**Answers**

**1. What is the difference between Memstore and Hfile in HBase?**

**Ans:**

The Memstore stores updates in memory as sorted KeyValues, the same as it would be stored in an Hfile. There is one Memstore per column family. The updates are sorted per column family. When the Memstore accumulates enough data, the entire sorted set is written to a new HFile in HDFS. HBase uses multiple Hfiles per column family, which contain the actual cells, or KeyValue instances.

Data is stored in an HFile which contains sorted key/values. When the MemStore accumulates enough data, the entire sorted KeyValue set is written to a new HFile in HDFS. This is a sequential write. It is very fast, as it avoids moving the disk drive head.

**2. Describe compactions in HBase.**

**Ans:**

*Compaction*, the process by which HBase cleans up after itself, comes in two flavors: major and minor. Major compactions can be a big deal, but first you need to understand minor compactions. Minor compactions combine a configurable number of smaller HFiles into one larger HFile. You can tune the number of HFiles to compact and the frequency of a minor compaction. Minor compactions are important because without them, reading a particular row can require many disk reads and cause slow overall performance.

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

**Ans:**

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

**Ans:** We cannot insert the data without providing the row key.

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

**Ans:** When reading data from HBase using Get or Scan operations, you 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.

HBase filters take zero or more arguments, in parentheses. Where the argument is a string, it is surrounded by single quotes ('string').

**KeyOnlyFilter** - takes no arguments. Returns the key portion of each key-value pair.

Syntax: KeyOnlyFilter ()

**FirstKeyOnlyFilter** - takes no arguments. Returns the key portion of the first key-value pair.

Syntax: FirstKeyOnlyFilter ()

**PrefixFilter** - takes a single argument, a prefix of a row key. It returns only those key-values present in a row that start with the specified row prefix

Syntax: PrefixFilter (‘<row\_prefix>’)

Example: PrefixFilter (‘Row’

**ColumnPrefixFilter** - takes a single argument, a column prefix. It returns only those key-values present in a column that starts with the specified column prefix.

Syntax: ColumnPrefixFilter (‘<column\_prefix>’)

Example: ColumnPrefixFilter (‘Col’)

**MultipleColumnPrefixFilter** - takes a list of column prefixes. It returns key-values that are present in a column that starts with any of the specified column prefixes.

Syntax: MultipleColumnPrefixFilter (‘<column\_prefix>’, ‘<column\_prefix>’, …, ‘<column\_prefix>’)

Example: MultipleColumnPrefixFilter (‘Col1’, ‘Col2’)

**ColumnCountGetFilter** - takes one argument, a limit. It returns the first limit number of columns in the table.

Syntax: ColumnCountGetFilter (‘<limit>’)

Example: ColumnCountGetFilter (4)

**6. What are the data model operations in HBase?**

**Ans:** The four primary data model operations are Get, Put, Scan, and Delete. Operations are applied via [HTable](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html) instances.

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

[**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#put%28org.apache.hadoop.hbase.client.Put%29) (writeBuffer) or [HTable.batch](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html#batch%28java.util.List%29) (non-writeBuffer).

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

[**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#delete%28org.apache.hadoop.hbase.client.Delete%29). HBase does not modify data in place, and so deletes are handled by creating new markers called tombstones. These tombstones, along with the dead values, are cleaned up on major compactions.

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

**Ans:**

1. 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.
2. 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.
3. The map process outputs its key/value pair based on its business logic in whatever form makes sense to your application.
4. The reduce process builds its results but emits the row key as an ImmutableBytesWritable and a Put command to store the results back to HBase.
5. Finally, the results are stored in HBase by the HBase MapReduce infrastructure. (You do not need to execute the Put commands.)

**8. What is regionserver?**

**Ans:** RegionServers are the software processes (often called daemons) you activate to store and retrieve data in HBase (Hadoop Database). In production environments, each RegionServer is deployed on its own dedicated compute node. When you start using HBase, you create a table and then begin storing and retrieving your data. However, at some point and perhaps quite quickly the table grows beyond a configurable limit. At this point, the HBase system automatically splits the table and distributes the load to another RegionServer. In this process, often referred to as *auto-sharding*, HBase automatically scales as you add data to the system — a huge benefit compared to most database management systems, which require manual intervention to scale the overall system beyond a single server. With HBase, as long as you have in the rack another spare server that’s configured, scaling is automatic!