HFile: A Block-Indexed File Format to Store Sorted Key-Value Pairs

Sep 10, 2009 Schubert Zhang (schubert.zhang@gmail.com) http://cloudepr.blogspot.com

1. Introduction

HFile is a mimic of Google's SSTable. Now, it is available in Hadoop HBase-0.20.0. And the previous releases of HBase temporarily use an alternate file format – MapFile[4], which is a common file format in Hadoop IO package. I think HFile should also become a common file format when it becomes mature, and should be moved into the common IO package of Hadoop in the future.

Following words of SSTable are from section 4 of Google's Bigtable paper.

The Google SSTable file format is used internally to store Bigtable data. An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings. Operations are provided to look up the value associated with a specified key, and to iterate over all key/value pairs in a specified key range. Internally, each SSTable contains a sequence of blocks (typically each block is 64KB in size, but this is configurable). A block index (stored at the end of the SSTable) is used to locate blocks; the index is loaded into memory when the SSTable is opened. A lookup can be performed with a single disk seek: we first find the appropriate block by performing a binary search in the in-memory index, and then reading the appropriate block from disk. Optionally, an SSTable can be completely mapped into memory, which allows us to perform lookups and scans without touching disk.[1]

The HFile implements the same features as SSTable, but may provide more or less.

2. File Format

Data Block Size

Whenever we say Block Size, it means the uncompressed size.
The size of each data block is 64KB by default, and is configurable in HFile.Writer. It means the data block will not exceed this size more than one key/value pair. The HFile.Writer starts a new data block to add key/value pairs if the current writing block is equal to or bigger than this size. The 64KB size is same as Google's [1].

To achieve better performance, we should select different block size. If the average key/value size is very short (e.g. 100 bytes), we should select small blocks (e.g. 16KB) to avoid too many key/value pairs in each block, which will increase the latency of in-block seek, because the seeking operation always finds the key from the first key/value pair in sequence within a block.

Maximum Key Length

The key of each key/value pair is currently up to 64KB in size. Usually, 10-100 bytes is a typical size for most of our applications. Even in the data model of HBase, the key (rowkey+column family:qualifier+timestamp) should not be too long.

Maximum File Size

The trailer, file-info and total data block indexes (optionally, may add meta block indexes) will be in memory when writing and reading of an HFile. So, a larger HFile (with more data blocks) requires more memory. For example, a 1GB uncompressed HFile would have about 15600 (1GB/64KB) data blocks, and correspondingly about 15600 indexes. Suppose the average key size is 64 bytes, then we need about 1.2MB RAM (15600X80) to hold these indexes in memory.

Compression Algorithm

- Compression reduces the number of bytes written to/read from HDFS.
- Compression effectively improves the efficiency of network bandwidth and disk space
- Compression reduces the size of data needed to be read when issuing a read

To be as low friction as necessary, a real-time compression library is preferred. Currently, HFile supports following three algorithms:

(1) NONE (Default, uncompressed, string name=”none”)

