**● What is the difference between memstore and hfile in HBase?**

Memstore – Memstrore is the memory where the actions and tables are stored while the insert, create statements are performed in the hbase. We can consider it as the cache memory or the first step of the storage. When this memstore gets full at time all the data is flushed out of the memstore into the disk(HDFS) called as the store file this is also called as the hfile.  This is a sequential write. It is very fast, as it avoids moving the disk drive head. Hbase uses multiple hfiles per column family, which contain the actual cells, or keyValue instances. These files are created over time as key value edits sorted in the memstore are flushed as files to disk. This flushing out operation is also written into the logs. Sometimes these files are merged together this is called as the compaction.

Every column family has one memstore. The updates are done column wise,this justifies the column wise structure of hbase.

**● Describe compactions in HBase.**

There might be many hfiles that being generated for a single memstore. In order to reduce the number of hfiles compaction process takes place.

This compaction is process of reducing the number of hfiles per memstore. This is done by copying the bunch of hfiles to form a larger hfile. Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones, performing a merge sort.

Major compaction is taking all the hfiles in the hdfs and form it into one big hfile.

Hence we can say that the compaction process of hbase is in steps first converted to reduce the number of hfiles and again use to reduced hfiles to form one single hfile. While doing the major compaction the delete makers the expired columns are removed. This process reduces the space needed for string the final hfile. Important point to note is that while doing the major compaction lots of rewriting of the data takes place in hdfs . this may hamper the ongoing real time writing of the data into hdfs. Because of this major compaction is done during holidays or when the traffic accessing the hdfs is less. There is another difference between minor and major compactions major compactions process delete markers, max versions, etc, while minor compactions don’t. This is because delete markers might also affect data in the non-merged files, so it is only possible to do this when merging all files.

Now after performing the major compaction if the newly formed hfile is greater than that of the default size set then the hfile is split into two.

**● List and explain the logical entities in HBase.**

Logical Entity in HBase are:

- Memstore

Memstore is a virtual memory reserved for each column family. This size is also configurable(size of the Memstore is defined in hbase-site.xml called hbase.hregion.memstore.flush.size) and it is used as write buffer. Hence all the contents related to a particular column family is stored in the memstore first and then stored in the hfiles. This transferring of the data from memstore to hfile in the hdfs is called as flushing of the data. For every flush a new hfile is created in hdfs. Later with minor and major compaction process the redundant data is removed and one hfile is formed. The crashing of the memstore is taken care of by the WAL entries.

- Region

In the real world scenario the tables are containing huge amount of data i.e billons and trillions of records. Hence hbase divides the records in the particular table this particular dividion is called as region. And a group of such regions is provided with the region server. So if the record in a specific region has to be accessed then the hamster contacts the region server to access the particular data.

- HLog

Hlogs are the files in which all the information of the modification is stored they are like register book. If any modification takes place then first it is put into hlog. There is one hLog per region server. All edits for all Regions carried by a particular region server are entered first in the hLog. Each hregion is identified by a unique long int. An HLog consists of multiple on-disk files, which have a chronological order. As data is flushed to other (better) on-disk structures, the log becomes obsolete. We can destroy all the log messages for a given hregion-id .

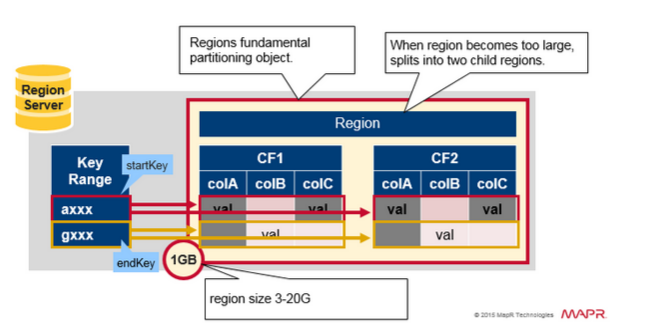
- Write-Ahead Log

WAL is a file which stores information about each transaction. Like flushing of file into hfile. The log is written into the WAL if and only if the transaction gets completed .Suppose if the memstore gets do flush and gets crashed the wal will not log this transaction and thus the memstore is informed about the same .

