|  |  |
| --- | --- |
| 7down vote[favorite](http://stackoverflow.com/questions/7681933/hbase-getting-all-timestamped-values-for-a-cell)  3 | i have the following scenario in my hbase instance  hbase(main):002:0> create 'test', 'cf'  0 row(s) in 1.4690 seconds  hbase(main):003:0> put 'test', 'row1', 'cf:a', 'value1'  0 row(s) in 0.1480 seconds  hbase(main):004:0> put 'test', 'row2', 'cf:b', 'value2'  0 row(s) in 0.0070 seconds  hbase(main):005:0> put 'test', 'row3', 'cf:c', 'value3'  0 row(s) in 0.0120 seconds  hbase(main):006:0> put 'test', 'row3', 'cf:c', 'value4'  0 row(s) in 0.0070 seconds  Now if you will see, the last two inserts are for the same column family, same column and same key. But if i understand hbase properly cf:c+row3 represent a cell which will have all timestamped versions of inserted value.  But a simple scan return only recent value  hbase(main):010:0> scan 'test'  ROW COLUMN+CELL  row1 column=cf:a, timestamp=1317945279379, value=value1  row2 column=cf:b, timestamp=1317945285731, value=value2  row3 column=cf:c, timestamp=1317945301466, value=value4  3 row(s) in 0.0250 seconds  How do i get all timestamped values for a cell, or how to perform time range based query? |

**Answer::**

scan 'test', {VERSIONS => 3}

will give you 2 versions of columns if they are available. you can use it in get aswell :

get 'test', 'row3', {COLUMN => 'cf:c', VERSIONS => 3}

for getting the value of a spesific time you can use TIMESTAMP aswell.

get 'test', 'row3', {COLUMN => 'cf:c', TIMESTAMP => 1317945301466}

if you need to get values "between" 2 timestamps you should use [TimestampsFilter](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/filter/TimestampsFilter.html).

**Delete From HBase table----**

Given below is the complete program to delete data from the HBase table.

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Delete;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.util.Bytes;

public class DeleteData {

public static void main(String[] args) throws IOException {

// Instantiating Configuration class

Configuration conf = HBaseConfiguration.create();

// Instantiating HTable class

HTable table = new HTable(conf, "employee");

// Instantiating Delete class

Delete delete = new Delete(Bytes.toBytes("row1"));

delete.deleteColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));

delete.deleteFamily(Bytes.toBytes("professional"));

// deleting the data

table.delete(delete);

// closing the HTable object

table.close();

System.out.println("data deleted.....");

}

}

Compile and execute the above program as shown below.

$javac Deletedata.java

$java DeleteData

Reading Data using HBase Shell

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Get;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Result;

import org.apache.hadoop.hbase.util.Bytes;

public class RetriveData{

public static void main(String[] args) throws IOException, Exception{

// Instantiating Configuration class

Configuration config = HBaseConfiguration.create();

// Instantiating HTable class

HTable table = new HTable(config, "emp");

// Instantiating Get class

Get g = new Get(Bytes.toBytes("row1"));

// Reading the data

Result result = table.get(g);

// Reading values from Result class object

byte [] value = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("name"));

byte [] value1 = result.getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));

// Printing the values

String name = Bytes.toString(value);

String city = Bytes.toString(value1);

System.out.println("name: " + name + " city: " + city);

}

}

**Update HBase –**

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.util.Bytes;

public class UpdateData{

public static void main(String[] args) throws IOException {

// Instantiating Configuration class

Configuration config = HBaseConfiguration.create();

// Instantiating HTable class

HTable hTable = new HTable(config, "emp");

// Instantiating Put class

//accepts a row name

Put p = new Put(Bytes.toBytes("row1"));

// Updating a cell value

p.add(Bytes.toBytes("personal"),

Bytes.toBytes("city"),Bytes.toBytes("Delih"));

// Saving the put Instance to the HTable.

hTable.put(p);

System.out.println("data Updated");

// closing HTable

hTable.close();

}

}

**Create Record –**

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.util.Bytes;

public class InsertData{

public static void main(String[] args) throws IOException {

// Instantiating Configuration class

Configuration config = HBaseConfiguration.create();

// Instantiating HTable class

HTable hTable = new HTable(config, "emp");

// Instantiating Put class

// accepts a row name.

Put p = new Put(Bytes.toBytes("row1"));

// adding values using add() method

// accepts column family name, qualifier/row name ,value

p.add(Bytes.toBytes("personal"),

Bytes.toBytes("name"),Bytes.toBytes("raju"));

p.add(Bytes.toBytes("personal"),

Bytes.toBytes("city"),Bytes.toBytes("hyderabad"));

p.add(Bytes.toBytes("professional"),Bytes.toBytes("designation"),

Bytes.toBytes("manager"));

p.add(Bytes.toBytes("professional"),Bytes.toBytes("salary"),

Bytes.toBytes("50000"));

// Saving the put Instance to the HTable.

hTable.put(p);

System.out.println("data inserted");

// closing HTable

hTable.close();

}

}

**Scan Table –**

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Result;

import org.apache.hadoop.hbase.client.ResultScanner;

import org.apache.hadoop.hbase.client.Scan;

public class ScanTable{

public static void main(String args[]) throws IOException{

// Instantiating Configuration class

Configuration config = HBaseConfiguration.create();

// Instantiating HTable class

HTable table = new HTable(config, "emp");

// Instantiating the Scan class

Scan scan = new Scan();

// Scanning the required columns

scan.addColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));

scan.addColumn(Bytes.toBytes("personal"), Bytes.toBytes("city"));

// Getting the scan result

ResultScanner scanner = table.getScanner(scan);

// Reading values from scan result

for (Result result = scanner.next(); result != null; result = Scanner.next())

System.out.println("Found row : " + result);

//closing the scanner

scanner.close();

}

}

Compile and execute the above program as shown below.

$javac ScanTable.java

$java ScanTable

The following should be the output:

Found row :

keyvalues={row1/personal:city/1418275612888/Put/vlen=5/mvcc=0,

row1/personal:name/1418035791555/Put/vlen=4/mvcc=0}

**WAL :::  
\*\*\*\*\*\*\*\*\*\***

With this feature enabled, a writer of HDFS can guarantee that data will be persisted by invoking a flush call.So, HBase can guarantee that when a region server dies, data can be recovered and replayed

on other region servers using its Write-Ahead Log (WAL).

**Using Mapreduce Read & Write in HBase::**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

package com.impetus.hbase.write;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.fs.Path;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.client.Result;

import org.apache.hadoop.hbase.client.Scan;

import org.apache.hadoop.hbase.io.ImmutableBytesWritable;

import org.apache.hadoop.hbase.mapreduce.TableMapReduceUtil;

import org.apache.hadoop.hbase.mapreduce.TableMapper;

import org.apache.hadoop.hbase.mapreduce.TableReducer;

import org.apache.hadoop.hbase.util.Bytes;

import org.apache.hadoop.io.IntWritable;

import org.apache.hadoop.io.Text;

import org.apache.hadoop.mapreduce.Job;

import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

/\*\*

\*

\* @author ubuntu

\*

\* hbase(main):052:0> create 'reducetable', 'cf'

0 row(s) in 1.1100 seconds

hbase(main):053:0> scan reducetable

NameError: undefined local variable or method `reducetable' for #<Object:0x1f54a25>

hbase(main):054:0> scan 'reducetable'

ROW COLUMN+CELL

Bangupdated column=cf:count, timestamp=1432044303877, value=1

calcutta column=cf:count, timestamp=1432044303877, value=1

chennai column=cf:count, timestamp=1432044303877, value=1

clacutta column=cf:count, timestamp=1432044303877, value=2

hyderabad column=cf:count, timestamp=1432044303877, value=1

5 row(s) in 0.0190 seconds

\*/

public class HBaseWriterDriver {

public static class MyMapper extends TableMapper<Text, IntWritable> {

private final IntWritable ONE = new IntWritable(1);

private Text text = new Text();

public void map(ImmutableBytesWritable row, Result value,

Context context) throws IOException, InterruptedException {

String val = new String(value.getValue(Bytes.toBytes("personal data"),

Bytes.toBytes("city")));

// get rowKey and convert it to string

String inKey = new String(row.get());

text.set(val); // we can only emit Writables...

context.write(text, ONE);

}

}

/\* public static class MyTableCombiner extends

TableReducer<Text, IntWritable, Text> {

private final IntWritable iw = new IntWritable();

public void reduce(Text key, Iterable<IntWritable> values,

Context context) throws IOException, InterruptedException {

int i = 0;

for (IntWritable val : values) {

i += val.get();

}

iw.set(i);

context.write(key, iw);

}

}\*/

public static class MyTableReducer extends

TableReducer<Text, IntWritable, ImmutableBytesWritable> {

public void reduce(Text key, Iterable<IntWritable> values,

Context context) throws IOException, InterruptedException {

int i = 0;

for (IntWritable val : values) {

i += val.get();

}

Put put = new Put(Bytes.toBytes(key.toString()));

put.add(Bytes.toBytes("cf"), Bytes.toBytes("count"),

Bytes.toBytes("" + i));

context.write(null, put);