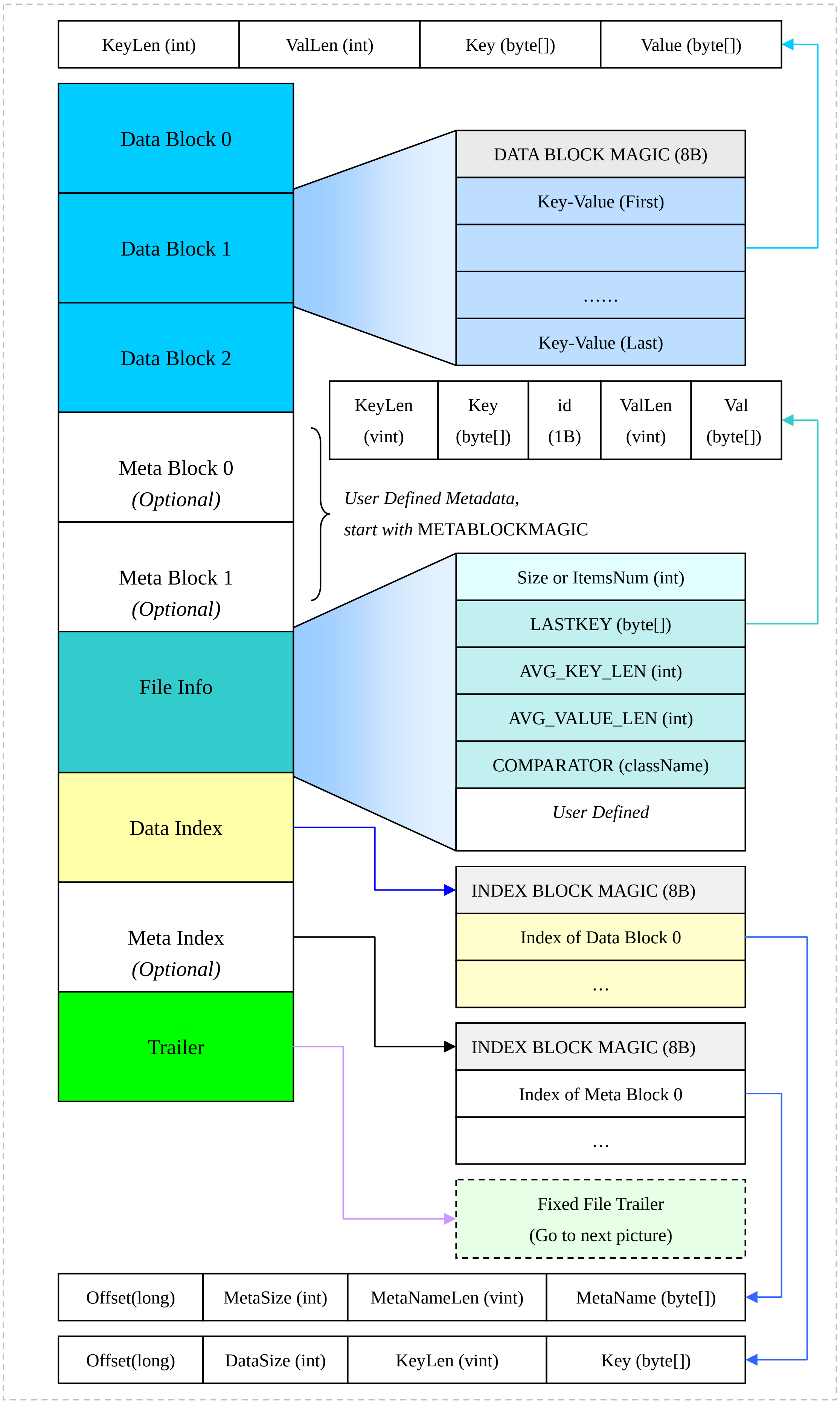
(2) GZ (Gzip, string name=”gz”)

Out of the box, HFile ships with only Gzip compression, which is fairly slow.

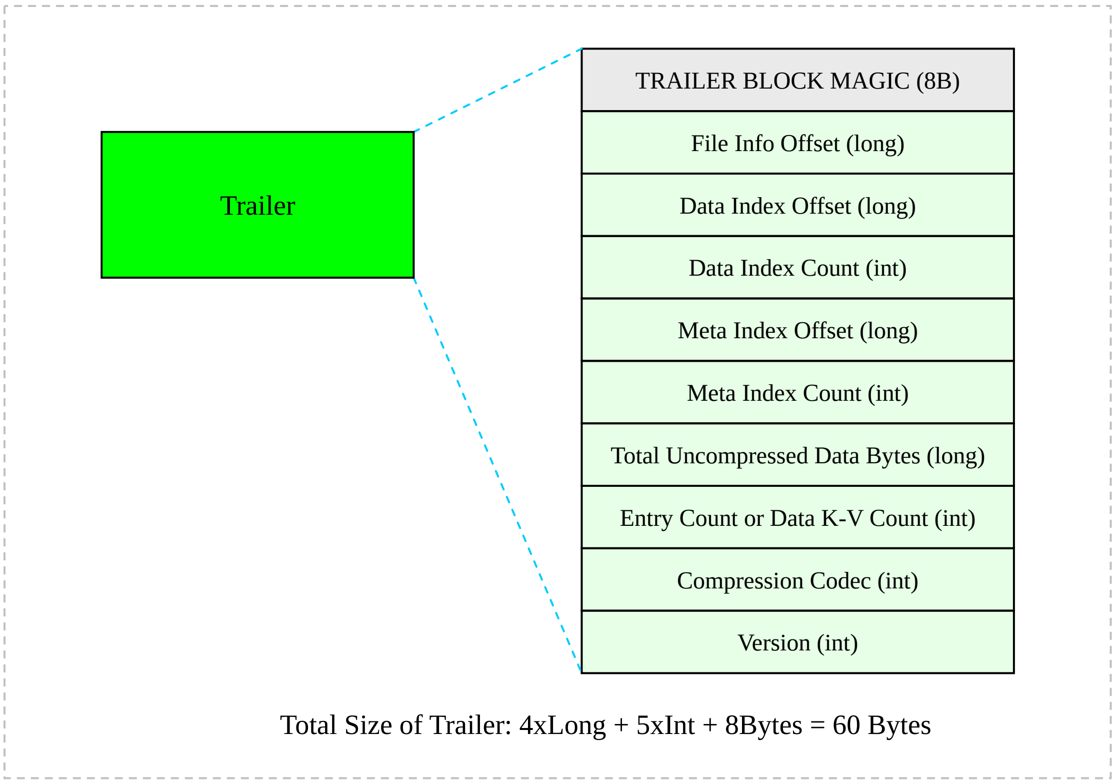
(3) LZO(Lempel-Ziv-Oberhumer, preferred, string name=”lzo”)

