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

When RegionServer (RS) receives write request, it directs the request to specific Region. Each Region stores set of rows. Rows data can be separated in multiple column families (CFs). Data of particular CF is stored in HStore which consists of Memstore and a set of HFiles. Memstore is kept in RS main memory, while HFiles are written to HDFS. When write request is processed, data is first written into the Memstore. Then, when certain thresholds are met (obviously, main memory is well-limited) Memstore data gets flushed into HFile.

The main reason for using Memstore is the need to store data on DFS ordered by row key. As HDFS is designed for sequential reads/writes, with no file modifications allowed, HBase cannot efficiently write data to disk as it is being received: the written data will not be sorted (when the input is not sorted) which means not optimized for future retrieval. To solve this problem HBase buffers last received data in memory (in Memstore), “sorts” it before flushing, and then writes to HDFS using fast sequential writes. Note that in reality HFile is not just a simple list of sorted rows, it is [much more than that](http://www.cloudera.com/blog/2012/06/hbase-io-hfile-input-output/).

Apart from solving the “non-ordered” problem, Memstore also has other benefits, e.g.:

* It acts as a in-memory cache which keeps recently added data. This is useful in numerous cases when last written data is accessed more frequently than older data
* There are certain optimizations that can be done to rows/cells when they are stored in memory before writing to persistent store. E.g. when it is configured to store one version of a cell for certain CF and Memstore contains multiple updates for that cell, only most recent one can be kept and older ones can be omitted (and never written to HFile).

1. **Describe compactions in HBase.**

[Apache HBase](http://hbase.apache.org/) is a distributed data store based upon a log-structured merge tree, so optimal read performance would come from having only one file per store (Column Family). However, that ideal isn’t possible during periods of heavy incoming writes. Instead, HBase will try to combine HFiles to reduce the maximum number of disk seeks needed for a read. This process is called *compaction*.

Compactions choose some files from a single store in a region and combine them. This process involves reading KeyValues in the input files and writing out any KeyValues that are not deleted, are inside of the time to live (TTL), and don’t violate the number of versions. The newly created combined file then replaces the input files in the region.

1. **How can filters be applied in HBase and what are the benefits?**

We can use custom filters to return a subset of results to the client. While this does not reduce server-side IO, it does reduce network bandwidth and reduces the amount of data the client needs to process. Filters are generally used via the Java API, but can be used from HBase Shell for testing and debugging purposes

1. **What are the data model operations in hBase?**

The four primary data model operations are Get, Put, Scan, and Delete.

**Get:**

[Get](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Get.html) returns attributes for a specified row. Gets are executed via [HTable.get](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html" \l "get%28org.apache.hadoop.hbase.client.Get%29" \t "_top).

**Put:**

[Put](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Put.html) either adds new rows to a table (if the key is new) or can update existing rows (if the key already exists). Puts are executed via [HTable.put](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html" \l "put%28org.apache.hadoop.hbase.client.Put%29" \t "_top) (writeBuffer) or[HTable.batch](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html#batch%28java.util.List%29) (non-writeBuffer).

**Scans:**

[Scan](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Scan.html) allow iteration over multiple rows for specified attributes.

**Delete:**

[Delete](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Delete.html) removes a row from a table. Deletes are executed via [HTable.delete](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html" \l "delete%28org.apache.hadoop.hbase.client.Delete%29" \t "_top).

1. **How can MapReduce be used with HBase?**

The following example uses HBase as a MapReduce source and sink with a summarization step. This example will count the number of distinct instances of a value in a table and write those summarized counts in another table.

1. Configuration config = HBaseConfiguration.create();
2. Job job = new Job(config,"ExampleSummary");
3. job.setJarByClass(MySummaryJob.class); // class that contains mapper and reducer
4. Scan scan = new Scan();
5. scan.setCaching(500); // 1 is the default in Scan, which will be bad for MapReduce jobs
6. scan.setCacheBlocks(false); // don't set to true for MR jobs
7. // set other scan attrs
8. TableMapReduceUtil.initTableMapperJob(
9. sourceTable, // input table
10. scan, // Scan instance to control CF and attribute selection
11. MyMapper.class, // mapper class
12. Text.class, // mapper output key
13. IntWritable.class, // mapper output value
14. job);
15. TableMapReduceUtil.initTableReducerJob(
16. targetTable, // output table
17. MyTableReducer.class, // reducer class
18. job);
19. job.setNumReduceTasks(1); // at least one, adjust as required
20. boolean b = job.waitForCompletion(true);
21. if (!b) {
22. throw new IOException("error with job!");
23. }

In this example mapper a column with a String-value is chosen as the value to summarize upon. This value is used as the key to emit from the mapper, and an IntWritablerepresents an instance counter.

