the region to be split. The reference files point to the region-to-be-split's files.
5. The RegionServer creates the region directories in HDFS and transfers the reference files to the region directories.

6. The RegionServer sends a `Put` request to the `hbase:meta` table to set the region-to-be-split offline and information about the new regions to be created. Individual entries for the regions to be created are not yet added to the `hbase:meta` table. Clients that scan the `hbase:meta` table find the region-to-be-split as split but won't find the regions created yet. The znode state of the new regions in ZooKeeper is `SPLITTING_NEW`. If the `Put` to `hbase:meta` succeeds, the region has been effectively split. If the RegionServer fails before the RPC completes successfully, the Master and the next RegionServer opening the region clean the dirty state about the region split. If the `hbase:meta` update completes successfully, the region split is rolled forward by the Master. If the split fails, the splitting region state is made `OPEN` from `SPLITTING` and the two new regions' state are made `OFFLINE` from `SPLITTING_NEW`.
7. The RegionServer opens the two new regions created in parallel.
8. The RegionServer adds the new regions to `hbase:meta` and the new regions are online.
9. Subsequently, clients are able to find the new regions and send requests to them. Client caches are cleared. Clients get information about new regions from `hbase:meta`.
10. The RegionServer updates znode `/hbase/region-in-transition/region-name` in ZooKeeper to the `SPLIT` state. The new regions' states are made `OPEN` from `SPLITTING_NEW`. The Master finds about the states from the znode. The load balancer may reassign the new regions to other region servers if required. The region split process is complete. The references to the old region in `hbase:meta` and HDFS are removed on compactions in the new regions. The Master periodically checks if the new regions still refer to the old region. If not, the old region is removed.

When creating splits with `Admin.createTable(byte[] startKey, byte[] endKey, numRegions)`, the split strategy used is `Bytes.split`.

Summary

In this chapter, I discussed region splitting, including the two kinds of splitting: automatic and managed splitting. I also discussed when managed splitting is a suitable option and on what workloads. Pre-splitting, which creates pre-split regions, was also discussed. Some configuration properties used for region splitting are listed and the procedure used for region splitting was discussed. In the next chapter, I will discuss defining the row keys for optimal read performance and locality.

CHAPTER 15

Defining the Row Keys

The primary data access pattern is by row key. No design-time way to specify row keys exists because to HBase they are simply byte arrays. When designing for optimal read performance, it is important to first understand the read path.

1. The read request is made by a client.
2. HBase identifies the files that store the rows.
3. The block index in each file identifies the block in which the row is found.
4. HBase performs a scan to fetch all the key/value pairs for the request.
5. A copy is stored in the block cache in memory before a row is returned to the client. Block cache stores the data in memory for subsequent reads. The data in the cache gets dropped with a LRU algorithm when the cache gets filled.

When scanning a file block, the following scenarios are possible:

1. No data that satisfies the read request is available.
2. HBase scans multiple blocks if a row spans multiple blocks.

Table Key Design

Table keys should prevent data skew; keys should distribute data storage and processing to all RegionServers for the design to scale well and perform well, making use of all the resources of a cluster. Avoid using monotonically increasing values of time series data as row keys. Such a design could cause a single region to be a hot spot because all the key values are next to each other and belong to the same region. The row key length should be as short as is reasonable, still keeping the key suitable for data access. A tradeoff has to be made between the better `Get`/`Scan` properties of a longer key and keeping the key short.

Filters

In a table scan in which only the row keys are needed (no column families or column qualifiers, or values or timestamps), add a `FilterList` with the `MUST_PASS_ALL` operator to the scanner using the `setFilter`. The filter list should include both the `FirstKeyOnlyFilter` and a `KeyOnlyFilter`.

FirstKeyOnlyFilter Filter

The `FirstKeyOnlyFilter` only returns the first KV from each row. The `FirstKeyOnlyFilter` filter can be used more efficiently to perform row count operations.

KeyOnlyFilter Filter

The `KeyOnlyFilter` only returns the key component of each KV (the value is rewritten as empty). The `KeyOnlyFilter` filter can be used to fetch all of the keys without having to also fetch all of the values. Using the `FirstKeyOnlyFilter/KeyOnlyFilter` combination results in minimal network traffic to the client for a single row. The combination results in a worst-case scenario of a RegionServer reading a single value from a disk.

Bloom Filters

Enabling bloom filters on tables reduces the number of block reads. Some storage overhead is involved but the read performance benefit outweighs it. Bloom filters are generated when an HFile is stored. Bloom filters are stored at the end of each HFile. Bloom filters are loaded into memory. Bloom filters provide a check on row and column levels. Bloom filters can filter entire store files from reads, which is useful when data is grouped. Bloom filters are also useful when many misses (missing keys) are expected during reads.

