Example D.5. Example Usage of LoadTestTool

```
$ hbase org.apache.hadoop.hbase.util.LoadTestTool -write 1:10:100 -num_keys 1000000
          -read 100:30 -num_tables 1 -data_block_encoding NONE -tn load_test_tool_NONE
```

D.3. Enable Data Block Encoding

Codecs are built into HBase so no extra configuration is needed. Codecs are enabled on a table by setting the `DATA_BLOCK_ENCODING` property. Disable the table before altering its DATA_BLOCK_ENCODING setting. Following is an example using HBase Shell:

Example D.6. Enable Data Block Encoding On a Table

```
hbase>  disable 'test'
hbase> alter 'test', { NAME => 'cf', DATA_BLOCK_ENCODING => 'FAST_DIFF' }
Updating all regions with the new schema...
0/1 regions updated.
1/1 regions updated.
Done.
0 row(s) in 2.2820 seconds
hbase> enable 'test'
0 row(s) in 0.1580 seconds
```

Example D.7. Verifying a ColumnFamily's Data Block Encoding

```
hbase> describe 'test'
DESCRIPTION                                          ENABLED
 'test', {NAME => 'cf', DATA_BLOCK_ENCODING => 'FAST true
 _DIFF', BLOOMFILTER => 'ROW', REPLICATION_SCOPE =>
 '0', VERSIONS => '1', COMPRESSION => 'GZ', MIN_VERS
 IONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS =
 > 'false', BLOCKSIZE => '65536', IN_MEMORY => 'fals
 e', BLOCKCACHE => 'true'}
1 row(s) in 0.0650 seconds
```

Appendix E. YCSB: The Yahoo! Cloud Serving Benchmark and HBase

TODO: Describe how YCSB is poor for putting up a decent cluster load.

TODO: Describe setup of YCSB for HBase. In particular, presplit your tables before you start a run. See HBASE-4163 Create Split Strategy for YCSB Benchmark for why and a little shell command for how to do it.

Ted Dunning redid YCSB so it's mavenized and added facility for verifying workloads. See Ted Dunning's YCSB.

Appendix F. HFile format

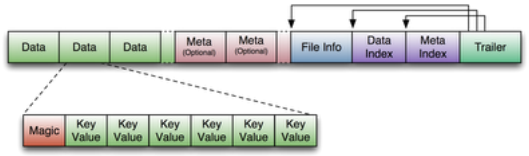
Table of Contents

This appendix describes the evolution of the HFile format.

F.1. HBase File Format (version 1)

As we will be discussing changes to the HFile format, it is useful to give a short overview of the original (HFile version 1) format.

F.1.1. Overview

An HFile in version 1 format is structured as follows: 

[33]

F.1.2. Block index format in version 1

The block index in version 1 is very straightforward. For each entry, it contains:

1. Offset (long)
2. Uncompressed size (int)
3. Key (a serialized byte array written using Bytes.writeByteArray)
    a. Key length as a variable-length integer (VInt)
    b. Key bytes

The number of entries in the block index is stored in the fixed file trailer, and has to be passed in to the method that reads the block index. One of the limitations of the block index in version 1 is that it does not provide the compressed size of a block, which turns out to be necessary for decompression. Therefore, the HFile reader has to infer this compressed size from the offset difference between blocks. We fix this limitation in version 2, where we store on-disk block size instead of uncompressed size, and get uncompressed size from the block header.

F.2. HBase file format with inline blocks (version 2)

Note: this feature was introduced in HBase 0.92

F.2.1. Motivation

We found it necessary to revise the HFile format after encountering high memory usage and slow startup times caused by large Bloom filters and block indexes in the region server. Bloom filters can get as large as 100 MB per

HFile, which adds up to 2 GB when aggregated over 20 regions. Block indexes can grow as large as 6 GB in aggregate size over the same set of regions. A region is not considered opened until all of its block index data is loaded. Large Bloom filters produce a different performance problem: the first get request that requires a Bloom filter lookup will incur the latency of loading the entire Bloom filter bit array.