}

}

public static void main(String[] args) throws Exception {

Configuration config = HBaseConfiguration.create();

Job job = new Job(config, "ExampleSummary");

job.setJarByClass(HBaseWriterDriver.class); // class that contains

// mapper and reducer

String sourceTable = "emp";

String targetTable = "reducetable";

Scan scan = new Scan();

scan.setCaching(500); // 1 is the default in Scan, which will be bad for

// MapReduce jobs

scan.setCacheBlocks(false); // don't set to true for MR jobs

// set other scan attrs

TableMapReduceUtil.initTableMapperJob(sourceTable, // input table

scan, // Scan instance to control CF and attribute selection

MyMapper.class, // mapper class

Text.class, // mapper output key

IntWritable.class, // mapper output value

job);

//job.setCombinerClass(MyTableCombiner.class);

TableMapReduceUtil.initTableReducerJob(targetTable, // output table

MyTableReducer.class, // reducer class

job);

job.setNumReduceTasks(1);

boolean b = job.waitForCompletion(true);

if (!b) {

throw new IOException("error with job!");

}

}

}  
  
**HBase scanners** are like cursors in a traditional database or Java iterators, except—

unlike the latter—they have to be closed after use. Scanners return rows in order. Users

obtain a scanner on an HBase table by calling HTable.getScanner(scan), where the

scan parameter is a configured instance of a Scan object. In the Scan instance, you can

pass the row at which to start and stop the scan, which columns in a row to return in

the row result, and optionally, a filter to run on the server side.9 The ResultScanner

interface, which is returned when you call HTable.getScanner(), is as follows:

public interface ResultScanner extends Closeable, Iterable<Result> {

public Result next() throws IOException;

public Result [] next(int nbRows) throws IOException;

public void close();

}

You can ask for the next row’s results or a number of rows. Each invocation of

next() involves a trip back to the regionserver, so grabbing a bunch of rows at once can

make for significant performance savings.10

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

A common problem in database processing is quickly finding the most recent version of a value. A technique using reverse timestamps as a part of the key can help greatly with a special case of this problem. Also found in the HBase chapter of Tom White's book Hadoop: The Definitive Guide (O'Reilly), the technique involves appending (Long.MAX\_VALUE - timestamp) to the end of any key, e.g., [key][reverse\_timestamp].

The most recent value for [key] in a table can be found by performing a Scan for [key] and obtaining the first record. Since HBase keys are in sorted order, this key sorts before any older row-keys for [key] and thus is first.

This technique would be used instead of using [Section 6.4, “ Number of Versions ”](http://hbase.apache.org/0.94/book/schema.versions.html) where the intent is to hold onto all versions "forever" (or a very long time) and at the same time quickly obtain access to any other version by using the same Scan technique.

To always get the latest cells from a scan all you need to do is use the method below -

Result.getColumnLatest(family, qualifier)

**we have hbase table with row key as AccountId and unixtimestamp.**

**eg: ACNTID1359694800000**

**Account Id: ACNTID**

**unixtimestamp: 1359694800000**

**1359694800000 is the value for 2/1/2013**

**I am looking for a query for Account Ids on a give date? can i use startrow, stop row logic. Any other ways?**

Your rowkey structure doesn't support getting account id's for any given unix timestamp, since the timestamp in your case is in the righter most part of the rowkey or atleast not using STARTROW & STOPROW alone. To get the desired result, your query should scan all rowkeys of the table and do the filtering for the given timestamp . HBase comes with a Filter called [RowFilter](https://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/filter/RowFilter.html), used along with[Scan](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/Scan.html), to restrict the rows returned by HBase. Since you store the rowkey as text, you can use[SubStringComparator](https://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/filter/SubstringComparator.html) or [RegexStringComparator](https://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/filter/RegexStringComparator.html) along with the [RowFilter](https://hbase.apache.org/0.94/apidocs/org/apache/hadoop/hbase/filter/RowFilter.html). The command line equivalent of this is,

scan 'table\_name', { FILTER =>"RowFilter(=,'substring:1359694800000')"}

The above command will return all66 rows that has 1359694800000 in their rowkey.

\_http://www.ibm.com/support/knowledgecenter/en/SSPT3X\_4.2.0/com.ibm.swg.im.infosphere.biginsights.analyze.doc/doc/bigsql\_TuneHbase.html

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

HBase sorts the data and stores - unlike RDBMS which never sorts.

Sorting helps super-fast lookups in HBase. Thats all.

But then, lookups which dont obey the row-key ordering (esepcially composite row-keys) - dont run fast. They can be utterly slow.

And, at that point of time, HBase will need secondary index etc..

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# What is the fastest way to load 1TB of data in HBase?

There are two ways to ingest data into HBase:  
1. Using the put(..) API call  
2. Bulkload HFiles directly  
  
For a quick 1 TB ingestion into an empty table, bulkloads is likely your best option. There is documentation available on how to do bulk loads @[Apache HBase ™ Reference Guide](http://hbase.apache.org/book.html).

Writing a custom MapReduce job that writes data in HBase's internal data format (using HFileOutputFormat), and then directly loads the generated StoreFiles into a running cluster (completebulkload).

#### **Preparing data via a MapReduce job**

The first step of a bulk load is to generate HBase data files (StoreFiles) from a MapReduce job using HFileOutputFormat2. This output format writes out data in HBase’s internal storage format so that they can be later loaded very efficiently into the cluster.

In order to function efficiently, HFileOutputFormat2 must be configured such that each output HFile fits within a single region. In order to do this, jobs whose output will be bulk loaded into HBase use Hadoop’s TotalOrderPartitioner class to partition the map output into disjoint ranges of the key space, corresponding to the key ranges of the regions in the table.

HFileOutputFormat2 includes a convenience function, configureIncrementalLoad(), which automatically sets up a TotalOrderPartitioner based on the current region boundaries of a table.

#### **71.3.2. Completing the data load**

After a data import has been prepared, either by using the importtsv tool with the “importtsv.bulk.output” option or by some other MapReduce job using the HFileOutputFormat, the completebulkload tool is used to import the data into the running cluster. This command line tool iterates through the prepared data files, and for each one determines the region the file belongs to. It then contacts the appropriate RegionServer which adopts the HFile, moving it into its storage directory and making the data available to clients.

If the region boundaries have changed during the course of bulk load preparation, or between the preparation and completion steps, the completebulkload utility will automatically split the data files into pieces corresponding to the new boundaries. This process is not optimally efficient, so users should take care to minimize the delay between preparing a bulk load and importing it into the cluster, especially if other clients are simultaneously loading data through other means.

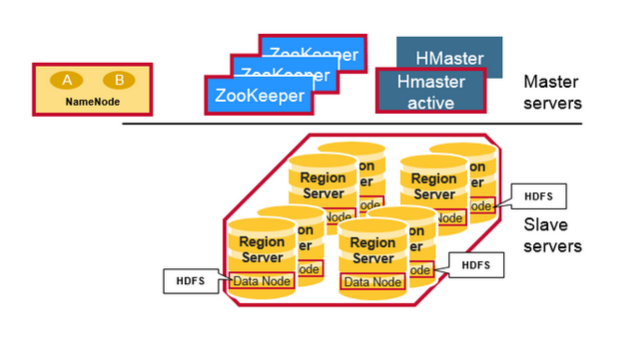
$ hadoop jar hbase-server-VERSION.jar completebulkload [-c /path/to/hbase/config/hbase-site.xml] /user/todd/myoutput mytable

## HBase Architectural Components

Physically, HBase is composed of three types of servers in a master slave type of architecture. Region servers serve data for reads and writes. When accessing data, clients communicate with HBase RegionServers directly. Region assignment, DDL (create, delete tables) operations are handled by the HBase Master process. Zookeeper, which is part of HDFS, maintains a live cluster state.

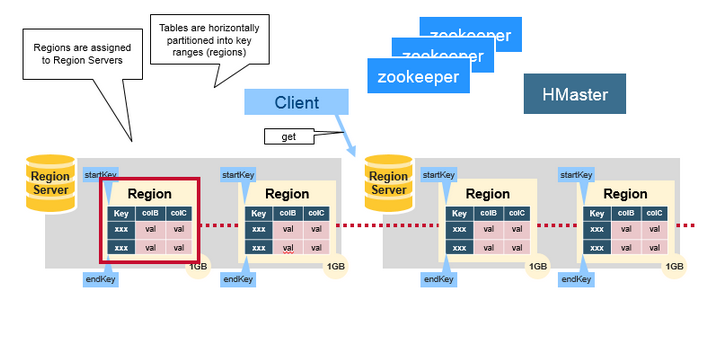
The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality (putting the data close to where it is needed) for the data served by the RegionServers. HBase data is local when it is written, but when a region is moved, it is not local until compaction.

The NameNode maintains metadata information for all the physical data blocks that comprise the files.



## Regions

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.



## HBase HMaster

Region assignment, DDL (create, delete tables) operations are handled by the HBase Master.

