Lineland

Everything's a dot.

Monday, October 12, 2009

HBase Architecture 101 - Storage

One of the more hidden aspects of HBase is how data is actually stored. While the majority of users may never have to bother about it you may have to get up to speed when you want to learn what the various advanced configuration options you have at your disposal mean. "How can I tune HBase to my needs?", and other similar questions are certainly interesting once you get over the (at times steep) learning curve of setting up a basic system. Another reason wanting to know more is if for whatever reason disaster strikes and you have to recover a HBase installation.

In my own efforts getting to know the respective classes that handle the various files I started to sketch a picture in my head illustrating the storage architecture of HBase. But while the ingenious and blessed committers of HBase easily navigate back and forth through that maze I find it much more difficult to keep a coherent image. So I decided to put that sketch to paper. Here it is.

Please note that this is not a UML or call graph but a merged picture of classes and the files they handle and by no means complete though focuses on the topic of this post. I will discuss the details below and also look at the configuration options and how they affect the low-level storage files.

The Big Picture

So what does my sketch of the HBase innards really say? You can see that HBase handles basically two kinds of file types. One is used for the write-ahead log and the other for the actual data storage. The files are primarily handled by the `HRegionServer`'s. But

About Me

LARS GEORGE

I am a Software Engineer by heart and work on large scale, distributed, failover-safe systems. Especially "NoSQL" type storage architectures and their related projects like Hadoop with HBase, Hive, Pig. Or Dynomite, Voldemort, Cassandra and so on. I am offering consulting services in this area and for these products. Contact me at info@larsgeorge.com or read about my offer!

View my complete profile

Where I am as well

LinkedIn

Xing

Facebook

Twitter

My Tweets

in certain scenarios even the `HMaster` will have to perform low-level file operations. You may also notice that the actual files are in fact divided up into smaller blocks when stored within the Hadoop Distributed Filesystem (HDFS). This is also one of the areas where you can configure the system to handle larger or smaller data better. More on that later.

The general flow is that a new client contacts the Zookeeper quorum (a separate cluster of Zookeeper nodes) first to find a particular row key. It does so by retrieving the server name (i.e. host name) that hosts the -ROOT- region from Zookeeper. With that information it can query that server to get the server that hosts the .META. table. Both of these two details are cached and only looked up once. Lastly it can query the .META. server and retrieve the server that has the row the client is looking for.

Once it has been told where the row resides, i.e. in what region, it caches this information as well and contacts the `HRegionServer` hosting that region directly. So over time the client has a pretty complete picture of where to get rows from without needing to query the .META. server again.

Note: The `HMaster` is responsible to assign the regions to each `HRegionServer` when you start HBase. This also includes the "special" -ROOT- and .META. tables.

Next the `HRegionServer` opens the region it creates a corresponding `HRegion` object. When the `HRegion` is "opened" it sets up a `Store` instance for each `HColumnFamily` for every table as defined by the user beforehand. Each of the `Store` instances can in turn have one or more `StoreFile` instances, which are lightweight wrappers around the actual storage file called `HFile`. A `HRegion` also has a `MemStore` and a `HLog` instance. We will now have a look at how they work together but also where there are exceptions to the rule.

Stay Put

So how is data written to the actual storage? The client issues a `HTable.put(Put)` request to the `HRegionServer` which hands the details to the matching `HRegion` instance. The first step is now to decide if the data should be first written to the "Write-Ahead-Log" (WAL) represented by the `HLog` class. The decision is based on the flag set by the client using `Put.writeToWAL(boolean)` method. The WAL is a standard Hadoop `SequenceFile` (although it is currently discussed if that should not be changed to a more HBase suitable file format) and it stores `HLogKey`'s. These keys contain a sequential number as well as the actual data and are used to replay not yet persisted data after a server crash.

Once the data is written (or not) to the WAL it is placed in the `MemStore`. At the same time it is checked if the `MemStore` is full and in that case a flush to disk is requested. When the request is served by a separate thread in the `HRegionServer` it writes the data to an `HFile` located in the HDFS. It also saves the last written sequence number so the system knows what was persisted so far. Let"s have

a look at the files now.

