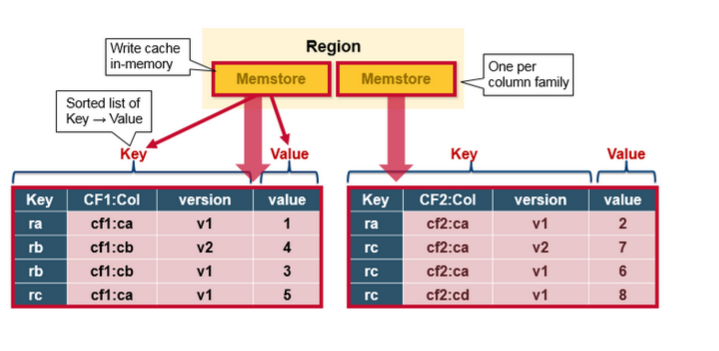
Explain in brief

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

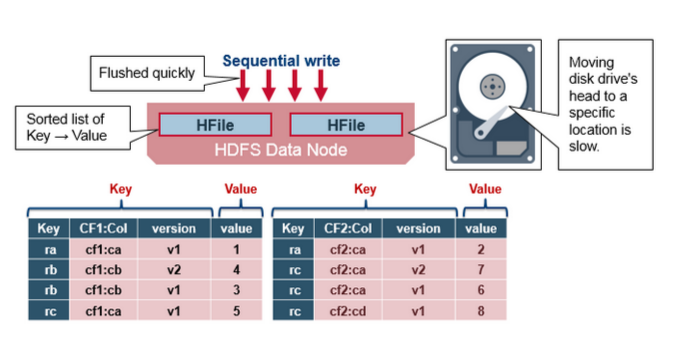
*MemStore*

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.



*Hbase Hfile*

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.



● Describe compactions in HBase.

Compaction, the process by which HBase cleans up after itself, comes in two flavors: major and minor.

*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.

*Major compaction* is different from the perspective of a system impact. However, the compaction is quite important to the overall functionality of the HBase system. A major compaction seeks to combine all HFiles into one large HFile.

In addition, a major compaction does the cleanup work after a user deletes a record. When a user issues a Delete call, the HBase system places a marker in the key-value pair so that it can be permanently removed during the next major compaction.

Additionally, because major compactions combine all HFiles into one large HFile, the time is right for the system to review the versions of the data and compare them against the time to live (TTL) property. Values older than the TTL are purged.

● 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.

*Table*: HBase organizes data into tables. Table names are Strings and composed of characters that are safe for use in a file system path.

*Row*: Within a table, data is stored according to its row. Rows are identified uniquely by their row key. Row keys do not have a data type and are always treated as a byte[ ] (byte array).

*Column Family*: Data within a row is grouped by column family. Column families also impact the physical arrangement of data stored in HBase. For this reason, they must be defined up front and are not easily modified. Every row in a table has the same column families, although a row need not store data in all its families. Column families are Strings and composed of characters that are safe for use in a file system path.

*Column Qualifier*: Data within a column family is addressed via its column qualifier, or simply, column. Column qualifiers need not be specified in advance. Column qualifiers need not be consistent between rows. Like row keys, column qualifiers do not have a data type and are always treated as a byte[ ].

*Cell*: A combination of row key, column family, and column qualifier uniquely identifies a cell. The data stored in a cell is referred to as that cell’s value. Values also do not have a data type and are always treated as a byte[ ].

*Timestamp*: Values within a cell are versioned. Versions are identified by their version number, which by default is the timestamp of when the cell was written. If a timestamp is not specified during a write, the current timestamp is used. If the timestamp is not specified for a read, the latest one is returned. The number of cell value versions retained by HBase is configured for each column family. The default number of cell versions is three.

*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. We cannot create a row without the rowkey as it is a key value pair. It will throw error.

● 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.

## Logical Operators, Comparison Operators and Comparators

Filters can be combined together with logical operators. Some filters take a combination of comparison operators and comparators. Following is the list of each.