A master is responsible for:

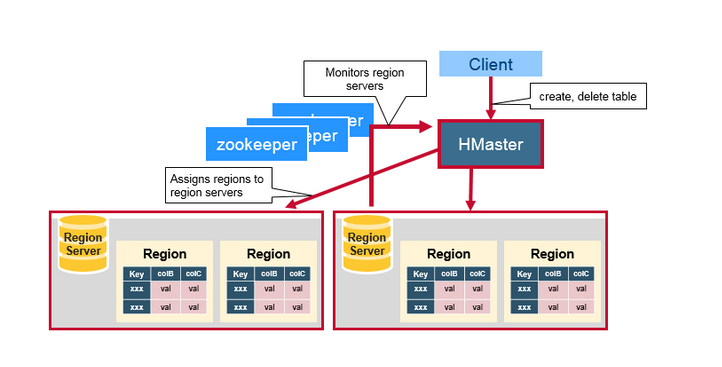
* Coordinating the region servers

- Assigning regions on startup , re-assigning regions for recovery or load balancing

- Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper)

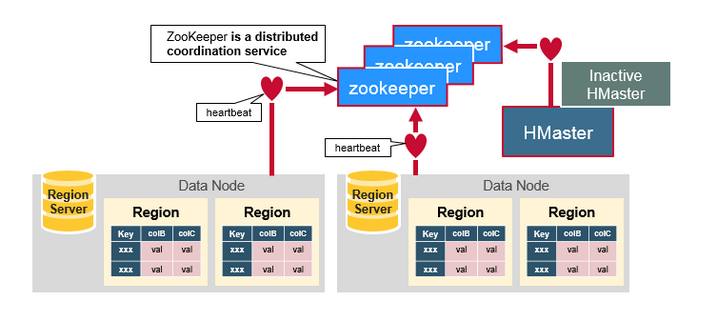
* Admin functions

- Interface for creating, deleting, and updating tables



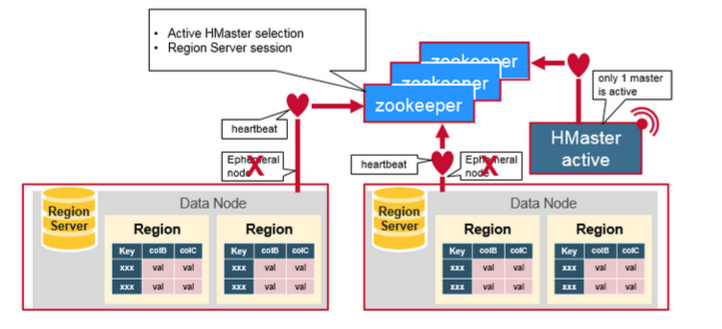
## ZooKeeper: The Coordinator

HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification. Zookeeper uses consensus to guarantee common shared state. Note that there should be three or five machines for consensus.



## How the Components Work Together

Zookeeper is used to coordinate shared state information for members of distributed systems. Region servers and the active HMaster connect with a session to ZooKeeper. The ZooKeeper maintains ephemeral nodes for active sessions via heartbeats.



Each Region Server creates an ephemeral node. The HMaster monitors these nodes to discover available region servers, and it also monitors these nodes for server failures. HMasters vie to create an ephemeral node. Zookeeper determines the first one and uses it to make sure that only one master is active. The active HMaster sends heartbeats to Zookeeper, and the inactive HMaster listens for notifications of the active HMaster failure.

If a region server or the active HMaster fails to send a heartbeat, the session is expired and the corresponding ephemeral node is deleted. Listeners for updates will be notified of the deleted nodes. The active HMaster listens for region servers, and will recover region servers on failure. The Inactive HMaster listens for active HMaster failure, and if an active HMaster fails, the inactive HMaster becomes active.

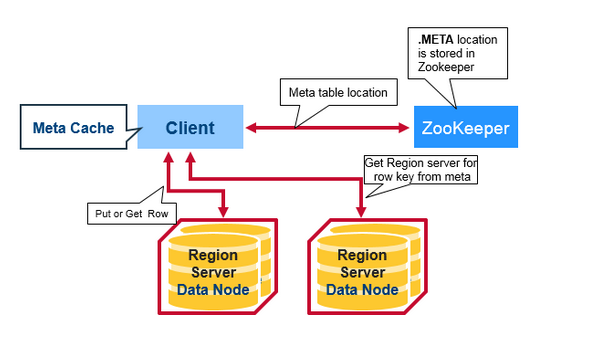
## HBase First Read or Write

There is a special HBase Catalog table called the META table, which holds the location of the regions in the cluster. ZooKeeper stores the location of the META table.

This is what happens the first time a client reads or writes to HBase:

1. The client gets the Region server that hosts the META table from ZooKeeper.
2. The client will query the .META. server to get the region server corresponding to the row key it wants to access. The client caches this information along with the META table location.
3. It will get the Row from the corresponding Region Server.

For future reads, the client uses the cache to retrieve the META location and previously read row keys. Over time, it does not need to query the META table, unless there is a miss because a region has moved; then it will re-query and update the cache.

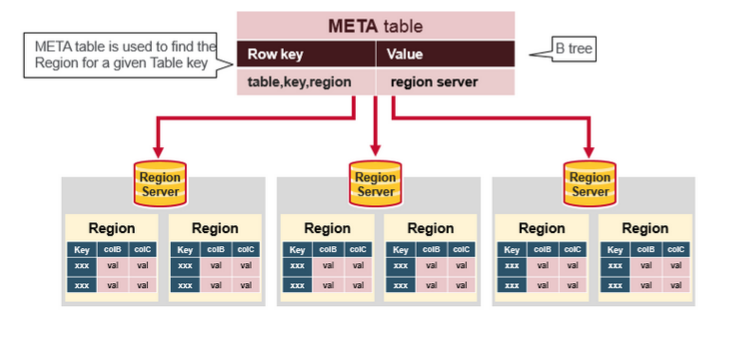


## HBase Meta Table

* This META table is an HBase table that keeps a list of all regions in the system.
* The .META. table is like a b tree.
* The .META. table structure is as follows:

- Key: region start key,region id

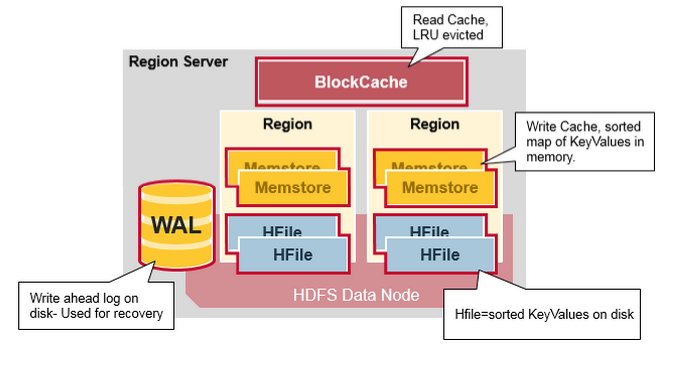
- Values: RegionServer



## 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.

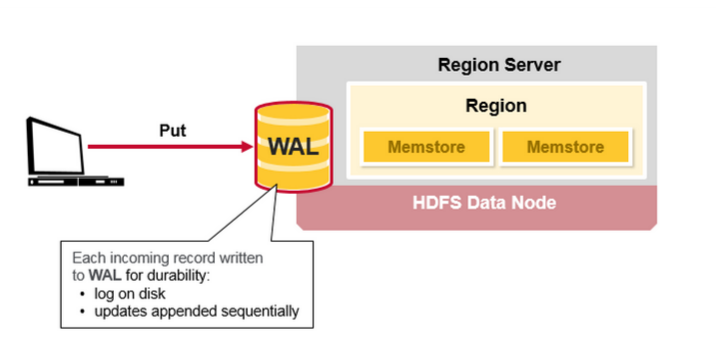


## HBase Write Steps (1)

When the client issues a Put request, the first step is to write the data to the write-ahead log, the WAL:

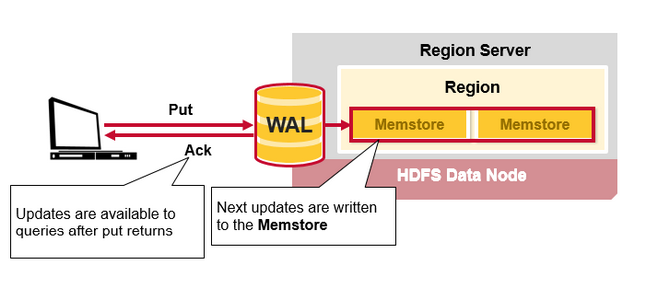
- Edits are appended to the end of the WAL file that is stored on disk.

- The WAL is used to recover not-yet-persisted data in case a server crashes.



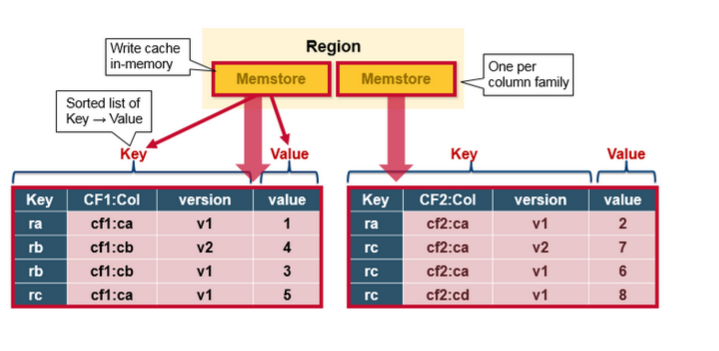
## HBase Write Steps (2)

Once the data is written to the WAL, it is placed in the MemStore. Then, the put request acknowledgement returns to the client.



