**What is the difference between memstore and hfile in HBase:**

# HBase MemStore:

The MemStore is a write buffer where HBase accumulates data in memory before a permanent write.

Its contents are flushed to disk to form an HFile when the MemStore fills up.

It doesn't write to an existing HFile but instead forms a new file on every flush.

The HFile is the underlying storage format for HBase.

HFiles belong to a column family(one MemStore per column family). A column family can have multiple HFiles, but the reverse isn't true.

size of the MemStore is defined in hbase-site.xml called hbase.hregion.memstore.flush.size

The MemStore holds in-memory modifications to the Store. Modifications are [KeyValue](https://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/KeyValue.html)s. When asked to flush, current memstore is moved to snapshot and is cleared. We continue to serve edits out of new memstore and backing snapshot until flusher reports in that the flush succeeded. At this point we let the snapshot go.

The MemStore functions should not be called in parallel. Callers should hold write and read locks. This is done in [Store](https://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/regionserver/Store.html).

**hfile in HBase:**

HBase uses HDFS for saving the data on the region servers. HBase writes the data using HFile onto the HDFS. Data stored in HFile are in the format of key/value pairs.

Row key is primary identifier.

HFiles store the rows as sorted KeyValues on disk.

HFile is Unit of Storage used by Hbase

HFile is data file HBase which is stored on HDFS

**Describe compactions in HBase.**

[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.

Now, whenever a client asks for data, HBase knows the data from the input files are held in one contiguous file on disk — hence only one seek is needed, whereas previously one for each file could be required.

But disk IO isn’t free, and without careful attention, rewriting data over and over can lead to some serious network and disk over-subscription.

In other words, compaction is about trading some disk IO now for fewer seeks later.

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

Logical entities are the simplest of entities because they have no visual component and only exist to service [input](https://developer.valvesoftware.com/wiki/Input) from other entities.

[math\_counter](https://developer.valvesoftware.com/wiki/Math_counter) for example stores a value that can be added to or subtracted from; other entities in the map can modify the data with inputs or receive information from it with an output.

**Normalization**

In a relational database, you normalize the schema to eliminate redundancy by putting repeating information into a table of its own. This has the following benefits:

You don’t have to update multiple copies when an update happens, which makes writes faster.

You reduce the storage size by having a single copy instead of multiple copies.

**De-normalization**

In a de-normalized datastore, you store in one table what would be multiple indexes in a relational world. De-normalization can be thought of as a replacement for joins.

Often with HBase, you de-normalize or duplicate data so that data is accessed and stored together.

**Generic Data, Event Data, and Entity-Attribute-Value**

Generic data that is schemaless is often expressed as name value or entity attribute value. In a relational database, this is complicated to represent.

A conventional relational table consists of attribute columns that are relevant for every row in the table, because every row represents an instance of a **similar object**.

A different set of attributes represents a different type of object, and thus belongs in a different table.

The advantage of HBase is that you can define columns on the fly, put attribute names in column qualifiers, and group data by column families.

**Self-Join Relationship – HBase**

A self-join is a relationship in which both match fields are defined in the same table.

Consider a schema for twitter relationships, where the queries are: which users does userX follow, and which users follow userX? Here’s a possible solution:

The userids are put in a composite row key with the relationship type as a separator. For example, Carol follows Steve Jobs and Carol is followed by BillyBob.

This allows for row key scans for everyone carol:follows or carol:followedby

**Tree, Graph Data**

Each row shows a node, and the row key is equal to the node id. There is a column family for parent p, and a column family children c.

The column qualifiers are equal to the parent or child node ids, and the value is equal to the type to node.

This allows to quickly find the parent or children nodes from the row key.

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

Every interaction you are going to do in database will start with the RowKey only, so a row key can not be empty.so , data will not be inserted in the table.

**How can filters be applied in HBase and what are the benefits:**

# HBase Filtering:

# Benefit

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 using the Java API, but can be used from HBase Shell for testing and debugging purposes.

## Filter Types

HBase includes several filter types, as well as the ability to group filters together and create your own custom filters.

**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)

**PageFilter** - takes one argument, a page size. It returns page size number of rows from the table.

Syntax: PageFilter (‘<page\_size>’)

Example: PageFilter (2)

**ColumnPaginationFilter** - takes two arguments, a limit and offset. It returnslimit number of columns after offset number of columns. It does this for all the rows.

Syntax: ColumnPaginationFilter (‘<limit>’, ‘<offset>’)

Example: ColumnPaginationFilter (3, 5)

**InclusiveStopFilter** - takes one argument, a row key on which to stop scanning. It returns all key-values present in rows *up to and including* the specified row.

Syntax: InclusiveStopFilter (‘<stop\_row\_key>’)

Example: InclusiveStopFilter (‘Row2’)

**TimeStampsFilter** - takes a list of timestamps. It returns those key-values whose timestamps matches *any* of the specified timestamps.

Syntax: TimeStampsFilter (<timestamp>, <timestamp>, ... ,<timestamp>)

Example: TimeStampsFilter (5985489, 48895495, 58489845945)

Syntax: ValueFilter (<compareOp>, ‘<value\_comparator>’)

Example: ValueFilter (!=, ‘binary:Value’)

**What are the data model operations in hBase:**

Put Method – To store data in HBase

Get Method – To retrieve data stored in HBase.

Delete Method- To delete the data from HBase tables.

Scan Method –To iterate over the data with larger key ranges or the entire table.

**How can MapReduce be used with HBase:**

**Map Reduce in 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 ImmutableBytesWritable and 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.)

**What is regionserver**:

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 in big data use cases — 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.