Files

HBase has a configurable root directory in the HDFS but the default is `/hbase`. You can simply use the DFS tool of the Hadoop command line tool to look at the various files HBase stores.

```
$ hadoop dfs -lsr /hbase/docs
...
drwxr-xr-x   - hadoop supergroup          0 2009-09-28 14:22 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-15 14:33 /hbase
-rw-r--r--   3 hadoop supergroup      14980 2009-10-14 01:32 /hbase
-rw-r--r--   3 hadoop supergroup       1773 2009-10-14 02:33 /hbase
-rw-r--r--   3 hadoop supergroup      37902 2009-10-14 03:33 /hbase
...
-rw-r--r--   3 hadoop supergroup  137648437 2009-09-28 14:20 /hbase
...
drwxr-xr-x   - hadoop supergroup          0 2009-09-27 18:03 /hbase
-rw-r--r--   3 hadoop supergroup       2323 2009-09-01 23:16 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-13 01:36 /hbase
-rw-r--r--   3 hadoop supergroup   91540404 2009-10-13 01:36 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-09-27 18:03 /hbase
-rw-r--r--   3 hadoop supergroup  333470401 2009-09-27 18:02 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-09-04 01:16 /hbase
-rw-r--r--   3 hadoop supergroup      39499 2009-09-04 01:16 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-09-04 01:16 /hbase
-rw-r--r--   3 hadoop supergroup     134729 2009-09-04 01:16 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-08 22:45 /hbase
-rw-r--r--   3 hadoop supergroup       2867 2009-10-08 22:45 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-09 23:01 /hbase
-rw-r--r--   3 hadoop supergroup   45473255 2009-10-09 23:01 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-12 00:37 /hbase
-rw-r--r--   3 hadoop supergroup  467410053 2009-10-12 00:36 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-09 23:02 /hbase
-rw-r--r--   3 hadoop supergroup        541 2009-10-09 23:02 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-09 23:02 /hbase
-rw-r--r--   3 hadoop supergroup      84447 2009-10-09 23:02 /hbase
drwxr-xr-x   - hadoop supergroup          0 2009-10-14 10:58 /hbase
```

The first set of files are the log files handled by the `HLog` instances and which are created in a directory called `.logs` underneath the HBase root directory. Then there is another subdirectory for each `HRegionServer` and then a log for each `HRegion`.

Next there is a file called `oldlogfile.log` which you may not even see on your cluster. They are created by one of the exceptions I mentioned earlier as far as file access is concerned. They are a result of so called "log splits". When the `HMaster` starts and finds that there is a log file that is not handled by a `HRegionServer` anymore it splits the log copying the `HLogKey`'s to the new regions they should be in. It places them directly in the region's directory in a file named `oldlogfile.log`. Now when the respective `HRegion` is

Labels

hbase (19)
hadoop (16)
work (10)
linux (6)
java (4)
nosql (4)
openhug (3)
erlang (2)
music (2)
vserver (2)
apache (1)
aws (1)
bigtable (1)
couchdb (1)
ec2 (1)
eclipse (1)
fosdem (1)
home (1)
iphone (1)
katta (1)
lucene (1)
macos (1)
xen (1)
xml (1)
xsl (1)
xslt (1)

instantiated it reads these files and inserts the contained data into its local `MemStore` and starts a flush to persist the data right away and delete the file.

Note: Sometimes you may see left-over `oldlogfile.log.old` (yes, there is another .old at the end) which are caused by the `HMaster` trying repeatedly to split the log and found there was already another split log in place. At that point you have to consult with the `HRegionServer` or `HMaster` logs to see what is going on and if you can remove those files. I found at times that they were empty and therefore could safely be removed.

The next set of files are the actual regions. Each region name is encoded using a Jenkins Hash function and a directory created for it. The reason to hash the region name is because it may contain characters that cannot be used in a path name in DFS. The Jenkins Hash always returns legal characters, as simple as that. So you get the following path structure:

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

In the root of the region directory there is also a `.regioninfo` holding meta data about the region. This will be used in the future by an HBase `fsck` utility (see HBASE-7) to be able to rebuild a broken `.META.` table. For a first usage of the region info can be seen in HBASE-1867.