## 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.

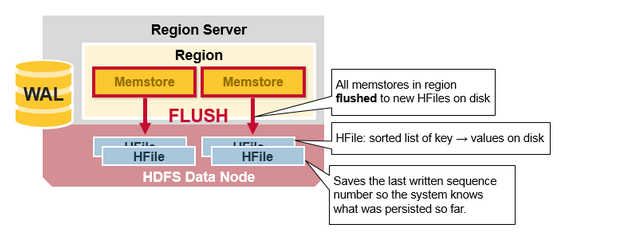


## HBase Region Flush

When the MemStore accumulates enough data, the entire sorted set is written to a new HFile in HDFS. HBase uses multiple HFiles per column family, which contain the actual cells, or KeyValue instances. These files are created over time as KeyValue edits sorted in the MemStores are flushed as files to disk.

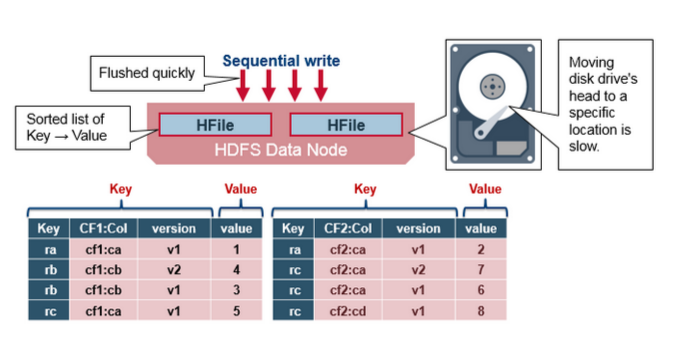
Note that this is one reason why there is a limit to the number of column families in HBase. There is one MemStore per CF; when one is full, they all flush. It also saves the last written sequence number so the system knows what was persisted so far.

The highest sequence number is stored as a meta field in each HFile, to reflect where persisting has ended and where to continue. On region startup, the sequence number is read, and the highest is used as the sequence number for new edits.



## 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.

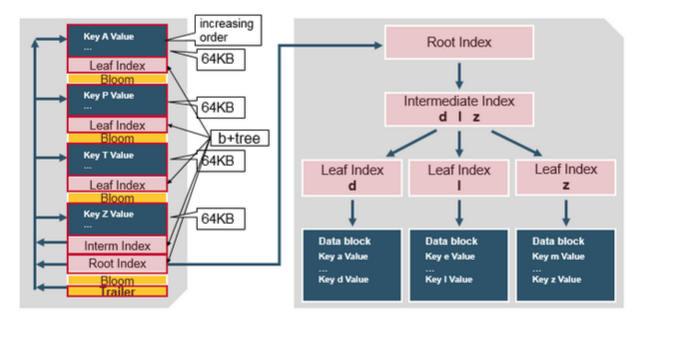


## HBase HFile Structure

An HFile contains a multi-layered index which allows HBase to seek to the data without having to read the whole file. The multi-level index is like a b+tree:

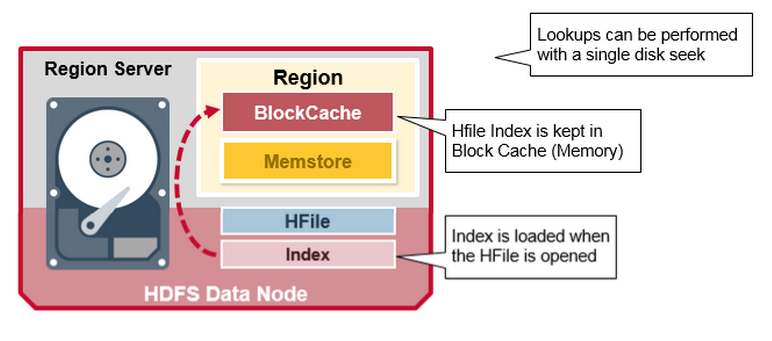
* Key value pairs are stored in increasing order
* Indexes point by row key to the key value data in 64KB “blocks”
* Each block has its own leaf-index
* The last key of each block is put in the intermediate index
* The root index points to the intermediate index

The trailer points to the meta blocks, and is written at the end of persisting the data to the file. The trailer also has information like bloom filters and time range info. Bloom filters help to skip files that do not contain a certain row key. The time range info is useful for skipping the file if it is not in the time range the read is looking for.



## HFile Index

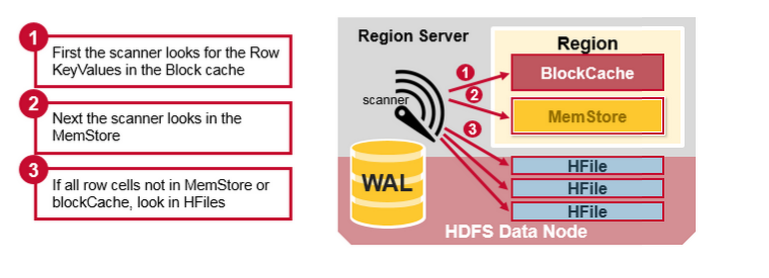
The index, which we just discussed, is loaded when the HFile is opened and kept in memory. This allows lookups to be performed with a single disk seek.



## HBase Read Merge

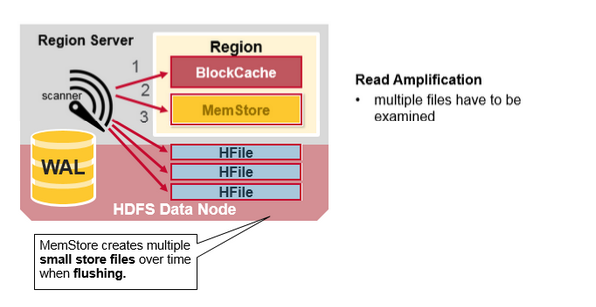
We have seen that the KeyValue cells corresponding to one row can be in multiple places, row cells already persisted are in Hfiles, recently updated cells are in the MemStore, and recently read cells are in the Block cache. So when you read a row, how does the system get the corresponding cells to return? A Read merges Key Values from the block cache, MemStore, and HFiles in the following steps:

1. First, the scanner looks for the Row cells in the Block cache - the read cache. Recently Read Key Values are cached here, and Least Recently Used are evicted when memory is needed.
2. Next, the scanner looks in the MemStore, the write cache in memory containing the most recent writes.
3. If the scanner does not find all of the row cells in the MemStore and Block Cache, then HBase will use the Block Cache indexes and bloom filters to load HFiles into memory, which may contain the target row cells.



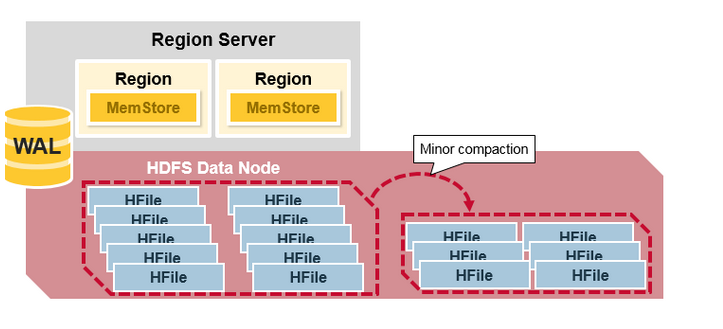
## HBase Read Merge

As discussed earlier, there may be many HFiles per MemStore, which means for a read, multiple files may have to be examined, which can affect the performance. This is called read amplification.



## HBase Minor Compaction

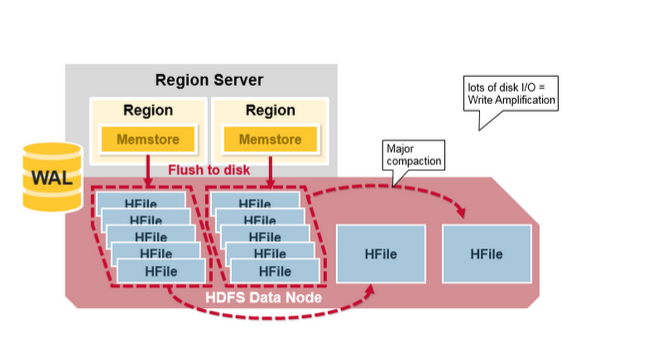
HBase will automatically pick some smaller HFiles and rewrite them into fewer bigger Hfiles. This process is called minor compaction. Minor compaction reduces the number of storage files by rewriting smaller files into fewer but larger ones, performing a merge sort.



## HBase Major Compaction

Major compaction merges and rewrites all the HFiles in a region to one HFile per column family, and in the process, drops deleted or expired cells. This improves read performance; however, since major compaction rewrites all of the files, lots of disk I/O and network traffic might occur during the process. This is called write amplification.