To achieve maximal performance and benefit, you must enable LZO, which is a lossless data compression algorithm that is focused on decompression speed.

Following figures show the format of an HFile.



KeyLen (int)
ValLen (int)
Key (byte[])
Value (byte[])
Data Block 0
Data Block 1
Data Block 2
DATA BLOCK MAGIC (8B)
Key-Value (First)
……
Key-Value (Last)
KeyLen (vint)
Key (byte[])
id (1B)
ValLen (vint)
Val (byte[])
Meta Block 0 *(Optional)*
Meta Block 1 *(Optional)*
User Defined Metadata,
start with METABLOCKMAGIC
File Info
Size or ItemsNum (int)
LASTKEY (byte[])
AVG_KEY_LEN (int)
AVG_VALUE_LEN (int)
COMPARATOR (className)
User Defined
Data Index
INDEX BLOCK MAGIC (8B)
Index of Data Block 0
…
Meta Index *(Optional)*
INDEX BLOCK MAGIC (8B)
Index of Meta Block 0
…
Trailer
Fixed File Trailer
(Go to next picture)
Offset(long)
MetaSize (int)
MetaNameLen (vint)
MetaName (byte[])
Offset(long)
DataSize (int)
KeyLen (vint)
Key (byte[])



In above figures, an HFile is separated into multiple segments, from beginning to end, they are:

- Data Block segment
  To store key/value pairs, may be compressed.
- Meta Block segment (Optional)
  To store user defined large metadata, may be compressed.
- File Info segment
  It is a small metadata of the HFile, without compression. User can add user defined small metadata (name/value) here.
- Data Block Index segment
  Indexes the data block offset in the HFile. The key of each index is the key of first key/value pair in the block.
- Meta Block Index segment (Optional)
  Indexes the meta block offset in the HFile. The key of each index is the user defined unique name of the meta block.
- Trailer
  The fix sized metadata. To hold the offset of each segment, etc. To read an HFile, we should always read the Trailer firstly.

The current implementation of HFile does not include Bloom Filter, which should be added in the future.

The FileInfo is a SortedMap in implementation. So the actual order of those fields is alphabetically based on the key.

3. LZO Compression

LZO is now removed from Hadoop or HBase 0.20+ because of GPL restrictions. To enable it, we should install native library firstly as following. [6][7][8][9]

(1) Download LZO: http://www.oberhumer.com/, and build.

```
# ./configure --build=x86_64-redhat-linux-gnu --enable-shared
--disable-asm
# make
# make install
```

Then the libraries have been installed in: /usr/local/lib

(2) Download the native connector library http://code.google.com/p/hadoop-gpl-compression/, and build.
Copy hadoo-0.20.0-core.jar to ./lib.

```
# ant compile-native
# ant jar
```

(3) Copy the native library (build/native/ Linux-amd64-64) and *hadoop-gpl-compression-0.1.0-dev.jar* to your application's lib directory. If your application is a MapReduce job, copy them to hadoop's lib directory. Your application should follow the $HADOOP_HOME/bin/hadoop script to ensure that the native hadoop library is on the library path via the system property *-Djava.library.path=<path>*. [9] For example:

```
# setup 'java.library.path' for native-hadoop code if necessary
JAVA_LIBRARY_PATH=''
if [ -d "${HADOOP_HOME}/build/native" -o -d "${HADOOP_HOME}/lib/native" ]; then
  JAVA_PLATFORM=`CLASSPATH=${CLASSPATH} ${JAVA} -Xmx32m
org.apache.hadoop.util.PlatformName | sed -e "s/ /_/g"`

  if [ -d "$HADOOP_HOME/build/native" ]; then
    JAVA_LIBRARY_PATH=${HADOOP_HOME}/build/native/${JAVA_PLATFORM}/lib
  fi

  if [ -d "${HADOOP_HOME}/lib/native" ]; then
    if [ "x$JAVA_LIBRARY_PATH" != "x" ]; then
JAVA_LIBRARY_PATH=${JAVA_LIBRARY_PATH}:${HADOOP_HOME}/lib/native/${JAVA_PLATFORM}
    else
      JAVA_LIBRARY_PATH=${HADOOP_HOME}/lib/native/${JAVA_PLATFORM}
    fi
  fi
fi
```