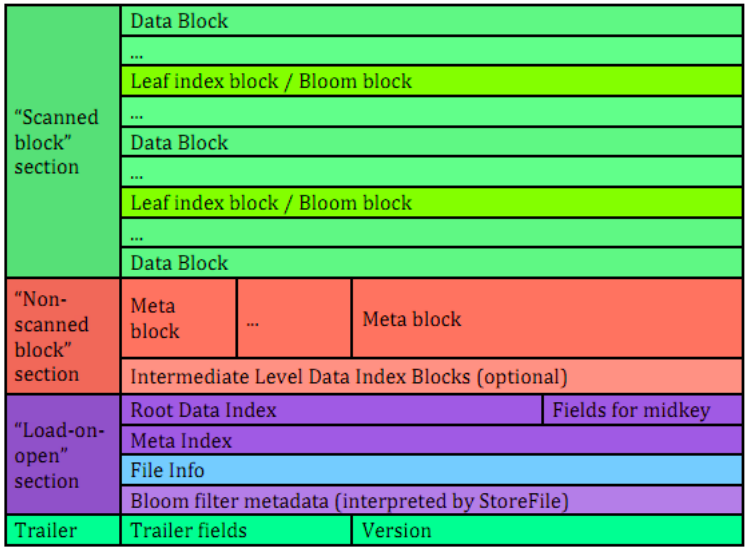
To speed up region server startup we break Bloom filters and block indexes into multiple blocks and write those blocks out as they fill up, which also reduces the HFile writer's memory footprint. In the Bloom filter case, "filling up a block" means accumulating enough keys to efficiently utilize a fixed-size bit array, and in the block index case we accumulate an "index block" of the desired size. Bloom filter blocks and index blocks (we call these "inline blocks") become interspersed with data blocks, and as a side effect we can no longer rely on the difference between block offsets to determine data block length, as it was done in version 1.

HFile is a low-level file format by design, and it should not deal with application-specific details such as Bloom filters, which are handled at StoreFile level. Therefore, we call Bloom filter blocks in an HFile "inline" blocks. We also supply HFile with an interface to write those inline blocks.

Another format modification aimed at reducing the region server startup time is to use a contiguous "load-on-open" section that has to be loaded in memory at the time an HFile is being opened. Currently, as an HFile opens, there are separate seek operations to read the trailer, data/meta indexes, and file info. To read the Bloom filter, there are two more seek operations for its "data" and "meta" portions. In version 2, we seek once to read the trailer and seek again to read everything else we need to open the file from a contiguous block.

F.2.2. Overview

The version of HBase introducing the above features reads both version 1 and 2 HFiles, but only writes version 2 HFiles. A version 2 HFile is structured as follows:



F.2.3. Unified version 2 block format

In the version 2 every block in the data section contains the following fields:

1. 8 bytes: Block type, a sequence of bytes equivalent to version 1's "magic records". Supported block types are:
    a. DATA – data blocks
    b. LEAF_INDEX – leaf-level index blocks in a multi-level-block-index
    c. BLOOM_CHUNK – Bloom filter chunks
    d. META – meta blocks (not used for Bloom filters in version 2 anymore)
    e. INTERMEDIATE_INDEX – intermediate-level index blocks in a multi-level blockindex
    f. ROOT_INDEX – root>level index blocks in a multi>level block index
    g. FILE_INFO – the "file info" block, a small key>value map of metadata
    h. BLOOM_META – a Bloom filter metadata block in the load>on>open section

i. TRAILER – a fixed>size file trailer. As opposed to the above, this is not an HFile v2 block but a fixed>size (for each HFile version) data structure

j. INDEX_V1 – this block type is only used for legacy HFile v1 block

2. Compressed size of the block's data, not including the header (int).

   Can be used for skipping the current data block when scanning HFile data.

3. Uncompressed size of the block's data, not including the header (int)

   This is equal to the compressed size if the compression algorithm is NON

4. File offset of the previous block of the same type (long)

   Can be used for seeking to the previous data/index block

5. Compressed data (or uncompressed data if the compression algorithm is NONE).

The above format of blocks is used in the following HFile sections:

1. Scanned block section. The section is named so because it contains all data blocks that need to be read when an HFile is scanned sequentially. Also contains leaf block index and Bloom chunk blocks.
2. Non-scanned block section. This section still contains unified-format v2 blocks but it does not have to be read when doing a sequential scan. This section contains “meta” blocks and intermediate-level index blocks.

We are supporting “meta” blocks in version 2 the same way they were supported in version 1, even though we do not store Bloom filter data in these blocks anymore.

F.2.4. Block index in version 2

There are three types of block indexes in HFile version 2, stored in two different formats (root and non-root):

1. Data index — version 2 multi-level block index, consisting of:
   a. Version 2 root index, stored in the data block index section of the file
   b. Optionally, version 2 intermediate levels, stored in the non%root format in the data index section of the file. Intermediate levels can only be present if leaf level blocks are present
   c. Optionally, version 2 leaf levels, stored in the non%root format inline with data blocks
2. Meta index — version 2 root index format only, stored in the meta index section of the file
3. Bloom index — version 2 root index format only, stored in the “load-on-open” section as part of Bloom filter metadata.