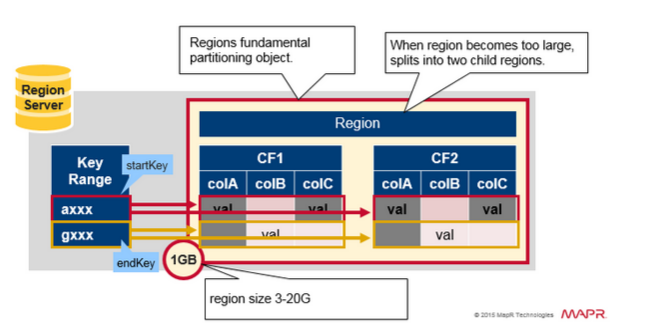
Major compactions can be scheduled to run automatically. Due to write amplification, major compactions are usually scheduled for weekends or evenings. Note that MapR-DB has made improvements and does not need to do compactions. A major compaction also makes any data files that were remote, due to server failure or load balancing, local to the region server.



## Region = Contiguous Keys

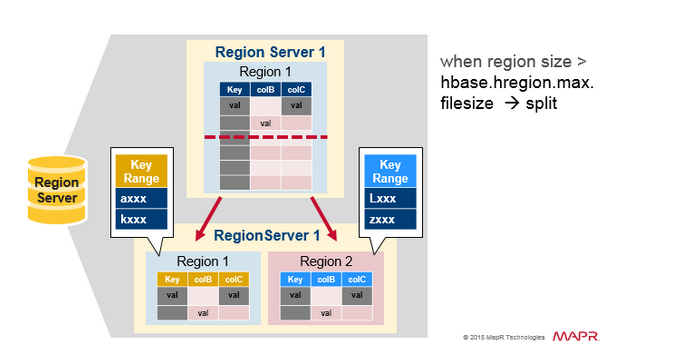
Let’s do a quick review of regions:

* A table can be divided horizontally into one or more regions. A region contains a contiguous, sorted range of rows between a start key and an end key
* Each region is 1GB in size (default)
* A region of a table is served to the client by a RegionServer
* A region server can serve about 1,000 regions (which may belong to the same table or different tables)



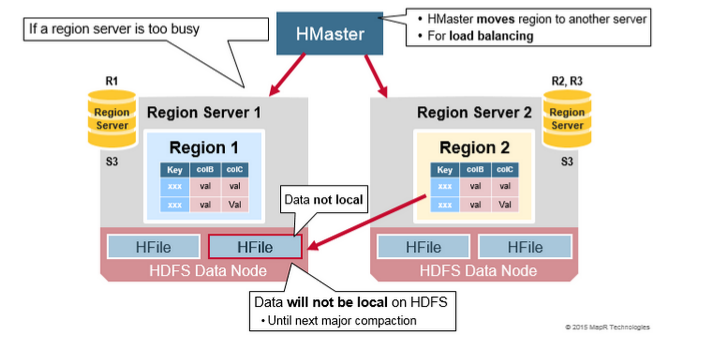
## Region Split

Initially there is one region per table. When a region grows too large, it splits into two child regions. Both child regions, representing one-half of the original region, are opened in parallel on the same Region server, and then the split is reported to the HMaster. For load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers.



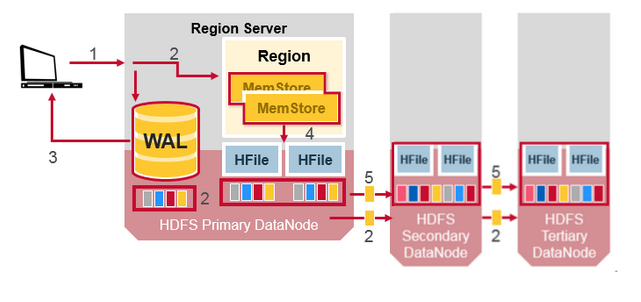
## Read Load Balancing

Splitting happens initially on the same region server, but for load balancing reasons, the HMaster may schedule for new regions to be moved off to other servers. This results in the new Region server serving data from a remote HDFS node until a major compaction moves the data files to the Regions server’s local node. HBase data is local when it is written, but when a region is moved (for load balancing or recovery), it is not local until major compaction.



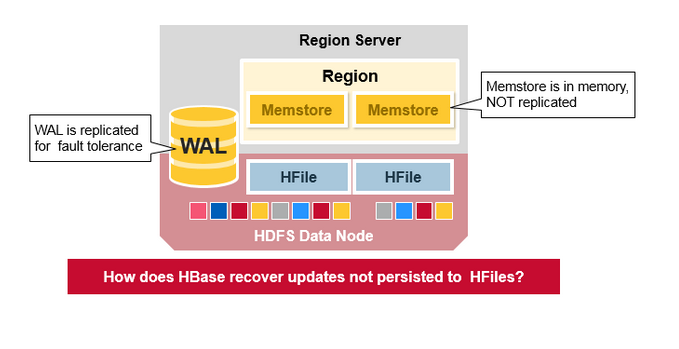
## HDFS Data Replication

All writes and Reads are to/from the primary node. HDFS replicates the WAL and HFile blocks. HFile block replication happens automatically. HBase relies on HDFS to provide the data safety as it stores its files. When data is written in HDFS, one copy is written locally, and then it is replicated to a secondary node, and a third copy is written to a tertiary node.



## HDFS Data Replication (2)

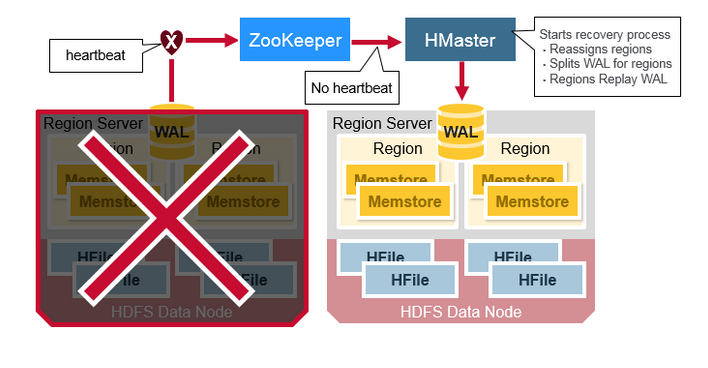
The WAL file and the Hfiles are persisted on disk and replicated, so how does HBase recover the MemStore updates not persisted to HFiles? See the next section for the answer.



## HBase Crash Recovery

When a RegionServer fails, Crashed Regions are unavailable until detection and recovery steps have happened. Zookeeper will determine Node failure when it loses region server heart beats. The HMaster will then be notified that the Region Server has failed.

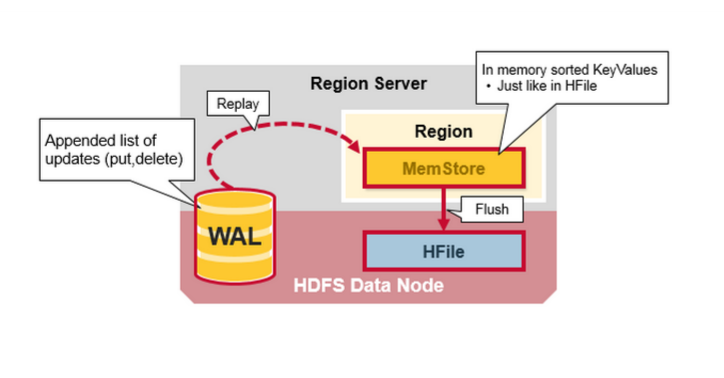
When the HMaster detects that a region server has crashed, the HMaster reassigns the regions from the crashed server to active Region servers. In order to recover the crashed region server’s memstore edits that were not flushed to disk. The HMaster splits the WAL belonging to the crashed region server into separate files and stores these file in the new region servers’ data nodes. Each Region Server then replays the WAL from the respective split WAL, to rebuild the memstore for that region.



## Data Recovery

WAL files contain a list of edits, with one edit representing a single put or delete. Edits are written chronologically, so, for persistence, additions are appended to the end of the WAL file that is stored on disk.

What happens if there is a failure when the data is still in memory and not persisted to an HFile? The WAL is replayed. Replaying a WAL is done by reading the WAL, adding and sorting the contained edits to the current MemStore. At the end, the MemStore is flush to write changes to an HFile.