Then our application and hadoop/MapReduce can use LZO.

4. Performance Evaluation

Testbed

- 4 slaves + 1 master
- Machine: 4 CPU cores (2.0G), 2x500GB 7200RPM SATA disks, 8GB RAM.
- Linux: RedHat 5.1 (2.6.18-53.el5), ext3, no RAID, noatime
- 1Gbps network, all nodes under the same switch.
- Hadoop-0.20.0 (1GB heap), lzo-2.0.3

Some MapReduce-based benchmarks are designed to evaluate the performance of operations to HFiles, in parallel.

- Total key/value entries: 30,000,000.
- Key/Value size: 1000 bytes (10 for key, and 990 for value). We have totally 30GB of data.
- Sequential key ranges: 60, i.e. each range have 500,000 entries.
- Use default block size.
- The entry value is a string, each continuous 8 bytes are a filled with a same letter (A~Z). E.g. "BBBBBBBBXXXXXXXXGGGGGGGG……".

We set mapred.tasktracker.map.tasks.maximum=3 to avoid client side bottleneck.

(1) Write
Each MapTask for each range of key, which writes a separate HFile with 500,000 key/value entries.

(2) Full Scan
Each MapTask scans a separate HFile from beginning to end.

(3) Random Seek a specified key
Each MapTask opens one separate HFile, and selects a random key within that file to seek it. Each MapTask runs 50,000 (1/10 of the entries) random seeks.

(4) Random Short Scan
Each MapTask opens one separate HFile, and selects a random key within that file as a beginning to scan 30 entries. Each MapTask runs 50,000 scans, i.e. scans 50,000*30=1,500,000 entries.

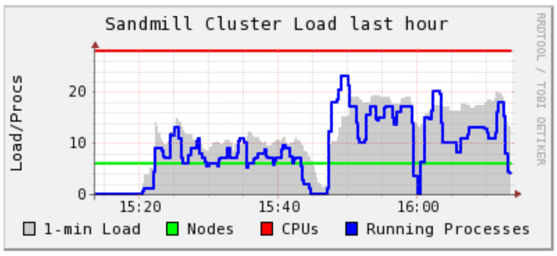
This table shows the average entries which are written/seek/scanned per second, and per node.

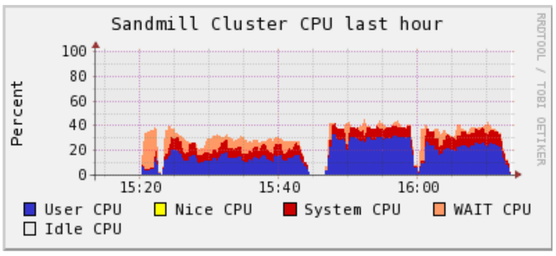
Compress / Operation	none	gz	lzo	SequenceFile (none compress)
Write	20718	23885	55147	19789
Full Scan	41436	94937	100000	28626
Random Seek	600	989	956	N/A
Random Short Scan	12241	25568	25655	N/A

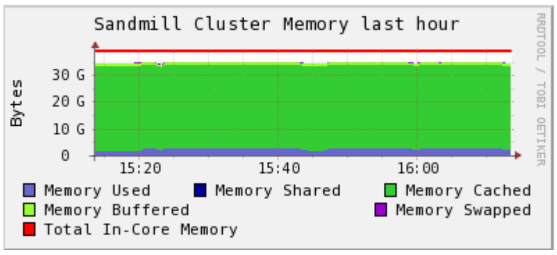
In this evaluation case, the compression ratio is about 7:1 for gz(Gzip), and about 4:1 for lzo. Even through the compression ratio is just moderate, the lzo column shows the best performance, especially for writes.