**● What will happen if we do not create a row key while inserting the data?**

Hbase is a Nosql database and hence has a columnar structure.

When we consider the storage of the data in hbase it is divide into regions. These regions are horizontal partitions of table. Each row in the table should be given to specify start and end key of the region.



So in order specify region row key is must.

Hbase provides us to random access to particular record and modify it. This is also done with the help of the specifying the row key of that record.

Even in case of the deleting the record markers are set rather than directly deleting the data. Then when the major compaction is done after a period of time the record is deleted.

So we can say that row key is very important aspect of the hbase. It is used to denote the lowest unit of storage that is record.

Without he row key no operations on the particular record will be possible.

● How can filters be applied in HBase and what are the benefits?

Differernt types of filters applied in hbase are

**KeyOnlyFilter**

It gives only key of keyvalue pair. No arguments needed.

**FirstKeyOnlyFilter**

This filter doesntt take any arguments. It returns only the first key-value from each row.

**PrefixFilter**

This filter takes one argument  a prefix of a row key. It returns only those key-values present in a row that starts with the specified row prefix

**ColumnPrefixFilter**

This returns all the key value pairs of the columns with specific prefix mentioned in the argumernt. It takes only one argument.

**MultipleColumnPrefixFilter**

Similar to prefix filter but it takes multiple arguments for prefixes and returns all the rows that which are matching with any of the prefix arguments.

**ColumnCountGetFilter**

This filter takes one argument  a limit. It returns the first limit number of columns in the table.

**PageFilter**

This filter takes one argument  a page size. It returns page size number of rows from the table.

**InclusiveStopFilter**

It returns all key-values present in rows up to and including the specified row. The specified row is mentioned in the argument.

**Family Filter**

It compares using the operartor. It compares the whole data with given column family for the given operator if the value is true then it will return that key value.

**Value Filter**

This is similar to family filter except here it compares the value and not the column family.

**SingleColumnValueFilter**

This filter takes a column family, a qualifier, a compare operator and a comparator. If the specified column is not found – all the columns of that row will be emitted. If the column is found and the comparison with the comparator returns true, all the columns of the row will be emitted. If the condition fails, the row will not be emitted

**● What are the data model operations in hBase?**

Different data model operations are

1. Get
2. Put
3. Scans
4. Delete

Get - The Get operation can fetch certain records from an HBase table. It is similar to select command in sql

Put – it is used for adding new values to the table or update the exsisting tables.

Scan - scan returns set of rows depending upon your search conditions. It returns all the details such as the value length,timestamp when the value was inserted ,column family in which the value was updated.

Delete – [Delete](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Delete.html) removes a row from a table. Delete command actually does not delete the value rather it sets the delete marker to the row

While doing the major compaction the value gets deleted from the table.

**● How can MapReduce be used with HBase?**

HBase provides a TableInputFormat, to which you provided a table scan, that splits the rows resulting from the table scan into the regions in which those rows reside.

The map process is passed an ImmutableBytesWritable that contains the row key for a row and a Result that contains the columns for that row.

The map process outputs its key/value pair based on its business logic in whatever form makes sense to your application.

The reduce process builds its results but emits the row key as an ImmutableBytesWritableand a Put command to store the results back to HBase.

Finally, the results are stored in HBase by the HBase MapReduce infrastructure. (You do not need to execute the Put commands.)

Map reduce can be used to process the data stored in hbase. For processing the data there is a special implementation called tableinputformatbase whose subclass is tableinputformat. The former implements the majority of the functionality but remains abstract. The subclass is a lightweight concrete version of tableinputformat and is used by many supplied samples and real mapreduce classes.