F.2.5. Root block index format in version 2

This format applies to:

1. Root level of the version 2 data index
2. Entire meta and Bloom indexes in version 2, which are always single-level.

A version 2 root index block is a sequence of entries of the following format, similar to entries of a version 1 block index, but storing on-disk size instead of uncompressed size.

1. Offset (long)

   This offset may point to a data block or to a deeper>level index block.

2. On-disk size (int)
3. Key (a serialized byte array stored using Bytes.writeByteArray)
   a. Key (VInt)
   b. Key bytes

A single-level version 2 block index consists of just a single root index block. To read a root index block of version 2, one needs to know the number of entries. For the data index and the meta index the number of entries is stored in the trailer, and for the Bloom index it is stored in the compound Bloom filter metadata.

For a multi-level block index we also store the following fields in the root index block in the load-on-open section of the HFile, in addition to the data structure described above:

1. Middle leaf index block offset
2. Middle leaf block on-disk size (meaning the leaf index block containing the reference to the “middle” data block of the file)
3. The index of the mid-key (defined below) in the middle leaf-level block.

These additional fields are used to efficiently retrieve the mid-key of the HFile used in HFile splits, which we define as the first key of the block with a zero-based index of (n – 1) / 2, if the total number of blocks in the HFile is n. This definition is consistent with how the mid-key was determined in HFile version 1, and is reasonable in general, because blocks are likely to be the same size on average, but we don’t have any estimates on individual key/value pair sizes.

When writing a version 2 HFile, the total number of data blocks pointed to by every leaf-level index block is kept track of. When we finish writing and the total number of leaf-level blocks is determined, it is clear which leaf-level block contains the mid-key, and the fields listed above are computed. When reading the HFile and the mid-key is requested, we retrieve the middle leaf index block (potentially from the block cache) and get the mid-key value from the appropriate position inside that leaf block.

F.2.6. Non-root block index format in version 2

This format applies to intermediate-level and leaf index blocks of a version 2 multi-level data block index. Every non-root index block is structured as follows.

1. numEntries: the number of entries (int).
2. entryOffsets: the “secondary index” of offsets of entries in the block, to facilitate a quick binary search on the key (numEntries + 1 int values). The last value is the total length of all entries in this index block. For example, in a non-root index block with entry sizes 60, 80, 50 the “secondary index” will contain the following int array: {0, 60, 140, 190}.
3. Entries. Each entry contains:
    a. Offset of the block referenced by this entry in the file (long)
    b. On>disk size of the referenced block (int)
    c. Key. The length can be calculated from entryOffsets.

F.2.7. Bloom filters in version 2

In contrast with version 1, in a version 2 HFile Bloom filter metadata is stored in the load-on-open section of the HFile for quick startup.

1. A compound Bloom filter.
    a. Bloom filter version = 3 (int). There used to be a DynamicByteBloomFilter class that had the Bloom filter version number 2
    b. The total byte size of all compound Bloom filter chunks (long)
    c. Number of hash functions (int
    d. Type of hash functions (int)
    e. The total key count inserted into the Bloom filter (long)
    f. The maximum total number of keys in the Bloom filter (long)
    g. The number of chunks (int)
    h. Comparator class used for Bloom filter keys, a UTF>8 encoded string stored using Bytes.writeByteArray
    i. Bloom block index in the version 2 root block index format

F.2.8. File Info format in versions 1 and 2

The file info block is a serialized HbaseMapWritable (essentially a map from byte arrays to byte arrays) with the following keys, among others. StoreFile-level logic adds more keys to this.

hfile.LASTKEY	The last key of the file (byte array)
hfile.AVG_KEY_LEN	The average key length in the file (int)
hfile.AVG_VALUE_LEN	The average value length in the file (int)

File info format did not change in version 2. However, we moved the file info to the final section of the file, which can be loaded as one block at the time the HFile is being opened. Also, we do not store comparator in the version 2 file info anymore. Instead, we store it in the fixed file trailer. This is because we need to know the comparator at the time of parsing the load-on-open section of the HFile.

F.2.9. Fixed file trailer format differences between versions 1 and 2

The following table shows common and different fields between fixed file trailers in versions 1 and 2. Note that the size of the trailer is different depending on the version, so it is "fixed" only within one version. However, the version is always stored as the last four-byte integer in the file.