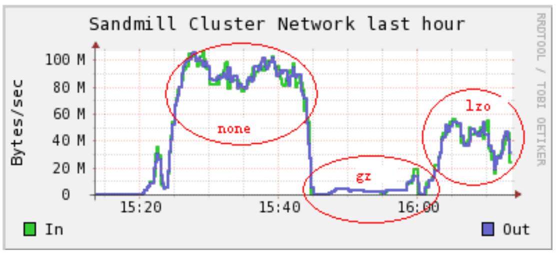
The performance of full scan is much better than SequenceFile, so HFile may provide better performance to MapReduce-based analytical applications.

The random seek in HFiles is slow, especially in none-compressed HFiles. But the above numbers already show 6X~10X better performance than a disk seek (10ms). Following Ganglia charts show us the overhead of load, CPU, and network. The random short scan makes the similar phenomena.









5. Implementation and API

5.1 HFile.Writer : How to create and write an HFile

(1) Constructors

There are 5 constructors. We suggest using following two:

```
public Writer(FileSystem fs, Path path, int blocksize,
              String compress,
              final RawComparator<byte []> comparator)
public Writer(FileSystem fs, Path path, int blocksize,
              Compression.Algorithm compress,
              final RawComparator<byte []> comparator)
```

These two constructors are same. They create file (call fs.create(…)) and get an FSDataOutputStream for writing. Since the FSDataOutputStream is created when constructing the HFile.Writer, it will be automatically closed when the HFile.Writer is closed.

The other two constructors provide FSDataOutputStream as a parameter. It means the file is created and opened outside of the HFile.Writer, so, when we close the HFile.Writer, the FSDataOutputStream will not be closed. But we do not suggest using these two constructors directly.

```
public Writer(final FSDataOutputStream ostream, final int blocksize,
              final String  compress,
              final RawComparator<byte []> c)
public Writer(final FSDataOutputStream ostream, final int blocksize,
              final Compression.Algorithm  compress,
              final RawComparator<byte []> c)
```

Another constructor only provides fs and path as parameters, all other attributes are default, i.e. NONE of compression, 64KB of block size, raw ByteArrayComparator, etc.

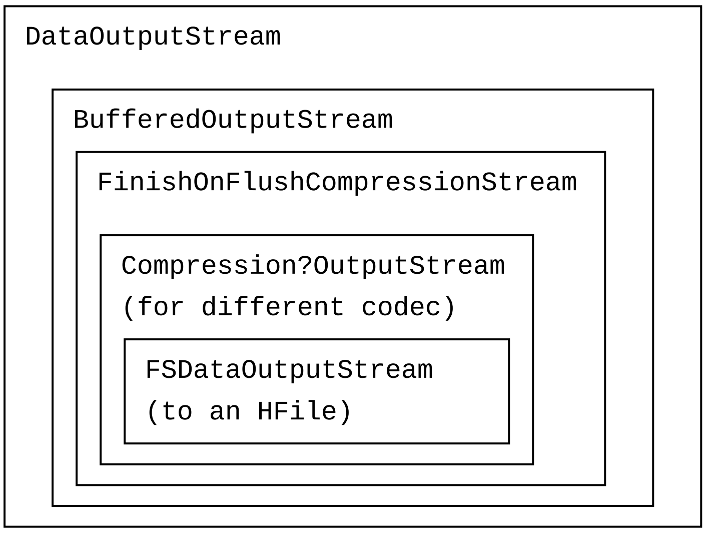
(2) Write Key/Value pairs into HFile

Before key/value pairs are written into an HFile, the application must sort them using the same comparator, i.e. all key/value pairs must be sequentially and increasingly write/append into an HFile. There are following methods to write/append key/value pairs:

```
public void append(final KeyValue kv)
public void append(final byte [] key, final byte [] value)
public void append(final byte [] key, final int koffset, final int klength,
              final byte [] value, final int voffset, final int vlength)
```

When adding a key/value pair, they will check the current block size. If the size reach the maximum size of a block, the current block will be compressed and written to the output stream (of the HFile), and then create a new block for writing. The compression is based on each block. For each block, an output stream for compression will be created from beginning of a new block and released when finish.

Following chart is the relationship of the output steams OO design:



The key/value appending operation is written from the outside (DataOutputStream), and the above OO mechanism will handle the buffer and compression functions and then write to the file in under layer file system.