Scan Time

Blocks storing data required for a query are identified quickly in the order of ~O(3), but scanning the block to fetch the data takes more time, O(n), in which n is the number of key values stored in a block. Therefore, it is important to create tables with an optimal block size, which also utilizes cache optimally. Partial key scans should be used when feasible.

Sequential Keys

HBase stores row keys lexciographically, which provides fast, random lookup given a *startKey* and a *stopKey*. Sequential keys are suitable for read performance. Sequential keys make use of block cache. Sequential keys provide locality. Sequential keys can cause RegionServer hotspotting, which may be alleviated by using one of the following:

- Random keys: Random keys do compromise the ability to fetch given a startKey and a stopKey.

- Salting row keys with a prefix and bucketing row keys across regions: Prefixing row keys provides spread. Numbered prefixes are recommended. Row keys are sorted by prefix first. Row keys of bucketed records are not in one sequence as before but records in each sequence preserve their original sequence. Multiple scans based on the original startKey and stopKey running in parallel scan multiple buckets and merge data. Salted keys provide the best compromise between read and write performance.
- Key field swap/promotion

Based on the access pattern, either use random keys or sequential keys. Random keys are best for random access patterns. Sequential keys are best when the access pattern involves a range of keys. Hashing provides spread but is not suitable for range scans.

Defining the Row Keys for Locality

Locality may be implemented using schema design. HBase stores data lexicographically by row key, which implies that rows with row keys close to each other are stored together. Sequential reads of range of rows is efficient and requires access of fewer regions and region servers. Sequential keys are the most suitable for locality because they fetch data from a single or fewer regions/RegionServers. Sequential keys make use of block cache. Sequential keys can cause RegionServer hotspotting but the issue is alleviated by using salting or splitting regions while keeping them small.

The new key after salting is defined as follows:

```
new_row_key = (++index % BUCKETS_NUMBER) + original_key
```

where

- `index` is the numeric/sequential component of the row ID.
- `BUCKETS_NUMBER` is the number of buckets the new row key is to be spread across. As records are spread, each bucket preserves the sequential notion of original record IDs.
- `original_key` is the original key.

Use bulk import for sequential keys and reads.

In a Webtable, pages in the same domain are grouped together into contiguous rows by reversing the hostname component of the URLs.

Summary

In this chapter, I discussed defining the row keys for optimal performance and locality. In the next chapter, I will discuss the `HBaseAmin` class.

PART V

Apache HBase Java API

CHAPTER 16

The HBaseAdmin Class

The org.apache.hadoop.hbase.client.HBaseAdmin is a Java interface for managing the HBase database table metadata and also for general administrative functions. HBase is used to create, drop, list, enable, and disable tables. HBase is also used to add and drop column families. A HBaseAdmin instance may be created using one of the constructors HBaseAdmin(org.apache.hadoop.conf.Configuration c) or HBaseAdmin(HConnection connection).

```
HBaseConfiguration conf = new HBaseConfiguration();
conf.set("hbase.master","localhost:60000");
HBaseAdmin admin=new HBaseAdmin(conf);
```

An HConnection may be obtained from HConnectionManager as follows:

```
HBaseConfiguration conf = new HBaseConfiguration();
conf.set("hbase.master","localhost:60000");
HConnection connection = HConnectionManager.createConnection(conf);
HBaseAdmin  admin=new HBaseAdmin(connection);
```

Subsequently, HBaseAdmin method/s may be invoked. For example,

```
TableName tableName=TableName.valueOf('test');
HTableDescriptor hTableDescriptor=new HTableDescriptor(tableName);
HColumnDescriptor cf1 = new HColumnDescriptor("cf1".getBytes());
HColumnDescriptor cf2 = new HColumnDescriptor("cf2".getBytes());
hTableDescriptor.addFamily(cf1);
hTableDescriptor.addFamily(cf2);
admin.createTable(hTableDescriptor);
```

HBaseAdmin instances do not override a Master restart. The methods for the different functions of the HBaseAdmin class are discussed in Table 16-1.

