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

**MemStore is a location in region server where the data is stored before it is flushed into HDFS.This is because when any data is written to HDFS, It is not directly written as it is cumbersome to write each time for a small input.Instead the Data will be first written in MemStore and we can set the flush size say 100 MB Now the data will be converted into a Immutable file(Read Only ) which is wriiten into Hdfs known as Hfile**

**data in the memstores is already sorted by keys matching exactly what**

**HFiles represent on disk, so no sorting or other special processing has to be performed**

**Further Since HDFS works well with Large Files rather than small files Since flushing memstores to disk causes more and more HFiles to be created, HBase has a housekeeping mechanism that merges the files into larger ones using *compaction*.**

**There are two types of compaction: *minor compactions* and *major compactions*.**

**The former reduce the number of storage files by rewriting smaller files into fewer but**

**larger ones, performing an *n*-way merge. Since all the data is already sorted in each**

**HFile, that merge is fast and bound only by disk I/O performance.**

**The *major compactions* rewrite all files within a column family for a region into a single**

**new one. They also have another distinct feature compared to the minor compactions:**

**based on the fact that they scan all key/value pairs, they can drop deleted entries including**

**their deletion marker.**

**Describe compactions in HBase.**

**Further Since HDFS works well with Large Files rather than small files Since flushing memstores to disk causes more and more HFiles to be created, HBase has a housekeeping mechanism that merges the files into larger ones using *compaction*.**

**There are two types of compaction:**

**1. *minor compactions***

***2.major compactions*.**

***minor compactions***

**The former reduce the number of storage files by rewriting smaller files into fewer but**

**larger ones, performing an *n*-way merge. Since all the data is already sorted in each**

**HFile, that merge is fast and bound only by disk I/O performance.**

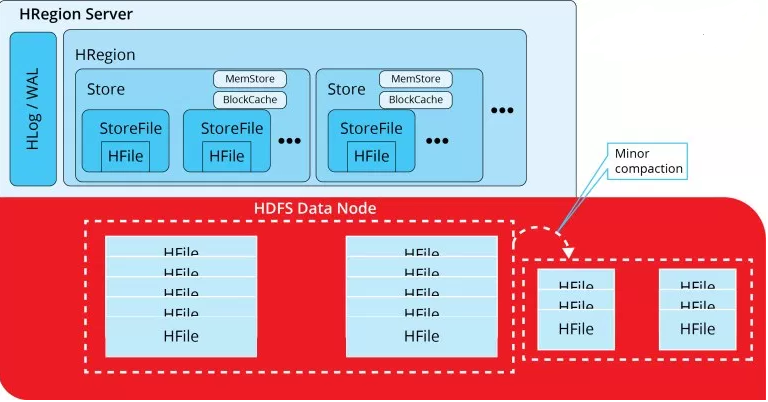
**Here are the various processes involved in Minor Compaction:**

**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.**

****

***major compactions***

**The *major compactions* rewrite all files within a column family for a region into a single**

**new one. They also have another distinct feature compared to the minor compactions:**

**based on the fact that they scan all key/value pairs, they can drop deleted entries including**

**their deletion marker.**

**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**

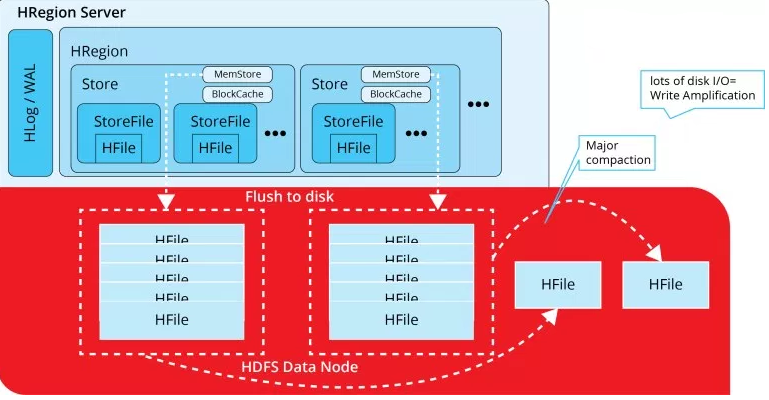
**Increase read performance of newly created Hfile.**