## Apache HBase Architecture Benefits

HBase provides the following benefits:

* **Strong consistency model**

- When a write returns, all readers will see same value

* **Scales automatically**

- Regions split when data grows too large

- Uses HDFS to spread and replicate data

* **Built-in recovery**

- Using Write Ahead Log (similar to journaling on file system)

* **Integrated with Hadoop**

- MapReduce on HBase is straightforward

## Apache HBase Has Problems Too…

* **Business continuity reliability:**

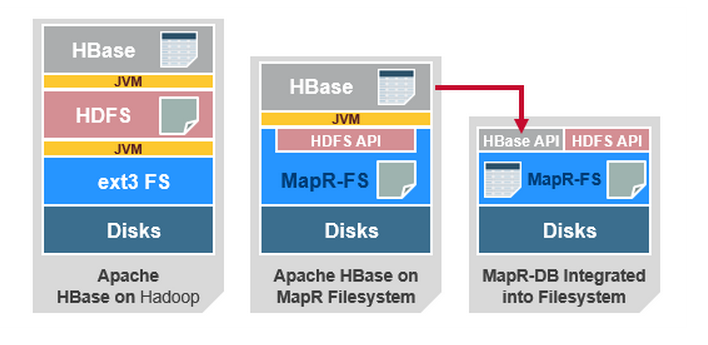
- WAL replay slow

- Slow complex crash recovery

- Major Compaction I/O storms

## MapR-DB with MapR-FS does not have these problems

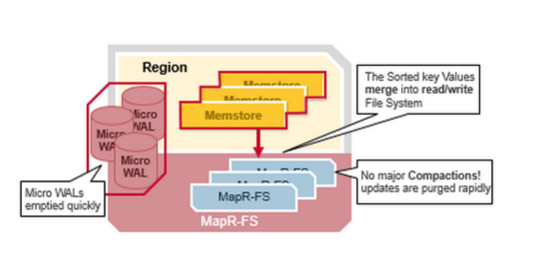
The diagram below compares the application stacks for Apache HBase on top of HDFS on the left, Apache HBase on top of MapR's read/write file system MapR-FS in the middle, and MapR-DB and MapR-FS in a Unified Storage Layer on the right.



MapR-DB exposes the same HBase API and the Data model for MapR-DB is the same as for Apache HBase. However the MapR-DB implementation integrates table storage into the MapR file system, eliminating all JVM layers and interacting directly with disks for both file and table storage.

MapR-DB offers many benefits over HBase, while maintaining the virtues of the HBase API and the idea of data being sorted according to primary key. MapR-DB provides operational benefits such as no compaction delays and automated region splits that do not impact the performance of the database. The tables in MapR-DB can also be isolated to certain machines in a cluster by utilizing the topology feature of MapR. The final differentiator is that MapR-DB is just plain fast, due primarily to the fact that it is tightly integrated into the MapR file system itself, rather than being layered on top of a distributed file system that is layered on top of a conventional file system.

## Key differences between MapR-DB and Apache HBase



* Tables part of the MapR Read/Write File system
  + Guaranteed data locality
* Smarter load balancing
  + Uses container Replicas
* Smarter fail over
  + Uses container replicas
* Multiple small WALs
  + Faster recovery
* Memstore Flushes Merged into Read/Write File System
  + No compaction !

[You can take this free On Demand training to learn more about MapR-FS and MapR-DB](https://www.mapr.com/training/hadoop-demand-training/hde-110)

In this blog post, you learned more about the HBase architecture and its main benefits over NoSQL data store solutions. If you have any questions about HBase, please ask them in the comments section below.

**To check Hbase Status –**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**http://hbase.apache.org/book.html#perf.writing**

**What is the difference between get and scan in HBase?**

**would a get(...) provide any performance benefit over the scan?**

Get operates directly on a particular row identified by the rowkey passed as a parameter to the the Get instance. While Scan operates on all the rows, if you haven't used range query by providing start and end rowkeys to your Scan instance. Clearly it is more efficient if you know it beforehand which row to operate on. You can directly go there and perform the desired operation.

**How to verify that the columns are in-memory(of course, the describe statement and the browser reflect it) and accessed from there and not the disk?**

You can use **isInMemory()** method provided by **HColumnDescriptor** to verify if a particular CF is in-memory or not. But, you cannot find out that the entire table is in memory and whether fetch is happening from disk or the memory. Although in-memory blocks have the highest priority, but it is not 100% sure that everything is in-memory all the time. One important thing here is that data is persisted to disk even in case of in-memory CF.

**Is the from-memory or from-disk read transparent to the client? In simple words, do I need to change the HTable access code in my reducer class? If yes, what are the changes?**

Yes. It is totally transparent. You don't have to do anything extra.

Get operation returns single row from hbase table where as scan returns set of rows depending upon your search conditions.  
If you are looking for faster retrieval of data from hbase table then you should look for get rather than scan. But again it depends upon how your rowkey is formed.

On the backend HRegion both Scan and Get amount to *nearly* the same thing. They both end up executed by HRegion.RegionScannerImpl. Note below that the get() within that class instantiates a RegionScanner - similarly to invoking a Scan.

org.apache.hadoop.hbase.regionserver.HRegion.RegionScannerImpl

public List<Cell> get(Get get, boolean withCoprocessor)

throws IOException {

List<Cell> results = new ArrayList<Cell>();

// pre-get CP hook

if (withCoprocessor && (coprocessorHost != null)) {

if (coprocessorHost.preGet(get, results)) {

return results;

}

}

Scan scan = new Scan(get);

In the case of a get(), only a single row is returned - by invoking scanner.next() one time:

RegionScanner scanner = null;

try {

scanner = getScanner(scan);

scanner.next(results);

-When I use a partial key scan on that table to retrieve a value for a given key I get almost constant time value retrieval.  
-When I use a Get, the time taken is far greater than with the scan. However when I looked inside the code, I found that Get itself uses a Scan.when you issue a Get, there is a scan happening behind the scenes."Each time a get or a scan is issued, HBase scan (sic) through each file to find the result."When you compare a partial key scan and a get, remember that the row key you use for Get can be a much longer string than the partial key you use for the scan.  
In that case, for the Get, HBase has to do a deterministic lookup to ascertain the exact location of the row key that it needs to match and fetch it. But with the partial key, HBase does not need to lookup the exact key match, and just needs to find the more approximate location of that key prefix.  
The answer for this is: it depends. I think it will depend on:

1. Your row key "schema" or composition
2. The length of the Get key and the Scan prefix
3. How many regions you have and possibly other factors.

**HBASE RowKey Design?**

1. **Avoid using a timestamp or a sequence (e.g. 1, 2, 3) as the row-key**. With monotonically increasing row-keys (i.e., using a timestamp) **pile-up on a single region**. It can be mitigated by randomizing the input records to not be in sorted order.

## Try to minimize row and column sizes Or why are my StoreFile indices large? Column mapping: one to many

You can map a single HBase entity (row key or a column) to multiple SQL columns. This kind of mapping is called one to many. HBase stores a lot of information for each value. If you stored each SQL column individually, the required storage space would be very large. For the best performance, put columns that are queried together into a single dense HBase column to help reduce the data that is fetched from HBase. A dense column is a single HBase column that maps to multiple SQL columns.

For example, if table T1 has nine columns with 1.5 million rows. and you use a one-to-one mapping, this table requires 522 MB of storage. However, if table T1 uses a one-to-many mapping, the table requires only 276 MB of storage.

Queries on columns that are mapped one-to-many perform better than one-to-one queries. A one-to-many model maximizes predicate pushdown. When querying back, the query returns the entire key for each value. For example, assume the following query:

SELECT \* FROM T1;

| *Table 1. Comparing column mapping models* | |
| --- | --- |
| **Column mapping** | **Response time** |
| One to one mapping | 1m 31 s |
| One to many mapping | 1m 2s |

For the best performance, put the column that you use the most in a query predicate as the leading part of the column, since only SQL predicates that are mapped to a leading HBase column are pushed down. If you have other columns on which you usually query, define indexes on those other columns. An index can be defined on a single column or a set of columns.

Design your rowkeys in such a way that query predicates can be pushed down. Only predicates on the leading rowkeys are pushed down.

Do not use monotonically increasing values or time series data as row keys. For example, in the [ORDERS table](http://www.ibm.com/support/knowledgecenter/en/SSPT3X_4.0.0/com.ibm.swg.im.infosphere.biginsights.analyze.doc/doc/bigsql_designhints.html?view=kc#concept_mxc_3j1_km__cf_orders), do not use the O\_ORDERKEY as the leading rowkey. This design can cause a single region to be a hot spot since all of the key values are in close proximity to each other and will belong to the same region.

1. Try to keep the ColumnFamily names as small as possible, preferably one character (e.g. "d" for data/default).

An HBase table is made of column families which are the logical and physical grouping of columns. The columns in one family are stored separately from the columns in another family. If you have data that is not often queried, assign that data to a separate column family.

Because column families are stored in separate HFiles, keep the number of column families as small as possible. You also want to **reduce the number of column families to reduce the frequency of MemStore flushes, and the frequency of compactions**. And, by using the **smallest number of column families possible, you can improve the LOAD time and reduce disk consumption.**

1. Although verbose attribute names (e.g., "myVeryImportantAttribute") are easier to read, prefer shorter attribute names (e.g., "via") to store in HBase.

Keep them as short as is reasonable such that they can still be useful for required data access (e.g., Get vs. Scan). A short key that is useless for data access is not better than a longer key with better get/scan properties. Expect tradeoffs when designing rowkeys.

1. **A long is 8 bytes. You can store an unsigned number up to 18,446,744,073,709,551,615 in those eight bytes. If you stored this number as a String -- presuming a byte per character -- you need nearly 3x the bytes.**

Not convinced? Below is some sample code that you can run on your own.