**Logical Operators**

* AND - the key-value must pass both the filters to be included in the results.
* OR - the key-value must pass at least one of the filters to be included in the results.
* SKIP - for a particular row, if any of the key-values do not pass the filter condition, the entire row is skipped.
* WHILE - For a particular row, it continues to emit key-values until a key-value is reached that fails the filter condition.
* Compound Filters - Using these operators, a hierarchy of filters can be created. For example:
* (Filter1 AND Filter2)OR(Filter3 AND Filter4)

**Comparison Operators**

* LESS (<)
* LESS\_OR\_EQUAL (<=)
* EQUAL (=)
* NOT\_EQUAL (!=)
* GREATER\_OR\_EQUAL (>=)
* GREATER (>)
* NO\_OP (no operation)

**Comparators**

* BinaryComparator - lexicographically compares against the specified byte array using the Bytes.compareTo(byte[], byte[]) method.
* BinaryPrefixComparator - lexicographically compares against a specified byte array. It only compares up to the length of this byte array.
* RegexStringComparator - compares against the specified byte array using the given regular expression. Only EQUAL and NOT\_EQUAL comparisons are valid with this comparator.
* SubStringComparator - tests whether or not the given substring appears in a specified byte array. The comparison is case insensitive. Only EQUAL and NOT\_EQUAL comparisons are valid with this comparator.

● What are the data model operations in hBase?

The operations that are the basic blocks of data model are Get, Put, Scan, and Delete, using which we can read, write, and delete records from an HBase table.

*Get:*

The Get operation can fetch certain records from an HBase table. It is similar to the select [fields] where RowKey=<Row Key value here> statement in relational databases, where we fetch a row from the table.

The following is the representation of Get:

public Result get(Get get)throws IOException

*Put*

Put either adds new rows to a table or can update existing rows (if the key already exists). Puts are executed via HTable.put or HTable.batch.

*Scans*

Scan allow iteration over multiple rows for specified attributes.

● How can MapReduce be used with HBase?

An explanation is required of what TableMapReduceUtil is doing, especially with the reducer. TableOutputFormat is being used as the outputFormat class, and several parameters are being set on the config (e.g., TableOutputFormat.OUTPUT\_TABLE), as well as setting the reducer output key to ImmutableBytesWritable and reducer value to Writable. These could be set by the programmer on the job and conf, but TableMapReduceUtil tries to make things easier.

HBase distributed storage for stock price information

The example is going to process Apple stock prices downloaded from Yahoo finance web site, this is the same dataset – Apple stock prices – that we used previously to demonstrate Hive capabilities on Amazon Elastic MapReduce. It is stored in an AWS S3 bucket called stockprice. The MapReduce job will retrieve the file from there using s3n://AWS Access Key ID:AWS Secret Access Key//bucket/object url and will store the output in a HBase table called aapl\_marketdata. The test environment was based on Hadoop-0.20.2 and HBase-0.90.6.

The following is the example mapper, which will create a Put and matching the input Result and emit it. Note: this is what the CopyTable utility does.

public static class MyMapper extends TableMapper<ImmutableBytesWritable, Put> {

public void map(ImmutableBytesWritable row, Result value, Context context) throws IOException, InterruptedException {

// this example is just copying the data from the source table...

context.write(row, resultToPut(row,value));

}

private static Put resultToPut(ImmutableBytesWritable key, Result result) throws IOException {

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

for (KeyValue kv : result.raw()) {

put.add(kv);

}

return put;

}

}

There isn't actually a reducer step, so TableOutputFormat takes care of sending the Put to the target table.

This is just an example, developers could choose not to use TableOutputFormat and connect to the target table themselves.

Before the MapReduce job can be run, the table needs to be created

$ bin/hbase shell

HBase Shell; enter 'help' for list of supported commands.

Type "exit" to leave the HBase Shell

Version 0.90.6, r1295128, Wed Feb 29 14:29:21 UTC 2012

hbase(main):005:0> create 'aapl\_marketdata', 'marketdata'

0 row(s) in 1.4290 seconds

hbase(main):001:0> list

TABLE

aapl\_marketdata

1 row(s) in 0.3950 seconds

Now we are ready to run the MapReduce job. It is advisable to have a driver script to run your job and set all the required arguments in there for easier configuration but in essence it is just a plain old java code.

script looks like this:

$ cat hb.sh

java -classpath /home/ec2-user/hadoop/hadoop-0.20.2-ant.jar:/home/ec2-user/hadoop/hadoop-0.20.2-core.jar:/home/ec2-user/hadoop/hadoop-0.20.2-tools.jar:/home/ec2-user/hadoop/lib/jets3t-0.6.1.jar:/home/ec2-user/aws-java-sdk-1.3.11/aws-java-sdk-1.3.11.jar:/home/ec2-user/hbase/hbase-0.90.6.jar:/home/ec2-user/hbase/lib/commons-codec-1.4.jar:/home/ec2-user/hbase/lib/commons-httpclient-3.1.jar:/home/ec2-user/hbase/lib/commons-cli-1.2.jar:/home/ec2-user/hbase/lib/commons-logging-1.1.1.jar:/home/ec2-user/hbase/lib/zookeeper-3.3.2.jar:/home/ec2-user/hbase/lib/log4j-1.2.16.jar:json\_io\_1.0.4.jar:awsdemo-hbase.jar:/home/ec2-user/core-site.xml org.awsdemo.hbase.MarketDataApplication s3n://AWSAccessKeyId:AWSSecretAccessKey@stockprice/apple/input/APPL\_StockPrices.csv s3n://AWSAccessKeyId:AWSSecretAccessKey@stockprice/apple/output/

Once the MapReduce job was successfully finished, we can check the result in HBase table using bin/hbase shell.

hbase(main):001:0> get ‘table\_name’, ‘rowkey’

e.g. hbase(main):001:0> get ‘aapl\_marketdata’, ‘AAPL-1984-10-25’

hbase(main):001:0> get 'aapl\_marketdata', 'AAPL-1984-10-25'

COLUMN CELL

marketdata:daily timestamp=1341590928097, value={"@type":"org.apache.hadoop.io.MapWritable","@keys":[{"@type":"org.apache.hadoop.io

.Text","bytes":[115,116,111,99,107,83,121,109,98,111,108],"length":11},{"@type":"org.apache.hadoop.io.Text","bytes

":[115,116,111,99,107,80,114,105,99,101,76,111,119],"length":13},{"@type":"org.apache.hadoop.io.Text","bytes":[115

,116,111,99,107,80,114,105,99,101,79,112,101,110],"length":14},{"@type":"org.apache.hadoop.io.Text","bytes":[100,9

7,116,101],"length":4},{"@type":"org.apache.hadoop.io.Text","bytes":[115,116,111,99,107,80,114,105,99,101,67,108,1

11,115,101],"length":15},{"@type":"org.apache.hadoop.io.Text","bytes":[115,116,111,99,107,80,114,105,99,101,65,100

,106,67,108,111,115,101],"length":18},{"@type":"org.apache.hadoop.io.Text","bytes":[115,116,111,99,107,86,111,108,

117,109,101],"length":11},{"@type":"org.apache.hadoop.io.Text","bytes":[115,116,111,99,107,80,114,105,99,101,72,10

5,103,104],"length":14}],"@items":[{"@type":"org.apache.hadoop.io.Text","bytes":[65,65,80,76],"length":4},{"@type"

:"org.apache.hadoop.io.Text","bytes":[50,53,46,50,53],"length":5},{"@type":"org.apache.hadoop.io.Text","bytes":[50

,54,46,50,53],"length":5},{"@type":"org.apache.hadoop.io.Text","bytes":[49,57,56,52,45,49,48,45,50,53],"length":10

},{"@type":"org.apache.hadoop.io.Text","bytes":[50,53,46,50,53],"length":5},{"@type":"org.apache.hadoop.io.Text","

bytes":[50,46,56,56],"length":4},{"@type":"org.apache.hadoop.io.Text","bytes":[53,54,55,54,48,48,48],"length":7},{ "@type":"org.apache.hadoop.io.Text","bytes":[50,54,46,50,53],"length":5}]}

1 row(s) in 0.4140 seconds

The output was generated by JsonWriter and then serialized and stored in HBase, so it requires some ASCII skills to decode the values. E.g. “115,116,111,99,107,83,121,109,98,111,108” means stockSymbol, “115,116,111,99,107,80,114,105,99,101,76,111,119” means stockPriceLow, “115 ,116,111,99,107,80,114,105,99,101,79,112,101,110” means stockPriceOpen, etc. “65,65,80,76” means AAPL, “50,53,46,50,53” means 25.25, you know the rest.

You can also scan the entire table with Hbase shell using

hbase(main):001:0> scan 'aapl\_marketdata'

command. If you are done and want to get rid of the data, you need to disable the table and then you can drop it.

hbase(main):002:0> disable 'aapl\_marketdata'

0 row(s) in 2.1490 seconds

hbase(main):004:0> drop 'aapl\_marketdata'

0 row(s) in 1.1790 seconds

● What is regionserver?

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.

## *Region Server Components*

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.