In each column-family directory you can see the actual data files, which I explain in the following section in detail.

Something that I have not shown above are split regions with their initial daughter reference files. When a data file within a region grows larger than the configured `hbase.hregion.max.filesize` then the region is split in two. This is done initially very quickly because the system simply creates two reference files in the new regions now supposed to host each half. The name of the reference file is an ID with the hashed name of the referenced region as a postfix, e.g. `1278437856009925445.3323223323`. The reference files only hold little information: the key the original region was split at and wether it is the top or bottom reference. Of note is that these references are then used by the `HalfHFileReader` class (which I also omitted from the big picture above as it is only used temporarily) to read the original region data files. Only upon a compaction the original files are rewritten into separate files in the new region directory. This also removes the small reference files as well as the original data file in the original region.

And this also concludes the file dump here, the last thing you see is a `compaction.dir` directory in each table directory. They are used when splitting or compacting regions as noted above. They are usually empty and are used as a scratch area to stage the new data files before swapping them into place.

HFile

So we are now at a very low level of HBase's architecture. `HFile`'s (kudos to Ryan Rawson) are the actual storage files, specifically created to serve one purpose: store HBase's data fast and efficiently. They are apparently based on Hadoop's `TFile` (see HADOOP-3315) and mimic the SSTable format used in Googles BigTable architecture. The previous use of Hadoop's `MapFile`'s in HBase proved to be not good enough performance wise. So how do the files look like?

The files have a variable length, the only fixed blocks are the FileInfo and Trailer block. As the picture shows it is the Trailer that has the pointers to the other blocks and it is written at the end of persisting the data to the file, finalizing the now immutable data store. The Index blocks record the offsets of the Data and Meta blocks. Both the Data and the Meta blocks are actually optional. But you most likely you would always find data in a data store file.

How is the block size configured? It is driven solely by the `HColumnDescriptor` which in turn is specified at table creation time by the user or defaults to reasonable standard values. Here is an example as shown in the master web based interface:

```
{NAME => 'docs', FAMILIES => [{NAME => 'cache', COMPRESSION =>
'NONE', VERSIONS => '3', TTL => '2147483647', BLOCKSIZE =>
'65536', IN_MEMORY => 'false', BLOCKCACHE => 'false'}, {NAME =>
'contents', COMPRESSION => 'NONE', VERSIONS => '3', TTL =>
'2147483647', BLOCKSIZE => '65536', IN_MEMORY => 'false',
BLOCKCACHE => 'false'}, ...
```

The default is "64KB" (or 65535 bytes). Here is what the HFile JavaDoc explains:

> "Minimum block size. We recommend a setting of minimum block size between 8KB to 1MB for general usage. Larger block size is preferred if files are primarily for sequential access. However, it would lead to inefficient random access (because there are more data to decompress). Smaller blocks are good for random access, but require more memory to hold the block index, and may be slower to create (because we must flush the compressor stream at the conclusion of each data block, which leads to an FS I/O flush). Further, due to the internal caching in Compression codec, the smallest possible block size would be around 20KB-30KB."

So each block with its prefixed "magic" header contains either plain or compressed data. How that looks like we will have a look at in the next section.

One thing you may notice is that the default block size for files in DFS is 64MB, which is 1024 times what the `HFile` default block size is. So the HBase storage files blocks do not match the Hadoop blocks. Therefore you have to think about both parameters separately and find the sweet spot in terms of performance for your particular setup.

One option in the HBase configuration you may see is `hfile.min.blocksize.size`. It seems to be only used during migration from earlier versions of HBase (since it had no block file format) and when directly creating `HFile` during bulk imports for example.

So far so good, but how can you see if a `HFile` is OK or what data it contains? There is an App for that!