1. public static class MyMapper extends TableMapper<Text, IntWritable> {
2. private final IntWritable ONE = new IntWritable(1);
3. private Text text = new Text();
4. public void map(ImmutableBytesWritable row, Result value, Context context) throws IOException, InterruptedException {
5. String val = new String(value.getValue(Bytes.toBytes("cf"), Bytes.toBytes("attr1")));
6. text.set(val); // we can only emit Writables...
7. context.write(text, ONE);
8. }
9. }

In the reducer, the "ones" are counted (just like any other MR example that does this), and then emits a Put.

1. public static class MyTableReducer extends TableReducer<Text, IntWritable, ImmutableBytesWritable> {
2. public void reduce(Text key, Iterable<IntWritable> values, Context context) throws IOException, InterruptedException {
3. int i = 0;
4. for (IntWritable val : values) {
5. i += val.get();
6. }
7. Put put = new Put(Bytes.toBytes(key.toString()));
8. put.add(Bytes.toBytes("cf"), Bytes.toBytes("count"), Bytes.toBytes(i));
9. context.write(null, put);
10. }
11. }

**6. What is regionserver?**

Data nodes store data. Region server(s) essentially buffer I/O operations; data is permanently stored on HDFS (that is, data nodes). I do not think that putting region server on your 'master' node is a good idea.

Here is a simplified picture of how regions are managed:

You have a cluster running HDFS (NameNode + DataNodes) with replication factor of 3 (each HDFS block is copied into 3 different DataNodes).

You run RegionServers on the same servers as DataNodes. When write request comes to RegionServer it first writes changes into memory and commit log; then at some point it decides that it is time to write changes to permanent storage on HDFS. Here is where data locality comes into play: since you run RegionServer and DataNode on the same server, first HDFS block replica of the file will be written to the same server. Two other replicas will be written to, well, other DataNodes. As a result RegionServer serving the region will almost always have access to local copy of data.

What if RegionServer crashes or RegionMaster decided to reassign region to another RegionServer (to keep cluster balanced)? New RegionServer will be forced to perform remote read first, but as soon as compaction is performed (merging of change log into the data) - new file will be written to HDFS by the new RegionServer, and local copy will be created on the RegionServer (again, because DataNode and RegionServer runs on the same server).

Note: in case of RegionServer crash, regions previously assigned to it will be reassigned to multiple RegionServers.

Good reads:

* Tom White, "Hadoop, The Definitive Guide" has good explanation of HDFS architecture. Unfortunately I did not read original Google GFS paper, so I cannot tell if it is easy to follow.
* [Google BigTable](http://research.google.com/archive/bigtable-osdi06.pdf) article. HBase is implementation of Google BigTable, and I found that architecture description in this article is the easiest to follow.

Here are nomenclature differences between Google Bigtable and HBase implementation (from Lars George, "HBase, The Definitive Guide"):

* HBase - Bigtable
* Region - Tablet
* RegionServer - Tablet server
* Flush - Minor compaction
* Minor compaction - Merging compaction
* Major compaction - Major compaction
* Write ahead log - Commit log
* HDFS - GFS
* Hadoop MapReduce - MapReduce
* MemStore - memtable
* HFile - SSTable
* Zookeeper - Chubby

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

Rows in HBase are sorted lexicographically by row key. This design optimizes for scans, allowing you to store related rows, or rows that will be read together, near each other. However, poorly designed row keys are a common source of *hotspotting*. Hotspotting occurs when a large amount of client traffic is directed at one node, or only a few nodes, of a cluster. This traffic may represent reads, writes, or other operations. The traffic overwhelms the single machine responsible for hosting that region, causing performance degradation and potentially leading to region unavailability. This can also have adverse effects on other regions hosted by the same region server as that host is unable to service the requested load. It is important to design data access patterns such that the cluster is fully and evenly utilized.

To prevent hotspotting on writes, design your row keys such that rows that truly do need to be in the same region are, but in the bigger picture, data is being written to multiple regions across the cluster, rather than one at a time. Some common techniques for avoiding hotspotting are described below, along with some of their advantages and drawbacks