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_16

***Table 16-1.** Methods for Different Functions of HBaseAdmin Class*

Function	Methods

Add column	`addColumn(byte[] tableName, HColumnDescriptor column)`
	`addColumn(String tableName, HColumnDescriptor column)`
	`addColumn(TableName tableName, HColumnDescriptor column)`

Create table	`createTable(HTableDescriptor desc)`
	`createTable(HTableDescriptor desc, byte[][] splitKeys)`
	`createTable(HTableDescriptor desc, byte[] startKey, byte[] endKey, int numRegions)`
	`createTableAsync(HTableDescriptor desc, byte[][] splitKeys)`

Delete column	`deleteColumn(byte[] tableName, String columnName)`
	`deleteColumn(String tableName, String columnName)`
	`deleteColumn(TableName tableName, byte[] columnName)`

Compact a table or a column family	`compact(byte[] tableNameOrRegionName)`
	`compact(byte[] tableNameOrRegionName, byte[] columnFamily)`
	`compact(String tableNameOrRegionName)`
	`compact(String tableOrRegionName, String columnFamily)`

Delete table/s	`deleteTable(byte[] tableName)`
	`deleteTable(String tableName)`
	`deleteTable(TableName tableName)`
	`deleteTables(Pattern pattern)`
	`deleteTables(String regex)`

Disable table/s	`disableTable(byte[] tableName)`
	`disableTable(String tableName)`
	`disableTable(TableName tableName)`
	`disableTableAsync(byte[] tableName)`
	`disableTableAsync(String tableName)`
	`disableTableAsync(TableName tableName)`
	`disableTables(Pattern pattern)`
	`disableTables(String regex)`

Enable table	`enableTable(byte[] tableName)`
	`enableTable(String tableName)`
	`enableTable(TableName tableName)`
	`enableTableAsync(byte[] tableName)`
	`enableTableAsync(String tableName)`
	`enableTableAsync(TableName tableName)`
	`enableTables(Pattern pattern)`
	`enableTables(String regex)`

(*continued*)

Table 16-1. (*continued*)

Function	Methods
Find if HBase is running	`checkHBaseAvailable(org.apache.hadoop.conf.Configuration conf)`. Static method

Assign and unassign a region to a region server	`assign(byte[] regionName)`
	`unassign(byte[] regionName, boolean force)`

Run balancer	`balancer()`

Close region	`closeRegion(byte[] regionname, String serverName)`
	`closeRegion(ServerName sn, HRegionInfo hri)`
	`closeRegion(String regionname, String serverName)`

Flush table	`flush(byte[] tableNameOrRegionName)`
	`flush(String tableNameOrRegionName)`

Get online regions	`getOnlineRegions(ServerName sn)`

Get table names	`getTableNames()`
	`getTableNames(Pattern pattern)`
	`getTableNames(String regex)`

Get table regions	`getTableRegions(byte[] tableName)`
	`getTableRegions(TableName tableName)`

Find if the Master is running	`isMasterRunning()`

Find if a table is available	`isTableAvailable(byte[] tableName)`
	`isTableAvailable(byte[] tableName, byte[][] splitKeys)`
	`isTableAvailable(String tableName)`
	`isTableAvailable(String tableName, byte[][] splitKeys)`
	`isTableAvailable(TableName tableName)`
	`isTableAvailable(TableName tableName, byte[][] splitKeys)`

Find if a table is enabled or disabled	`isTableDisabled(byte[] tableName)`
	`isTableDisabled(String tableName)`
	`isTableDisabled(TableName tableName)`
	`isTableEnabled(byte[] tableName)`
	`isTableEnabled(String tableName)`
	`isTableEnabled(TableName tableName)`

List tables and table names	`listTableNames()`
	`listTables()`
	`listTables(Pattern pattern)`
	`listTables(String regex)`

(*continued*)

Table 16-1. (*continued*)

Function	Methods

Run a major compaction	`majorCompact(byte[] tableNameOrRegionName)`
	`majorCompact(byte[] tableNameOrRegionName, byte[] columnFamily)`
	`majorCompact(String tableNameOrRegionName)`
	`majorCompact(String tableNameOrRegionName, String columnFamily)`

Merge regions	`mergeRegions(byte[] encodedNameOfRegionA, byte[] encodedNameOfRegionB, boolean forcible)`

Modify column	`modifyColumn(byte[] tableName, HColumnDescriptor descriptor)`
	`modifyColumn(String tableName, HColumnDescriptor descriptor)`
	`modifyColumn(TableName tableName, HColumnDescriptor descriptor)`

Modify table	`modifyTable(byte[] tableName, HTableDescriptor htd)`
	`modifyTable(String tableName, HTableDescriptor htd)`
	`modifyTable(TableName tableName, HTableDescriptor htd)`

Move a region	`move(byte[] encodedRegionName, byte[] destServerName)`
Offline a region	`offline(byte[] regionName)`
Shutdown HBase cluster	`shutdown()`