**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.**

****

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

**1.Tables**

**2.row**

**3.Column Family**

**4.Column Qualifier**

**5.cell**

**5.Versiom**

**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?**

**Actually it is not possible to insert a data without rowkey because**

**The Hbase stores data as akey value pair where Key is row KEY**

**This is How Hbase stores data**

SortedMap<RowKey, List<SortedMap<Column, List<Value, Timestamp>>>>

So it is not possible to store data without rowkey

**How can we use MapReduce with HBase?  
Apache MapReduce is a software framework used to analyze large amounts of data, and is the framework used most often with Apache Hadoop. HBase can be used as a data source, TableInputFormat, and data sink, TableOutputFormat or MultiTableOutputFormat, for MapReduce jobs. MapReduce jobs using Hbase can be done by subclass TableMapper and/or TableReducer.**

**Example for reducer class that extends tablereducer**

public static class Reducer1 extends TableReducer<ImmutableBytesWritable, IntWritable, ImmutableBytesWritable> {

public void reduce(ImmutableBytesWritable key, Iterable<IntWritable> values, Context context)

throws IOException, InterruptedException {

int sum = 0;

for (IntWritable val : values) {

sum += val.get();

}

Put put = new Put(key.get());

put.add(Bytes.toBytes("details"), Bytes.toBytes("total"), Bytes.toBytes(sum));

System.out.println(String.format("stats : key : %d, count : %d", Bytes.toInt(key.get()), sum));

context.write(key, put);

}

}

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

**There are four data model operations**

**1.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#get%28org.apache.hadoop.hbase.client.Get%29)**.**

**2.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#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).**

**3.Scan**

**Scan scans and prints the entire table contents or a particular row value.**

**4.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#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.**

**Filters In Hbase**

**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.**

**Differernt  types of filters applied in hbase are**

**KeyOnlyFilter**

**It gives only key of keyvalue pair. No arguments needed.**

**FirstKeyOnlyFilter**

**This filter doesntt take any arguments. It returns only the first key-value from each row.**

**PrefixFilter**

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

**ColumnPrefixFilter**

**This returns all the key value pairs of the columns with specific prefix mentioned in the argumernt. It takes only one argument.**

**MultipleColumnPrefixFilter**

**Similar to prefix filter but it takes multiple arguments for prefixes and returns all the rows that which are matching with any of the prefix arguments.**

**ColumnCountGetFilter**

**This filter takes one argument  a limit. It returns the first limit number of columns in the table.**

**PageFilter**

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

**InclusiveStopFilter**

**It returns all key-values present in rows up to and including the specified row. The specified row is mentioned in the argument.**

**Family Filter**

**It compares using the operartor. It compares the whole data with given column family for the given operator if the value is true then it will return that key value.**

**Value Filter**

**This is similar to family filter except here it compares the value and not the column family.**

**SingleColumnValueFilter**

**This filter takes a column family, a qualifier, a compare operator and a comparator. If the specified column is not found – all the columns of that row will be emitted. If the column is found and the comparison with the comparator returns true, all the columns of the row will be emitted. If the condition fails, the row will not be emitted**

**What is regionserver**

**Region Server**

**● Region Server actually stores data.**

**● Region server does both the work of reading and writing data into the table.**

**● Write Ahead Log (WAL) gives fault tolerant feature, which is also known as Hlog.**

**● A region of a table is served to the client by a Region Server.**

**● A region server can serve about 1,000 regions (which may belong to the same table or different tables).**

**● When accessing data, the clients communicate with HBase Region Servers directly.**

**● Region Server has BlockCach, which is a read cache that frequently stores the read data in memory. Least Recently Used data is removed when the cache is full.**

**● Region Server have MemStore, which is the write cache. It keeps the new data which has not yet been written to disk. There is one MemStore available per column family per region.**

**● Hfiles store the rows as sorted KeyValues on disk.**

**● Multiple Hfile make 1 region.**