Before a key/value pair is written, following will checked:
- The length of Key
- The order of Key (must bigger than the last one)

(3) Add metadata into HFile

We can add metadata block into an HFile.
public void appendMetaBlock(String metaBlockName, byte [] bytes)

The application should provide a unique metaBlockName for each metadata block within an HFile.

Reminding: If your metadata is large enough (e.g. 32KB uncompressed), you can use this feature to add a separate meta block. It may be compressed in the file.
But if your metadata is very small (e.g. less than 1KB), please use following method to append it into file info. File info will not be compressed.

public void appendFileInfo(final byte [] k, final byte [] v)

(4) Close

Before the HFile.Writer is closed, the file is not completed written. So, we must call close() to:
- finish and flush the last block

- write all meta block into file (may be compressed)
- generate and write file info metadata
- write data block indexes
- write meta block indexes
- generate and write trailer metadata
- close the output-stream.

5.2 HFile.Reader: How to read HFile

Create an HFile.Reader to open an HFile, and we can seek, scan and read on it.

(1) Constructor

We suggest using following constructor to create an HFile.Reader.

```
public Reader(FileSystem fs, Path path, BlockCache cache,
              boolean inMemory)
```

It calls fs.open(…) to open the file, and gets an FSDataInputStream for reading. The input stream will be automatically closed when the HFile.Reader is closed.

Another constructor uses InputStream as parameter directly. It means the file is opened outside the HFile.Reader.

```
public Reader(final FSDataInputStream fsdis, final long size,
              final BlockCache cache, final boolean inMemory)
```

We can use BlockCache to improve the performance of read, and the mechanism of mechanism will be described in other document.

(2) Load metadata and block indexes of an HFile

The HFile is not readable before loadFileInfo() is explicitly called . It will read metadata (Trailer, File Info) and Block Indexes (data block and meta block) into memory. And the COMPARATOR instance will reconstruct from file info.

BlockIndex

The important method of BlockIndex is:

```
int blockContainingKey(final byte[] key, int offset, int length)
```

It uses binarySearch to check if a key is contained in a block. The return value of binarySearch() is very puzzled:

Data Block Index List	Before	0	1	2	3	4	5	6	…
binarySearch() return	-1	-2	-3	-4	-5	-6	-7	-8	…

HFileScanner

We must create an HFile.Reader.Scanner to seek, scan, and read on an HFile. HFile.Reader.Scanner is an implementation of HFileScanner interface.

To seek and scan in an HFIle, we should do as following:

(1) Create a HFile.Reader, and loadFileInfo().
(2) In this HFile.Reader, calls getScanner() to obtain an HFileScanner.
(3) .1 For a scan from the beginning of the HFile, calls seekTo() to seek to the beginning of the first block.
    .2 For a scan from a key, calls seekTo(key) to seek to the position of the key or before the key (if there is not such a key in this HFile).
    .3 For a scan from before of a key, calls seekBefore(key).
(4) Calls next() to iterate over all key/value pairs. The next() will return false when it reach the end of the HFile. If an application wants to stop at any condition, it should be implemented by the application itself. (e.g. stop at a special endKey.)
(5) If you want lookup a specified key, just call seekTo(key), the returned value=0 means you found it.
(6) After we seekTo(…) or next() to a position of specified key, we can call following methods to get the current key and value.

```
public KeyValue getKeyValue() // recommended
public ByteBuffer getKey()
public ByteBuffer getValue()
```

(7) Don’t forget to close the HFile.Reader. But a scanner need not be closed, since it does not hold any resource.

References

[1] Google, Bigtable: A Distributed Storage System for Structured Data, http://labs.google.com/papers/bigtable.html

[2] HBase-0.20.0 Documentation, http://hadoop.apache.org/hbase/docs/r0.20.0/

[3] HFile code review and refinement. http://issues.apache.org/jira/browse/HBASE-1818

[4] MapFile API: http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/MapFile.html

[5] Parallel LZO: Splittable Compression for Hadoop. http://www.cloudera.com/blog/2009/06/24/parallel-lzo-splittable-compression-for-hadoop/ http://blog.chrisgoffinet.com/2009/06/parallel-lzo-splittable-on-hadoop-using-cloudera/

[6] Using LZO in Hadoop and HBase: http://wiki.apache.org/hadoop/UsingLzoCompression

[7] LZO: http://www.oberhumer.com

[8] Hadoop LZO native connector library: http://code.google.com/p/hadoop-gpl-compression/

[9] Hadoop Native Libraries Guide: http://hadoop.apache.org/common/docs/r0.20.0/native_libraries.html