Hbase provides the tablemapper class that enforces key class 1 to be an immutablebyteswritable, and value class 1 to be a Result type—since that is what the tablerecordreader is returning.

There are specific implementations that allow output to files, or to hbase tables in the case of the tableoutputformat class. It uses a tablerecord Writer to write the data into the specific hbase output table.

**Below is the example for read and write data from tables**

public class ImportFromFile {

public static final String NAME = "ImportFromFile";

public enum Counters { LINES }

static class ImportMapper

extends Mapper<LongWritable, Text, ImmutableBytesWritable, Writable> {

private byte[] family = null;

private byte[] qualifier = null;

@Override

protected void setup(Context context)

throws IOException, InterruptedException {

String column = context.getConfiguration().get("conf.column");

byte[][] colkey = KeyValue.parseColumn(Bytes.toBytes(column));

family = colkey[0];

if (colkey.length > 1) {

qualifier = colkey[1];

}

}

@Override

public void map(LongWritable offset, Text line, Context context)

throws IOException {

try {

String lineString = line.toString();

byte[] rowkey = DigestUtils.md5(lineString);

Put put = new Put(rowkey);

put.add(family, qualifier, Bytes.toBytes(lineString));

context.write(new ImmutableBytesWritable(rowkey), put);

context.getCounter(Counters.LINES).increment(1);

} catch (Exception e) {

e.printStackTrace();

}

}

}

private static CommandLine parseArgs(String[] args) throws ParseException {

Options options = new Options();

Option o = new Option("t", "table", true,

"table to import into (must exist)");

o.setArgName("table-name");

o.setRequired(true);

options.addOption(o);

o = new Option("c", "column", true,

"column to store row data into (must exist)");

o.setArgName("family:qualifier");

o.setRequired(true);

options.addOption(o);

o = new Option("i", "input", true,

"the directory or file to read from");

o.setArgName("path-in-HDFS");

o.setRequired(true);

options.addOption(o);

options.addOption("d", "debug", false, "switch on DEBUG log level");

CommandLineParser parser = new PosixParser();

CommandLine cmd = null;

try {

cmd = parser.parse(options, args);

} catch (Exception e) {

System.err.println("ERROR: " + e.getMessage() + "\n");

HelpFormatter formatter = new HelpFormatter();

formatter.printHelp(NAME + " ", options, true);

System.exit(-1);

}

return cmd;

}

public static void main(String[] args) throws Exception {

Configuration conf = HBaseConfiguration.create();

String[] otherArgs =

new GenericOptionsParser(conf, args).getRemainingArgs();

CommandLine cmd = parseArgs(otherArgs);

String table = cmd.getOptionValue("t");

String input = cmd.getOptionValue("i");

String column = cmd.getOptionValue("c");

conf.set("conf.column", column);

Job job = new Job(conf, "Import from file " + input + " into table " + table);

job.setJarByClass(ImportFromFile.class);

job.setMapperClass(ImportMapper.class);

job.setOutputFormatClass(TableOutputFormat.class);

job.getConfiguration().set(TableOutputFormat.OUTPUT\_TABLE, table);

job.setOutputKeyClass(ImmutableBytesWritable.class);

job.setOutputValueClass(Writable.class);

job.setNumReduceTasks(0);

FileInputFormat.addInputPath(job, new Path(input));

System.exit(job.waitForCompletion(true) ? 0 : 1);

}

}

**Code Referenced from Hbase – Definitive guide**

**● What is regionserver?**

In the real world scenario the tables are containing huge amount of data i.e billons and trillions of records. Hence hbase divides the records in the particular table this particular dividion is called as region. And a group of such regions is provided with the region server. So if the record in a specific region has to be accessed then the hmaster contacts the region server to access the particular data.

A Region Server runs on an HDFS data node and has the following components:

1. WAL
2. Block Cache
3. Memstore
4. Hfiles

Below shows the block diagram of region Server

