

**Solution 1.** 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.

HFile-

* 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.
* HFiles are immutable.
* **Data from a single column family for a single row need not be stored in the same HFile.**

**Difference-**

When something is written to HBase, it is first written to an in-memory store (memstore), once this memstore reaches a certain size, it is flushed to disk into a store file (everything is also written immediately to a log file for durability). The store files created on disk are immutable. Sometimes the store files are merged together, this is done by a process called compaction.

**Solution 2. Compaction**

[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.

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.

**Solution 3. Logical entities of HBase**

1. **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.

1. **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.

1. **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.

1. **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

1. **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.

**Solution 4. Row Key-**

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

**Solution 5. HBase Filtering**

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

**Types of filters-**

### FirstKeyOnlyFilter

This filter doesn’t take any arguments. It returns solely the primary key-value from every row.

### Syntax

FirstKeyOnlyFilter ()

### KeyOnlyFilter

This filter doesn’t take any arguments. It returns solely the key part of every key-value.

### Syntax

KeyOnlyFilter ()

### prefixfilter:

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

### Syntax

PrefixFilter (<row\_prefix>)

### ColumnPrefixFilter

This filter takes one argument as column prefix. It returns solely those key-values present in the very column that starts with the specified column prefix. The column prefix should be the form qualifier

### Syntax

ColumnPrefixFilter(<column\_prefix>)

### MultipleColumnPrefixFilter

This filter takes a listing of column prefixes. It returns key-values that are present in the very column that starts with any of the specified column prefixes. every column prefixes should be a form qualifier.

Syntax

MultipleColumnPrefixFilter(â€˜<column\_prefix>,<column\_prefix>,….<column\_prefix>)

### ColumnCountGetFilter

This filter takes one argument a limit. It returns the primary limit number of columns within the table.

### Syntax

ColumnCountGetFilter(<limit>)

### PageFilter

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

### Syntax

PageFilter (<page\_size>)l

### InclusiveStopFilter

This filter takes one argument as row key on that to prevent scanning. It returns all key-values present in rows together with the specified row.

### Syntax

InclusiveStopFilter(<stop\_row\_key>)

### Qualifier Filter (Family Filter)

This filter takes a compare operator and a comparator. It compares every qualifier name with the comparator using the compare operator and if the comparison returns true, it returns all the key-values in this column.

### Syntax

QualifierFilter (<compareOp>, <qualifier\_comparator>)

### ValueFilter

### This filter takes a compare operator and a comparator. It compares every value with the comparator using the compare operator and if the comparison returns true, it returns that key-value.

### Syntax

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

**Solution 6. Data Model operations-**

1. **Get-** get(Get get)

* Extracts certain cells from a given row.
* return type: Result

1. **getScanner**- getScanner(Scan scan)

* Returns a scanner on the current table as specified by the Scan object.
* return type: ResultScanner

3. **Put**- put(Put put)

* Puts some data in the table.
* Put 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" \l "batch%28java.util.List%29" \t "_top) (non-writeBuffer).

1. **Scan**- [Scan](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Scan.html) allow iteration over multiple rows for specified attributes.
2. **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).

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.

**Solution 7. Map Reduce with HBase-**

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 ImmutableBytesWritableand 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.)

you can write MapReduce applications using HBase as a data source (the source of the data you’re analyzing), a sink (the destination to where your output will be written), or both.

**Solution 8. Region Server-**

* HBase Tables are divided horizontally by row key range into “Regions.”
* A region contains all rows in the table between the region’s start key and end key.
* Regions are assigned to the nodes in the cluster, called “Region Servers,” and these serve data for reads and writes.
* A region server can serve about 1,000 regions.
* Regions are the basic element of availability and distribution for tables, and are comprised of a Store per Column Family.

The region servers have regions that -

* Communicate with the client and handle data-related operations.
* Handle read and write requests for all the regions under it.
* Decide the size of the region by following the region size thresholds.

When we take a deeper look into the region server, it contain regions and stores as shown below:

The store contains memory store and HFiles. Memstore is just like a cache memory. Anything that is entered into the HBase is stored here initially. Later, the data is transferred and saved in Hfiles as blocks and the memstore is flushed.