The `HFile.main()` method provides the tools to dump a data file:

```
$ hbase org.apache.hadoop.hbase.io.hfile.HFile
usage: HFile  [-f ] [-v] [-r ] [-a] [-p] [-m] [-k]
 -a,--checkfamily    Enable family check
 -f,--file      File to scan. Pass full-path; e.g.
                     hdfs://a:9000/hbase/.META./12/34
 -k,--checkrow       Enable row order check; looks for out-of-order
 -m,--printmeta      Print meta data of file
 -p,--printkv        Print key/value pairs
 -r,--region    Region to scan. Pass region name; e.g. '.META.,,1'
 -v,--verbose        Verbose output; emits file and meta data delim
```

Here is an example of what the output will look like (shortened here):

```
$ hbase org.apache.hadoop.hbase.io.hfile.HFile -v -p -m -f \
  hdfs://srv1.foo.bar:9000/hbase/docs/999882558/mimetype/2642281535

Scanning -> hdfs://srv1.foo.bar:9000/hbase/docs/999882558/mimetype/
...
K: \x00\x04docA\x08mimetype\x00\x00\x01\x23y\x60\xE7\xB5\x04 V: tex
K: \x00\x04docB\x08mimetype\x00\x00\x01\x23x\x8C\x1C\x5E\x04 V: tex
K: \x00\x04docC\x08mimetype\x00\x00\x01\x23xz\xC08\x04 V: text\x2Fx
K: \x00\x04docD\x08mimetype\x00\x00\x01\x23y\x1EK\x15\x04 V: text\x
K: \x00\x04docE\x08mimetype\x00\x00\x01\x23x\xF3\x23n\x04 V: text\x
Scanned kv count -> 1554

Block index size as per heapsize: 296
reader=hdfs://srv1.foo.bar:9000/hbase/docs/999882558/mimetype/26422
  compression=none, inMemory=false, \
  firstKey=US6683275_20040127/mimetype:/1251853756871/Put, \
  lastKey=US6684814_20040203/mimetype:/1251864683374/Put, \
```

```
  avgKeyLen=37, avgValueLen=8, \
  entries=1554, length=84447
fileinfoOffset=84055, dataIndexOffset=84277, dataIndexCount=2, meta
  metaIndexCount=0, totalBytes=84055, entryCount=1554, version=1
Fileinfo:
MAJOR_COMPACTION_KEY = \xFF
MAX_SEQ_ID_KEY = 32041891
hfile.AVG_KEY_LEN = \x00\x00\x00\x25
hfile.AVG_VALUE_LEN = \x00\x00\x00\x08
hfile.COMPARATOR = org.apache.hadoop.hbase.KeyValue\x24KeyComparato
hfile.LASTKEY = \x00\x12US6684814_20040203\x08mimetype\x00\x00\x01\
```

The first part is the actual data stored as `KeyValue` pairs, explained in detail in the next section. The second part dumps the internal `HFile.Reader` properties as well as the Trailer block details and finally the FileInfo block values. This is a great way to check if a data file is still healthy.

KeyValue's

In essence each `KeyValue` in the `HFile` is simply a low-level byte array that allows for "zero-copy" access to the data, even with lazy or custom parsing if necessary. How are the instances arranged?

The structure starts with two fixed length numbers indicating the size of the key and the value part. With that info you can offset into the array to for example get direct access to the value, ignoring the key - if you know what you are doing. Otherwise you can get the required information from the key part. Once parsed into a `KeyValue` object you have getters to access the details.

Note: One thing to watch out for is the difference between `KeyValue.getKey()` and `KeyValue.getRow()`. I think for me the confusion arose from referring to "row keys" as the primary key to get a row out of HBase. That would be the latter of the two methods, i.e. `KeyValue.getRow()`. The former simply returns the complete byte array part representing the raw "key" as colored and labeled in the diagram.

This concludes my analysis of the HBase storage architecture. I hope it provides a starting point for your own efforts to dig into the grimy details. Have fun!

Update: Slightly updated with more links to JIRA issues. Also added Zookeeper to be more precise about the current mechanisms to look up a region.

Update 2: Added details about region references.

Update 3: Added more details about region lookup as requested.