Split a table or a region	`split(byte[] tableNameOrRegionName)`
	`split(byte[] tableNameOrRegionName, byte[] splitPoint)`
	`split(String tableNameOrRegionName)`
	`split(String tableNameOrRegionName, String splitPoint)`

Shutdown the Master	`stopMaster()`
Stop a RegionServer	`stopRegionServer(String hostnamePort)`

Find if table exists	`tableExists(byte[] tableName)`
	`tableExists(String tableName)`
	`tableExists(TableName tableName)`

From HBase 0.99.0 onward the HBaseAdmin class is not a client API and is replaced with the org.apache.hadoop.hbase.client.Admin interface. An instance of Admin may be created from a Connection with Connection.getAdmin(). Connection should be unmanged obtained with the org.apache.hadoop.hbase.client.ConnectionFactory.createConnection(org.apache.hadoop.conf.Configuration conf) instance method. The Admin interface has similar methods as HBaseAdmin. The HBaseAdmin class is an internal class from 1.0.

```
Connection connection = ConnectionFactory.createConnection(config);
Admin admin=connection.getAdmin();
```

Summary

In this chapter, I discussed the HBaseAdmin class. In the next chapter, I will discuss the Get Java class.

CHAPTER 17

Using the Get Class

Given a table and row key, you can use the get() operation to return specific versions of that row. The org.apache.hadoop.hbase.client.Get class is used to perform Get operations on a single row. Given a row with row key of row1 in a table named table1, the column value for a column with column family cf1 and column qualifier col1 is obtained as follows.

The org.apache.hadoop.hbase.TableName class represents a table name. Create a TableName instance:

```
TableName tableName=TableName.valueOf('table1');
```

The HBase configuration is represented with the org.apache.hadoop.hbase.HBaseConfiguration class. Create an HBaseConfiguration instance:

```
HBaseConfiguration conf = new HBaseConfiguration();
conf.set("hbase.master","localhost:60000");
```

The org.apache.hadoop.hbase.client.HConnection interface represents a client connection to an HBase cluster. Create an instance of HConnection using the static method createConnection(org.apache.hadoop.conf.Configuration conf) in org.apache.hadoop.hbase.client.HConnectionManager. Supply the HBaseConfiguration instance as the arg.

```
HConnection connection = HConnectionManager.createConnection(conf);
```

The org.apache.hadoop.hbase.client.HTable class is used to communicate with a table. Create an HTable instance using the TableName instance and the HConnection instance.

```
HTable hTable= new HTable(tableName, connection);
```

Alternatively, an HTable instance may be created using the HTable constructor HTable(org.apache.hadoop.conf.Configuration conf, String tableName).

```
HTable hTable= new HTable(conf,"table1");
```

© Deepak Vohra 2016
D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_17

Create a Get instance using the Get(byte[] row) constructor.

```
Get get = new Get(Bytes.toBytes("row1"));
```

The Get class provides several methods to set attributes of the Get operation such as maximum number of versions, the timestamps, column families, and columns. The Get class methods are discussed in Table 17-1.

Table 17-1. *Get Class Methods*

Method	Description	Return Type
addColumn(byte[] family, byte[] qualifier)	Adds column family and column qualifier. Multiple column families and column qualifiers may be set by invoking the method multiple times in succession.	Get
addFamily(byte[] family)	Adds a column family	Get
setCacheBlocks(boolean cacheBlocks)	Sets whether blocks should be cached	void
setMaxResultsPerColumnFamily (int limit)	Sets maximum results per column family	Get
setMaxVersions()	Sets all versions to be fetched	Get
setMaxVersions(int maxVersions)	Sets the maximum number of versions	Get
setRowOffsetPerColumnFamily (int offset)	Sets a row offset per column family	Get
setTimeRange(long minStamp, long maxStamp)	Sets a time range	Get
setTimeStamp(long timestamp)	Sets a timestamp for a specific version	Get

The setTimeStamp(long timestamp) method in Get is used to get versions of columns with a specific timestamp. Set the timestamp on the Get as follows:

```
long explicitTimeInMs = 555;
get=get.setTimeStamp(explicitTimeInMs);
```

Multiple timestamps may be set as follows:

```
long explicitTimeInMs1 = 555;
long explicitTimeInMs2 = 123;
long explicitTimeInMs3 = 456;
get=get.setTimeStamp(explicitTimeInMs1).setTimeStamp(explicitTimeInMs2).setT
imeStamp(explicitTimeInMs3);
```

Get the data from a specified row as follows using the get(Get get) method in HTable. A Result object, which represents a single row result, is returned. To test if a row has columns, use the exists(Get get) method.

```
if(hTable.exists(get))
Result r = hTable.get(get);
```

The key/value pairs in the Result r may be output as follows using the raw() method, which returns a KeyValue[ ]:

```
for(KeyValue kv : r.raw()){
                    System.out.print(new String(kv.getRow()) + " ");
                    System.out.print(new String(kv.getFamily()) + ":");
                    System.out.print(new String(kv.getQualifier()) + " ");
                    System.out.print(kv.getTimestamp() + " ");
                    System.out.println(new String(kv.getValue()));
                 }
```

The HTable class provides the get(List<Get> gets) method to get multiple rows and the exists(List<Get> gets) method to test if columns exist in the rows to be fetched. The return value of exists(List<Get> gets) is an array of Boolean.