Version 1	Version 2
File info offset (long)	
Data index offset (long)	loadOnOpenOffset (long) *The offset of the section that we need toload when opening the file.*
Number of data index entries (int)	
metaIndexOffset (long) This field is not being used by the version 1 reader, so we removed it from version 2.	uncompressedDataIndexSize (long) The total uncompressed size of the whole data block index, including root-level, intermediate-level, and leaf-level blocks.
Number of meta index entries (int)	
Total uncompressed bytes (long)	
numEntries (int)	numEntries (long)
Compression codec: 0 = LZO, 1 = GZ, 2 = NONE (int)	
	The number of levels in the data block index (int)
	firstDataBlockOffset (long) The offset of the first first data block. Used when scanning.
	lastDataBlockEnd (long) The offset of the first byte after the last key/value data block. We don't need to go beyond this offset when scanning.
Version: 1 (int)	Version: 2 (int)

F.2.10. getShortMidpointKey(an optimization for data index block)

Note: this optimization was introduced in HBase 0.95+

HFiles contain many blocks that contain a range of sorted Cells. Each cell has a key. To save IO when reading Cells, the HFile also has an index that maps a Cell's start key to the offset of the beginning of a particular block. Prior to this optimization, HBase would use the key of the first cell in each data block as the index key.

In HBASE-7845, we generate a new key that is lexicographically larger than the last key of the previous block and lexicographically equal or smaller than the start key of the current block. While actual keys can potentially be very long, this "fake key" or "virtual key" can be much shorter. For example, if the stop key of previous block is "the quick brown fox", the start key of current block is "the who", we could use "the r" as our virtual key in our hfile index.

There are two benefits to this:

- having shorter keys reduces the hfile index size, (allowing us to keep more indexes in memory), and
- using something closer to the end key of the previous block allows us to avoid a potential extra IO when the target key lives in between the "virtual key" and the key of the first element in the target block.

This optimization (implemented by the getShortMidpointKey method) is inspired by LevelDB's ByteWiseComparatorImpl::FindShortestSeparator() and FindShortSuccessor().

F.3. HBase File Format with Security Enhancements (version 3)

Note: this feature was introduced in HBase 0.98

F.3.1. Motivation

Version 3 of HFile makes changes needed to ease management of encryption at rest and cell-level metadata (which in turn is needed for cell-level ACLs and cell-level visibility labels). For more information see Section 8.7, "Transparent Server Side Encryption", Section 8.3, "Tags", Section 8.4, "Access Control", and Section 8.6, "Visibility Labels".

F.3.2. Overview

The version of HBase introducing the above features reads HFiles in versions 1, 2, and 3 but only writes version 3 HFiles. Version 3 HFiles are structured the same as version 2 HFiles. See Section F.2.2, "Overview" for more information.

F.3.3. Data Blocks in Version 3

Within an HFile, HBase cells are stored in Data Blocks as a sequence of KeyValues (see Section F.1.1, "Overview", or Lars George's excellent introduction to HBase Storage). In version 3, these KeyValue are serialized with a set of 0 or more tags:

Version 1 & 2	Version 3
Key Length (4 bytes)	
Value Length (4 bytes)	
Key bytes (variable)	
Value bytes (variable)	
	Tags Length (2 bytes)
	Tags bytes (variable)

Though the presence of tags on a cell is optional, the length of the tags is always serialized. In the case of a cell with no tags the value 0 is stored. The actual tags are stored as a sequence of tag length (2 bytes), tag type (1 byte), tag bytes (variable). The format an individual tag's bytes depends on the tag type.

F.3.4. File Info Block in Version 3

Version 3 added two additional pieces of information to the reserved keys in the file info block.

hfile.TAGS_COMPRESSED	Does the block encoder for this hfile compress tags? (boolean)
hfile.MAX_TAGS_LEN	The number of bytes in the longest tag in this hfile (int)

F.3.5. Fixed File Trailer in Version 3

The fixed file trailers written with HFile version 3 are always serialized with protocol buffers. Additionally, it adds an optional field to the version 2 protocol buffer named encryption_key. If HBase is configured to encrypt HFiles this field will store a data encryption key for this particular HFile, encrypted with the current cluster master key using AES. For more information see Section 8.7, "Transparent Server Side Encryption".

---

[33] Image courtesy of Lars George, hbase-architecture-101-storage.html.

Appendix G. Other Information About HBase

Table of Contents

G.1. HBase Videos