// long

//

long l = 1234567890L;

byte[] lb = Bytes.toBytes(l);

System.out.println("long bytes length: " + lb.length); // returns 8

String s = "" + l;

byte[] sb = Bytes.toBytes(s);

System.out.println("long as string length: " + sb.length); // returns 10

// hash

//

MessageDigest md = MessageDigest.getInstance("MD5");

byte[] digest = md.digest(Bytes.toBytes(s));

System.out.println("md5 digest bytes length: " + digest.length); // returns 16

String sDigest = new String(digest);

byte[] sbDigest = Bytes.toBytes(sDigest);

1. **Reverse Timestamps**

A common problem in database processing is quickly finding the most recent version of a value. A technique using reverse timestamps as a part of the key can help greatly with a special case of this problem. Also found in the HBase chapter of Tom White's book Hadoop: The Definitive Guide (O'Reilly), the technique involves appending (Long.MAX\_VALUE - timestamp) to the end of any key, e.g., [key][reverse\_timestamp].

The most recent value for [key] in a table can be found by performing a Scan for [key] and obtaining the first record. Since HBase keys are in sorted order, this key sorts before any older row-keys for [key] and thus is first.

1. Immutability of Rowkeys

Rowkeys cannot be changed. The only way they can be "changed" in a table is if the row is deleted and then re-inserted. This is a fairly common question on the HBase dist-list so it pays to get the rowkeys right the first time (and/or before you've inserted a lot of data).

**HBase is optimized for reads when data is queried on basis of row key.** Query predicates are applied to row key as start and stop keys. Query engines such as Impala, Hive and BigInsights reading data from HBase translate predicates against efficient row key lookup when operators such as =, <, BETWEEN are applied against row key. Reading data from HBase on the basis of rowkey is somewhat similar to using index for reading a RDBMS table. If you use the index in your query, you read only required records, else you scan entire table.

In one of the implementations, HBase table contained 50 million records and the query was written to retrieve 10 records. We were using Impala external tables to read HBase. One would expect this query to be very fast, but it was taking a long time. We noticed that the query had used 'LIKE' condition against the rowkey and hence the predicate was not getting pushed down against the rowkey. HBase was scanning millions of records and then returning 10 records. Changing 'LIKE' to 'BETWEEN' did the magic and the query execution time came down from 2 minutes to 1.5 seconds.

Let us get down to details.

Our HBase table was a time series table with "Parameter","Value" and "Timestamp" as columns. The rowkey was a concatenation of "Parameter-Timestamp". An external table was created in Impala against this HBase table with columns "Rowkey","Parameter","Value" and "Timestamp". Rowkey was defined as a string column. Our objective of the query is to find sum of values for a specific parameter (say P11) for a specific range of timestamp.   
All the below queries resulted in HBase table scan and performed inefficiently.

* SELECT SUM(VALUE) FROM TABLE WHERE PARAMETER = 'P11' AND TIMESTAMP BETWEEN 'T1' AND 'T2'; (rowkey not used)
* SELECT SUM(VALUE) FROM TABLE WHERE ROWKEY LIKE 'P11%' AND PARAMETER = 'P11' AND TIMESTAMP BETWEEN 'T1' AND 'T2';

Correct query to make use of rowkey lookup is like this:

* SELECT SUM(VALUE) FROM TABLE WHERE ROWKEY BETWEEN 'P10' AND 'P12' AND PARAMETER = 'P11' AND TIMESTAMP BETWEEN 'T1' AND 'T2';

Similarly comparing rowkey against a non constant value will result in scanning of entire table.

**Summary :**

1. If you plan to scan to entire HBase table or majority of it, probably you are using HBase for wrong purpose. HBase is most efficient for reading a single row or a range of rows.
2. If you are not using a filter against rowkey column in your query, your rowkey design may be wrong. The row key should be designed to contain the information you need to find specific subsets of data.
3. When creating external tables in Hive / Impala against HBase tables, map the HBase rowkey against a string column in Hive / Impala. If this is not done, rowkey is not used in the query and entire table is scanned.
4. Carefully evaluate your query to check if it will result in rowkey predicates. Use of "LIKE" against rowkey column does not result in rowkey predicate and it results in scanning entire table.

**You can optimize scans of HBase tables by modifying some properties.**

**hbase.client.scanner.caching**

This parameter, which is set in file hbase-site.xml, is the number of rows that are fetched when calling next on a scanner if it is not served from (local, client) memory. Higher caching values enable faster scanners but use more memory and some calls of next can take longer times when the cache is empty.

This value is important if data in your HBase table is used without any HBase row key based lookups, or when your query looks for wide range scans (wide rowkey lookups).

You can modify this parameter in these ways:

**As a property in hbase-site.xml:**

1. <property>
2. <name>hbase.client.scanner.caching</name>
3. <value>10000</value>
4. </property>

**From a SET command:**

SET HADOOP PROPERTY hbase.client.scanner.caching=10000

**Region size tuning (hbase.hregion.max.filesize)**

HBase region size is important because when accessing HBase data, a map reduce split is a region.

If the region size is too large, there is not sufficient parallelism in the map reduce jobs. If the region size is too small, there are many wasted cycles in creating and tearing down the map reduce tasks.

An optimal size depends on the workload and the cluster configuration.

This value is important if data in your HBase table is used without any HBase row key based lookups, or when your query looks for wide range scans (wide rowkey lookups).

Ref:\_ <http://www.ibm.com/support/knowledgecenter/en/SSPT3X_4.0.0/com.ibm.swg.im.infosphere.biginsights.analyze.doc/doc/bigsql_TuneHbase.html>

Disater Recovery in HBase –

http://bruteforcedata.blogspot.com/2012/08/hbase-disaster-recovery-and-whisky.html

**Is there a limit to the number of columns in an HBase row?**

**Lock granularity**  
When you do an operation within a row, the RegionServer code briefly holds a lock on that row while applying the mutation.  
On the plus side, this means that you can act atomically on several columns - concurrent readers will either see the entire update or won't see the update at all. They shouldn't (barring one or two bugs we're still stomping on) see a partial update.  
On the minus side, this means that the throughput of write operations within a single row is limited (probably a few hundred per second).  
We're currently working on some optimizations for specific cases like increment so that multiple incrementers can "line up" behind the lock and then batch their addition together into a single transaction.  
**Region distribution**  
The unit of load balancing and distribution is the region, and a row will never be split across regions. So, no matter how hot a row is, it will always be served by a single server. If the data were split across many rows, you could force a split in between two hot rows to distribute load between two hosts.  
**Bugs**  
In prior versions of HBase there were some bugs where we would accidentally load or deserialize an entire row into RAM. So if your row is very large (100s of MBs) you may have run into serious performance issues, OOMEs, etc. I think most of these bugs are since squashed, and the RS does a smart job of only loading the necessary columns, but it's something to be aware of.  
**Summary**In summary, if you don't need to do atomic operations across multiple cells, probably better to make a "tall" data layout.

1. On the number of column families

HBase currently does not do well with anything above two or three column families so keep the number of column families in your schema low. **Currently, flushing and compactions are done on a per Region basis** so if one column family is carrying the bulk of the data bringing on flushes, the adjacent families will also be flushed though the amount of data they carry is small. **When many column families the flushing and compaction interaction can make for a bunch of needless i/o loading** (To be addressed by changing flushing and compaction to work on a per column family basis). For more information on compactions, see compaction.

**Try to make do with one column family if you can in your schemas. Only introduce a second and third column family in the case where data access is usually column scoped; i.e. you query one column family or the other but usually not both at the one time.**

33.1. Cardinality of ColumnFamilies

Where multiple ColumnFamilies exist in a single table, be aware of the cardinality (i.e., number of rows). If ColumnFamilyA has 1 million rows and ColumnFamilyB has 1 billion rows, ColumnFamilyA’s data will likely be spread across many, many regions (and RegionServers). This makes mass scans for ColumnFamilyA less efficient.

One good example would be to have an analytics table with Daily, Monthly, Yearly and Total column families, each one with their own TTL settings (expiration) and columns for each date ranges (days, months, years...), they're different scopes and when you query the table, you usually fetch only one type of aggregation at a time, i.e.: retrieve daily stats of last 30 days

# **A million rows isn’t cool. You know what’s cool? A billion rows.**

If you’re just getting started doing analytic work with SQL on Hadoop, a table with a million rows might seem like a good starting point for experimentation. Isn’t that a lot of data? While you can exercise the features of a traditional database with a million rows, for Hadoop it’s not nearly enough. Think billions of rows instead.

Let’s look at the ways a million-row table falls short. Understanding the data volumes involved with big data can help you avoid going down unproductive pathways based on misleading assumptions.

With a million-row table, every byte in each row represents a megabyte of total data volume. Let’s say your table represents people and has fields for name, address, occupation, salary, height, weight, number of children, and favorite food. Here’s what a sample field might look like, with a scale underneath to illustrate length:

John_Russel_1_data_scale

This particular record takes up 78 characters, including the comma separators. A back-of-the-envelope calculation suggests that, if this is an average row, we’ll end up with about 78 megabytes of data in the table. (And don’t recycle that envelope just yet — doing analytics with Hadoop, you’ll do a lot of rough estimates like this to sanity-check your expectations about performance and scalability.)