Get the latest version of a specified column value using the getValue(byte[] family,byte[] qualifier) method in Result. The cf1:col1 column value is fetched as follows:

```
byte[] b = r.getValue(Bytes.toBytes("cf1"), Bytes.toBytes("col1"));
```

The byte[] may be output as a String.

```
String valueStr = Bytes.toString(b);
System.out.println("GET: " + valueStr);
```

Get all versions of a specified column, like cf1:col1, as follows using the getColumn(byte[] family, byte[] qualifier) method:

```
List<KeyValue> listKV = r.getColumn(Bytes.toBytes("cf1"), Bytes.
toBytes("col1"));
```

The key/value pairs in List<KeyValue> may be output as follows:

```
for(KeyValue kv : listKV){
                    System.out.print(new String(kv.getRow()) + " ");
                    System.out.print(new String(kv.getFamily()) + ":");
                    System.out.print(new String(kv.getQualifier()) + " ");
                    System.out.print(kv.getTimestamp() + " ");
                    System.out.println(new String(kv.getValue()));
                 }
```

The method getColumn(byte[] family, byte[] qualifier) and raw() are deprecated in 0.98.6. In later versions, the getColumnLatestCell(byte[] family, byte[] qualifier) method, which returns a Cell, and the rawCells() method, which returns Cell[], may be used.

The Result class also provides other methods for other purposes or functions, some which are discussed in Table 17-2. Some of these methods, such as getColumnLatestCell and tk, are available in later versions of HBase only.

Table 17-2. *Result Class Methods*

Function	Description	Return Value
Find if a column has a value or is empty.	containsColumn(byte[] family, byte[] qualifier) containsColumn(byte[] family, int foffset, int flength, byte[] qualifier, int qoffset, int qlength) containsEmptyColumn(byte[] family, byte[] qualifier) containsEmptyColumn(byte[] family, int foffset, int flength, byte[] qualifier, int qoffset, int qlength) containsNonEmptyColumn(byte[] family, byte[] qualifier) containsNonEmptyColumn(byte[] family, int foffset, int flength, byte[] qualifier, int qoffset, int qlength)	boolean
Get all cells for a specific column.	getColumnCells(byte[] family, byte[] qualifier)	List<Cell>
Get a specific cell version for a column.	getColumnLatestCell(byte[] family, byte[] qualifier)	Cell
Get the row key.	getRow()	byte[]
Get the latest version of a specified column.	getValue(byte[] family, byte[] qualifier)	byte[]
Get the latest version of a specified column as ByteBuffer.	getValueAsByteBuffer(byte[] family, byte[] qualifier)	ByteBuffer
Find if cell is empty.	isEmpty()	boolean
Get the value of the first column.	value()	byte[]
Get the array of cells.	rawCells()	Cell[]

Summary

In this chapter, I discussed the Get class. In the next chapter, I will discuss the HTable Java class.

CHAPTER 18

Using the HTable Class

The `checkAndPut()` method in the `HTable` class is used to put data in a table if a row/ column family/column qualifier ➤ value matches an expected value. If it does, a new value specified with a `Put` is put in the table. If not, the new value is not put. The method returns true if the new value is put and returns false if the new value is not put. Before discussing the `checkAndPut()` method, however, let's discuss the `put(Put put)` method, which puts data without first performing a check.

```
Configuration conf = HBaseConfiguration.create();
        HTable table = new HTable(conf, "table1");
        Put put = new Put(Bytes.toBytes("row1"));
        put.add(Bytes.toBytes("cf1"), Bytes.toBytes("col1"),
                Bytes.toBytes("val1"));
        put.add(Bytes.toBytes("cf2"), Bytes.toBytes("col2"),
                Bytes.toBytes("val2"));
        table.put(put);
```

