**Assignment 32.1**

**Problem Statement:**

**Explain in brief**

**● What is the difference between memstore and hfile in 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.  
There is one MemStore per column family.

The size of the MemStore is defined by the system-wide property in hbase-site.xml called hbase.hregion.memstore.flush.size

The data in the **MemStore** is ordered. If the **MemStore** becomes full, it is persisted to an HFile on disk.

Rows are written to the MemStore.

**HFile:**

The HFile is the underlying storage format for HBase.  
HFiles belong to a column family and a column family can have multiple HFiles.  
But a single HFile can’t have data for multiple column families.

HFiles are the physical representation of data in HBase. Clients do not read HFiles directly but go through region servers to get to the data.

**● Describe compactions in HBase.**

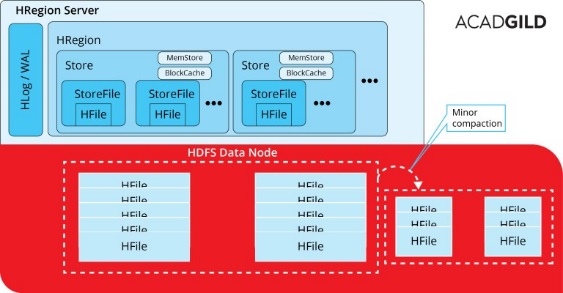
HBase is the database for [Hadoop ecosystem](https://acadgild.com/big-data/big-data-development), where distributed file system is used at the bottom layer, where the data is actually stored in physical form. Within HBase, Cache and RAM are used as a storage area, which gives speed to the ecosystem.

The servers are active all day long and while acting on [Big Data](https://acadgild.com/big-data/big-data-hadoop-administration), HBase hardly gets to write data. Therefore, it breaks the writing process into two parts: Minor Compaction and Major Compaction.

When the storage area of HBase is all most filled with data, it starts creating compressed files, which occupies less memory.

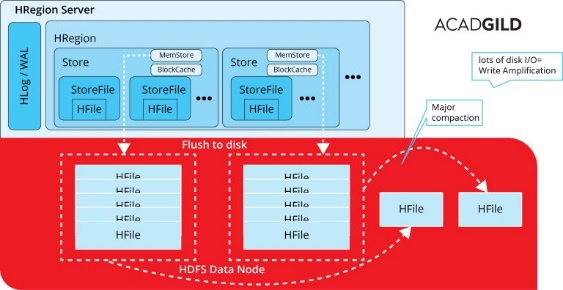
Here are the various processes involved in Minor Compaction:

* Bigger Hfile are created by combining smaller Hfiles.
* Hfile keeps the deleted file with them.
* Increases space in memory, useful to store more data.
* Merge sorting is used in process.



The other way to go around is major compaction

* Data present per column family in one region is accumulated to 1 Hfile.
* During this process, all deleted files or expired cells are deleted permanently
* Increase read performance of newly created Hfile.
* Accepts lots of I/O.
* Possibilities for traffic congestion.
* The Major compaction process is also known as Write Amplification Process.
* This process must be scheduled at a minimum bandwidth of network I/O.



**HBase compaction tuning tips:**

1. Disabling automatic major compactions
2. Maximum compaction selection size
3. Offpeakcompaction

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

The Data Model in HBase is designed to accommodate semi-structured data that could vary in field size, data type and columns.

Additionally, the layout of the data model makes it easier to partition the data and distribute it across the cluster.

The Data Model in HBase is made of different logical components such as Tables, Rows, Column Families, Columns, Cells and Versions.

**Tables**– The HBase Tables are more like logical collection of rows stored in separate partitions called Regions. As shown above, every Region is then served by exactly one Region Server. The figure above shows a representation of a Table.

**Rows**– A row is one instance of data in a table and is identified by a *rowkey*. Rowkeys are unique in a Table and are always treated as a byte[].