Now let's discuss the `checkAndPut(byte[] row, byte[] family, byte[] qualifier, byte[] value, Put put)` method. For example, let's put a new value called `val2` in `row1`, column family `cf1`, column `col1` using a `Put` instance if the `row1`, column family `cf1`, column `col1` has value `val1`:

```
Configuration conf = HBaseConfiguration.create();
        HTable table = new HTable(conf, "table1");
        Put put = new Put(Bytes.toBytes("row1"));
        put.add(Bytes.toBytes("cf1"), Bytes.toBytes("col1"),
                Bytes.toBytes("val2"));
boolean bool= table.checkAndPut(Bytes.toBytes("row1"), Bytes.toBytes("cf1"),
Bytes.toBytes("col1"), Bytes.toBytes("val1"), put);
System.out.println("New value put: "+bool);
```

If the value in the "`value`" arg is null, the check is for the lack (non-existence) of a column, implying that you put the new value if a value does not already exist.

Summary

In this chapter, I discussed the salient methods in the `HTable` class. In the next chapter, I will discuss using the HBase shell.

PART VI

Administration

CHAPTER 19

Using the HBase Shell

All names in the Apache HBase shell should be quoted, such as the table name, row key, and column name. Constants don't need to be quoted. Successful HBase commands return an exit code of 0. But a non-0 exit status does not necessarily indicate failure and could indicate other issues, such as loss of connectivity.

The HBase shell is started with the following command if the HBase `bin` directory is in the `PATH` environment variable:

```
hbase shell
```

HBase shell commands may also be run from a `.txt` script file. For example, if the commands are stored in the file `shell_commands.txt`, use the following command to run the commands in the script:

```
hbase shell shell_commands.txt
```

Creating a Table

A table is created using the `create` command. The syntax of the `create` command is as follows:

```
create '/path/tablename', {NAME =>'cfname'}, {NAME =>'cfname'}
```

As args to the command the first arg is the table name, followed by a dictionary of specifications per column family, followed optionally by a dictionary of a table configuration.

The following command creates a table called `t1` with column families of `f1`, `f2`, and `f3`:

```
create 't1', {NAME => 'f1'}, {NAME => 'f2'}, {NAME => 'f3'}
```

The short form of the preceding command is as follows:

```
create 't1', 'f1', 'f2', 'f3'
```

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_19

The following command creates a table called `t1` with column families of `f1`, `f2`, and `f3`. The maximum number of versions for each column family is also specified.

```
create 't1', {NAME => 'f1', VERSIONS => 1}, {NAME => 'f2', VERSIONS => 3},
{NAME => 'f3', VERSIONS => 5}
```

The following command includes whether to use block cache option `BLOCKCACHE` set to true for the `f1` column family:

```
create 't1', {NAME => 'f1', VERSIONS => 3, BLOCKCACHE => true}
```

Altering a Table

The `alter` command is used to alter the column family schema. Args provide the table name and the new column family specification. The following command alters column families `f1` and `f2`. The number of versions in `f1` is set to 2 and the number of versions in f2 is set to 3.

```
alter 't1', {NAME => 'f1', VERSIONS => 2}, {NAME => 'f2', VERSIONS => 3}
```

The following command deletes the column families `f1` and `f2` from table `t1`:

```
alter 't1', {NAME => 'f1', METHOD => 'delete'}, {NAME => 'f2', METHOD =>
'delete'}
```

Table scope attributes such as those in Table 19-1 may also be set.

Table 19-1. *Table Scope Attributes*

Attribute	Description
MAX_FILESIZE	Maximum size of the store file after which the region split occurs
DEFERRED_LOG_FLUSH	Indicates if the deferred log flush option is enabled
MEMSTORE_FLUSHSIZE	Maximum size of the MemStore after which the MemStore is flushed to disk
READONLY	Table is read-only

The following command sets the maximum file size to 256MB:

```
alter 't1', {METHOD => 'table_att', MAX_FILESIZE => '268435456'}
```

Adding Table Data

The put command is used to add data. The syntax of the put command is as follows:

```
put '/path/tablename', 'rowkey', 'cfname:colname', 'value', 'timestamp'
```

The timestamp is optional. The following command puts a value at coordinates {t1,r1,c1}, table t1 and row r1 and column c1 with timestamp ts1:

```
put 't1', 'r1', 'c1', 'value', 'ts1'
```

The column may be specified using a column family name and column qualifier. For example, the following command puts value val into column cf1:c1 in row with row key r1 in table t1 with timestamp ts1:

```
put 't1', 'r1', 'cf1:c1', 'val', 'ts1'
```

Describing a Table

The syntax of the describe command is as follows:

```
describe '/path/table'
```