**Column Families** – Data in a row are grouped together as Column Families. Each Column Family has one more Columns and these Columns in a family are stored together in a low level storage file known as HFile. Column Families form the basic unit of physical storage to which certain HBase features like compression are applied. Hence it’s important that proper care be taken when designing Column Families in table. The table above shows Customer and Sales Column Families. The Customer Column Family is made up 2 columns – Name and City, whereas the Sales Column Families is made up to 2 columns – Product and Amount.

**Columns** – A Column Family is made of one or more columns. A Column is identified by a Column Qualifier that consists of the Column Family name concatenated with the Column name using a colon – example: columnfamily : columnname. There can be multiple Columns within a Column Family and Rows within a table can have varied number of Columns.

**Cell**– A Cell stores data and is essentially a unique combination of rowkey, Column Family and the Column (Column Qualifier). The data stored in a Cell is called its value and the data type is always treated as byte[].

**Version**– The data stored in a cell is versioned and versions of data are identified by the timestamp. The number of versions of data retained in a column family is configurable and this value by default is 3.

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

Every row in an HBase table has a unique identifier called its rowkey (Which is equivalent to Primary key in RDBMS, which would be distinct throughout the table).

Every interaction you are going to do in database will start with the RowKey only.

Row key is defined by the application. As the combined key is pre-fixed by the rowkey, it enables the application to define the desired sort order.

It also allows logical grouping of cells and make sure that all cells with the same rowkey are co-located on the same server.

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

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.

**Dynamically loading the custom filter:**

CDH 5.5 and higher adds (and enables by default) the ability to dynamically load a custom filter by adding a JAR with your filter to the directory specified by the hbase.dynamic.jars.dir property (which defaults to the lib/ directory under the HBase root directory).

To disable automatic loading of dynamic JARs, set hbase.use.dynamic.jars to false in the advanced configuration snippet for hbase-site.xml if you use Cloudera Manager, or to hbase-site.xml otherwise.

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

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

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

**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

**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.

**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.

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

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

The four primary data model operations are:

1) Get

2) Put

3) Scan

4) 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" \l "batch%28java.util.List%29" \t "_top) (non-writeBuffer).

**Scan-**

[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).

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.

**● How can MapReduce be used 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 Immutable Bytes Writable 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 Immutable Bytes Writable and a Put command to store the results back to HBase.
5. Finally, the results are stored in HBase by the HBase MapReduce infrastructure.

We 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

**● What is regionserver?**

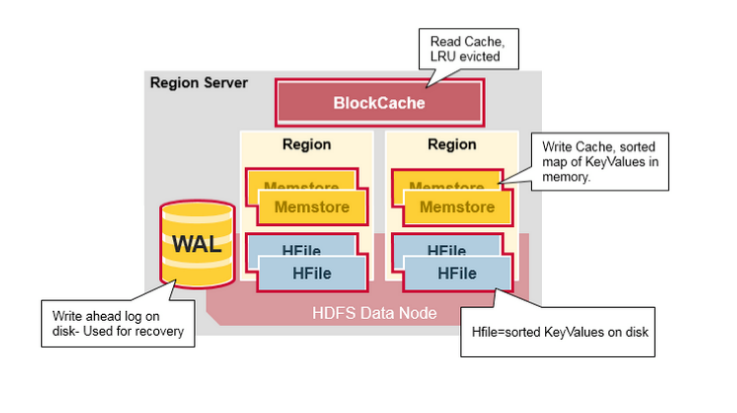
RegionServers are the software processes we activate to store and retrieve data in HBase . In production environments, each RegionServer is deployed on its own dedicated compute node. When we start using HBase, we create a table and then begin storing and retrieving our data.

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 we 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 we have in the rack another spare server that’s configured, scaling is automatic.

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

* WAL: Write Ahead Log is a file on the distributed file system. The WAL is used to store new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
* BlockCache: is the read cache. It stores frequently read data in memory. Least Recently Used data is evicted when full.
* MemStore: is the write cache. It stores new data which has not yet been written to disk. It is sorted before writing to disk. There is one MemStore per column family per region.
* Hfiles store the rows as sorted KeyValues on disk.

****