To describe a table t1, run the following command:

```
describe 't1'
```

Finding If a Table Exists

To find if table t1 exists, run the exists command.

```
exists 't1'
```

Listing Tables

The list command lists all the tables.

```
list
```

Scanning a Table

The scan command is used to scan a table. The syntax of the scan command is as follows:

```
scan '/path/tablename'
```

By default, the complete table is scanned. The following command scans the complete table t1:

```
scan 't1'
```

The output of the scan command has the following format:

```
ROW           COLUMN+CELL
 row1          column=cf1:c1, timestamp=12345, value=val1
 row1          column=cf1:c2, timestamp=34567, value=val2
 row1          column=cf1:c3, timestamp=678910, value=val3
```

The following command scans table t1. The COLUMNS option specifies that only columns c2 and c5 are scanned. The LIMIT option limits only five rows to be scanned and the STARTROW option sets the start row from which the table is to be scanned. The STOPTROW sets the last row after which table is not scanned. An HBase table is sorted lexicographically and the STARTROW and STOPTROW may not even be in the table. The table is scanned lexicographically from the first row after the STARTROW if the STARTROW does not exist. Similarly, if the STOPTROW does not exist, the table is scanned up to the last row before the STOPTROW.

```
scan 't1', {COLUMNS => ['c2', 'c5'], LIMIT => 5, STARTROW => 'row1234',
STOPTROW => 'row78910'}
```

The following command scans table t1 starting with row key c and stopping with row key n:

```
scan  't1',  { STARTROW => 'c', STOPROW => 'n'}
```

The following command scans a table based on a time range:

```
scan 't1', {TIMERANGE => [123456, 124567]}
```

A filter such as the ColumnPagination filter may be set using the FILTER option. The ColumnPagination filter takes two options, limit and offset, both of type int.

```
scan 't1', {FILTER => org.apache.hadoop.hbase.filter.ColumnPaginationFilter.
new(5, 1)}
```

Block caching, which is enabled by default, may be disabled using the CACHE_BLOCKS option.

```
scan 't1', {COLUMNS => ['c2', 'c3'], CACHE_BLOCKS => false}
```

Enabling and Disabling a Table

The enable command enables a table and the disable command disables a table.

```
enable 't1'
disable 't1'
```

Dropping a Table

The drop command drops a table. Before a table is dropped, the table must be disabled.

```
disable 't1'
drop 't1'
```

Counting the Number of Rows in a Table

To count the number of rows in table t1, run the following command. By default, the row count is shown every 1,000 rows.

```
count 't1'
```

The row count interval may be set to a non-default value as follows:

```
count 't1', 10000
```

Getting Table Data

The get command is used to get table row data. The syntax of the get command is as follows:

```
get '/path/tablename', 'rowkey'
```

A row of data may be accessed or a cell data may be accessed. A row with row key r1 from table t1 is accessed as follows:

```
get 't1', 'r1'
```

The output has the following format:

```
COLUMN                   CELL
 cf1:c1                 timestamp=123, value=val1
 cf1:c2                 timestamp=234, value=val2
 cf1:c3                 timestamp=123, value=val3
```

Cell data for a single column `c1` is accessed as follows:

```
get 't1', 'r1', {COLUMN => 'c1'}
```

The `COLUMN` specification may include the column family and the column qualifier.

```
get 't1', 'r1', {COLUMN => 'cf1:c1'}
```

Or just the column family may be specified.

```
get 't1', 'r1', {COLUMN => 'cf1'}
```

Cell data for a single column like `c1` with a particular timestamp is accessed as follows:

```
get 't1', 'r1', {COLUMN => 'c1', TIMESTAMP => ts1}
```

Cell data for multiple columns `c1`, `c3`, and `c5` from column family `cf1` are accessed as follows:

```
get 't1', 'r1', {COLUMN => ['cf1:c1', 'cf1:c3', 'cf1:c5']}
```

A `get` is essentially a scan limited to one row.

Truncating a Table

The `truncate` command disables, drops, and recreates a table.

```
truncate 't1'
```

Deleting Table Data

The `delete` command may be used to delete a row, a column family, or specific cell in a row in a table. The syntax for the `delete` command is as follows:

```
delete 'tablename', 'rowkey', 'columnfamily:columnqualifier'
```

The following command deletes row `r1` from table `t1`:

```
delete 't1', 'r1'
```

The following command deletes column family `cf1` from row `r1` from table `t1`:

```
delete 't1', 'r1', 'cf1'
```

The following command deletes column cell `c1` in column family `cf1` in row `r1` in table `t1`:

```
delete 't1', 'r1', 'cf1:c1'
```

The `deleteall` command deletes all rows in a given row. For example, the following command deletes row `r1` in table `t1`:

```
deleteall 't1' 'r1'
```

Optionally, a column and a timestamp may be specified. The following command deletes column `c1` in column family `cf1` from row key `r1` in table `t1` with timestamp `ts1`:

```
deleteall 't1', 'r1', 'cf1:c1' 'ts1'
```

Summary

In this chapter, I discussed the salient HBase shell commands. In the next chapter, I will discuss bulk loading data into HBase.

CHAPTER 20

Bulk Loading Data

The `org.apache.hadoop.hbase.mapreduce.ImportTsv` utility and the `completebulkload` tool are used to bulk load data into HBase. The procedure to upload is as follows:

1. Put the data file, which is a TSV file, to be uploaded into HDFS
2. Run the `ImportTsv utility` to generate multiple HFiles from the TSV file
3. Run the `completebulkload` tool to bulk load the HFiles into HBase

Let's discuss an example. You can use the following sample data file `input.tsv` in HDFS:

```
r1      c1      c2      c3
r2      c1      c2      c3
r3      c1      c2      c3
r4      c1      c2      c3
r5      c1      c2      c3
r6      c1      c2      c3
r7      c1      c2      c3
r8      c1      c2      c3
r9      c1      c2      c3
r10     c1      c2      c3
```

Run the following `importtsv` command to generate HFiles from input file `input.tsv`:

```
HADOOP_CLASSPATH=`${HBASE_HOME}/bin/hbase classpath` ${HADOOP_HOME}/
bin/hadoop jar ${HBASE_HOME}/hbase-VERSION.jar importtsv -Dimporttsv.
columns=HBASE_ROW_KEY,cf1:c1,cf1:c2,cf1:c3 -Dimporttsv.bulk.output=hdfs://
output t1 hdfs://input.tsv
```

D. Vohra, *Apache HBase Primer*, DOI 10.1007/978-1-4842-2424-3_20

The command without the classpath settings is as follows:

```
hadoop jar hbase-VERSION.jar importtsv -Dimporttsv.columns=HBASE_ROW_
KEY,cf1:c1,cf1:c2,cf1:c3 -Dimporttsv.bulk.output=hdfs://output t1 hdfs://
input.tsv
```

The `-Dimporttsv.columns` option specifies the column names of the TSV data. Comma-separated column names are to be provided with each column name as either a column family or a *column family:column qualifier*. A column name must be included for each column data in the input TSV file. The `HBASE_ROW_KEY` column is for specifying the row key and only one must be specified. The `-Dimporttsv.bulk.output=/path/for/output` specifies HFiles are to be generated at the specified output path, which should either be a relative path on the HDFS or the absolute HDFS path, as in the example as `hdfs://output`. The target table specified is `t1`; if a target table is not specified, a table is created with the default column family descriptors. The input file on the HDFS is specified as `hdfs://input.tsv`.

The `-Dimporttsv.bulk.output` option must be specified to generate HFiles for bulk upload because without it the input data is uploaded directly into an HBase table. The other options that may be specified are in Table 20-1.

Table 20-1. *The `importtsv Command Options`*

Option	Description
`-Dimporttsv.skip.bad.lines`	Whether to skip invalid data line. Default is false.
`-Dimporttsv.separator`	Input data separator. Default is tab.
`-Dimporttsv.timestamp`	The timestamp to use. Default is `currentTimeAsLong`.
`-Dimporttsv.mapper.class`	User-defined mapper to use instead of the default, which is `org.apache.hadoop.hbase.mapreduce.TsvImporterMapper`.

When `importtsv` is run, HFiles get generated.

Next, use the `completebulkload` utility to bulk upload the HFiles into an HBase table. The `completebulkload` may be run in one of two modes: as an explicit class name or via the driver. Using the class name, the syntax is as follows:

```
bin/hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles <hdfs://
output> <tablename>
```

With the driver, the syntax is as follows:

```
hadoop jar hbase-VERSION.jar completebulkload [-c /path/to/hbase/config/
hbase-site.xml] <hdfs://output> <tablename>
```

To load the HFile generated in `hdfs://output` from `importtsv`, run the following command:

```
hbase org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles hdfs://output t1
```

Bulk load bypasses the write path completely and does not use the WAL and does not cause a split storm. A custom MR job may also be used for bulk uploading.

Summary

In this chapter, I discussed bulk loading data into HBase. This chapter concludes the book. The book is a primer on the core concepts of Apache HBase, the HBase data model, schema design and architecture, the HBase API, and administration. For usage of HBase, including installation, configuration, and creating an HBase table, please refer another book from Apress: *Practical Hadoop Ecosystem*.

Index