Let’s do a thought experiment using a query that does a full-table scan, something like:

SELECT AVG(salary) FROM million\_row\_table;

This query considers every row in the table, so there are no shortcuts to avoid reading all 78 megabytes. Is 78 MB a lot of data to process with Hadoop on a parallel cluster of computers? Not at all — in fact, it doesn’t hit the minimum threshold to ensure any parallel execution at all for the query: the whole thing can fit into a single data block.

To see how our understanding of disk storage has evolved, let’s take a brief spin back at block sizes throughout history:

|  |  |
| --- | --- |
| Commodore 64 | Disk blocks were 256 bytes. (Am I dating myself by admitting that I used to  edit them in hex?) |
| Oracle Database | Default block size has traditionally been 4 KB. |
| MySQL Database | Default block size for InnoDB storage engine is 16 KB. |
| Dedicated data warehousing appliances | 64 MB is a popular block size. (That’s a huge jump from 16 KB) |
| Hadoop | Typical block size for HDFS is 128 MB, for example in recent versions of the CDH  distro from Cloudera. |

Even an experienced data wrangler might be surprised to realize that today on Hadoop, the million-row table we’re talking about doesn’t even fill up a single data block.

That fact is significant because when breaking down the work to execute SQL queries in parallel on Hadoop, the smallest unit of measure is the block. If you have a 5-node cluster, how many nodes are needed to process 78 MB? Just one. How about if you have a 10-node cluster? All the I/O can still be done by one node. Sure, it’s nice to have all that spare capacity in the cluster, but the actual work of your query doesn’t get done any faster.

Of course, you can split up the original data into more than one file. A file that’s smaller than 128 MB will just consist of one HDFS block — just a smaller-than-normal one. But keep in mind that a Hadoop cluster with fast disk drives can read a 128 MB block in the blink of an eye. If you split up the million-row table into multiple files, you can give the other nodes in the cluster something to do, but performance-wise, it’s still the blink of an eye. The overhead to send requests back and forth on the network roughly balances out the performance benefit from doing such a small query in parallel across multiple nodes.

Therefore, however you slice it, our hypothetical million-row table doesn’t pose a good challenge for interactive SQL on Hadoop. You won’t learn much about performance or scalability with it. You won’t take advantage of the capabilities of your Hadoop cluster. You won’t get useful performance numbers running benchmark queries against it.

The preceding discussion shows how even in the best and simplest case, this million-row table doesn’t come close to the volume we normally deal with for big data. Let’s continue the thought experiment by looking at how this volume of data would factor into more realistic scenarios. (Spoiler: it looks even less substantial when you start applying other best practices such as binary file formats and partitioning.)

In practice, you would store your data in an efficient binary format such as Parquet, instead of as a bulky text file with millions of commas and requiring six bytes to encode a six-figure salary. In Parquet, the numbers are compacted in binary format, repeated values are encoded to minimize repetition, columns with a modest number of different values are encoded using numeric IDs, and the result is compressed one final time at the end. The original 78 MB of text might be packed down to 20-30 MB, or even less. Now you would need tens of millions of rows to get close to filling up an HDFS block. Remember, the ultimate goal is for the data to span multiple blocks, to give the cluster a worthwhile amount of work to do in parallel. That goal would be even harder to achieve because, to take advantage of all this encoding and compression, Parquet data blocks are typically 256 MB, 512 MB, or even 1 GB.

For SQL queries in Hadoop, most people experiment with partitioned tables. Partitioning physically divides the data — for example: by year, month, day — resulting in smaller individual data files. For all the same reasons why a million rows isn’t very much data for a regular table, a million rows also isn’t very much for a partition in a partitioned table.

Now, I hope anyone with a million-row table is not feeling bad. You can still use them quite well as part of big data analytics, just in the appropriate context.

For example, maybe you are a librarian with a million items in your catalog, or maybe you operate a factory that has produced a million items, or maybe you are a successful business with a million customers. The key is to find some more granular data set (in data warehousing terms, a dimension table) to correlate against the million-row table.

The library analyzes every reservation, checkout, and book return to optimize their purchasing process and the inventory for each branch. The factory records information about every component and every step in the manufacturing process, to find more efficient manufacturing methods with fewer defects. The business analyzes every purchase or other interaction from each customer to design a more personalized experience. All of this fine-grained data goes into tables with billions of rows, and that’s where you really see the benefits of interactive SQL queries on a Hadoop cluster.

When you start working with SQL on Hadoop and dealing with performance and scalability aspects of join queries, you’ll hear tuning advice regarding “big tables” and “small tables” involved in the query. Just remember, the million-row table that once seemed so impressive is probably the small table in this scenario.

Now, go out and look for some tables with billions of rows. With your SQL skills and some Hadoop infrastructure backing you, you’ll be able to handle them with ease.

1. **How HBase Store the data? What is Lexicographical order?**

**Answer::**

Rowkeys in HBase are kept sorted [lexicographically](http://en.wikipedia.org/wiki/Lexicographical_order) irrespective of the insertion order. This means that they are sorted based on their string representations. Rowkeys in HBase are treated as an array of bytes having a string representation. The lowest order rowkey appears first in a table. That's why 10 appears before 2 and so on. See the sections **Rows** on this [page](http://wiki.apache.org/hadoop/Hbase/DataModel) to know more about this.

When you left pad the integers with zeros their natural ordering is kept intact while sorting lexicographically and that's why you see the scan order same as the order in which you had inserted the data.

In [mathematics](http://en.wikipedia.org/wiki/Mathematics), the **lexicographic or lexicographical order** (also known as lexical order, dictionary order, alphabetical order or lexicographic(al) product) is a generalization of the way the [alphabetical order](http://en.wikipedia.org/wiki/Alphabetical_order) of words is based on the alphabetical order of their component letters.

1. **How will you insert 1 million record in HBase without using tool?**

**Answer::**

Mapreduce HBase Integration.

I have HBase table with column name. Say I need to find out all the records which is starting with name ‘rajan’

I have university, college, department. What is the best way to store the data in Hive? What is the best way to store it using java Collection.

**Problem** :- In many situations while solving a business problem we need to do a prefix scan on a HBase table  for example consider the following scenario :-

Considering we have a “Customer"  table for which we have   row-key like  <customer\_id>\_<transaction\_id>\_<timestamp> =>  “CST1024\_TX2724981\_1479249799770”

This table is initially  designed  for key based fast lookup for a real-time (Storm) process which works fine .

However In some scenario we have to get all transactions for a customer using the prefix filter like =>   scan ‘customer', {ROWPREFIXFILTER => ‘CST1024’}

This query is really slow and was making the real-time process slower since the data is distributed and it has to performance a cluster wide search.

**Solution** :-  Salting the row keys  to help in this scenario , salting the HBase  tables helps from preventing the hotspots by distributing the data based on salt prefix . The same concept can be used to improve prefix scan , we had to redesign our HBase key like following :-

                   Hashcode(<customer\_id>)|<customer\_id>\_<transaction\_id>\_<timestamp> =>  “33465|CST1024\_TX2724981\_1479249799770”

Appending the hash-code or a number at beginning to HBase key provides a hint  to  HBase  for a region-specific key ranges , thus all the data specific to a customer\_id would fall in specific region .

After the above changes in row-key the prefix scan =>   scan ‘customer', {ROWPREFIXFILTER => ‘33465|CST1024’}

Works 10X faster since the salt-key in query works as a region hint for searching all the of a customer .  This helped us a lot to scale the performance of our real-time pipeline.

This effectively does this (and also works for binary situations)  
scan 'mytable', {STARTROW => 'abc', ENDROW => 'abd'}

This method is a LOT more efficient than the "PrefixFilter" approach because the latter puts all records through the comparison code the is present in this PrefixFilter class.

### **Hbase shell and Commands**

**Hbase Install**  
  
$ mkdir hbase-install  
$ cd hbase-install  
$ wget <http://apache.claz.org/hbase/hbase-0.92.1/hbase-0.92.1.tar.gz>  
$ tar xvfz hbase-0.92.1.tar.gz  
$HBASE\_HOME/bin/start-hbase.sh  
  
**configuration changes in Hbase**<property>  
<name>hbase.rootdir</name>  
<value>file:///home/user/myhbasedirectory/</value>  
</property>  
  
$ hbase shell  
hbase(main):001:0> list  
TABLE  
0 row(s) in 0.5710 seconds  
  
**Create a table**  
hbase(main):002:0> create 'mytable', 'cf'  
hbase(main):003:0> list  
TABLE  
mytable  
1 row(s) in 0.0080 seconds  
  
**WRITING DATA**  
hbase(main):004:0> put 'mytable', 'first', 'cf:message', 'hello HBase'  
  
**READING DATA**  
hbase(main):007:0> get 'mytable', 'first'  
hbase(main):008:0> scan 'mytable'  
  
**describe table**  
hbase(main):003:0> describe 'users'  
DESCRIPTION ENABLED  
{NAME => 'users', FAMILIES => [{NAME => 'info', true ,BLOOMFILTER => 'NONE', REPLICATION\_SCOPE => '0 , COMPRESSION => 'NONE', VERSIONS => '3', TTL=> '2147483647',   
BLOCKSIZE => '65536', IN\_MEMORY => 'false', BLOCKCACHE => 'true'}]}  
1 row(s) in 0.0330 seconds  
  
**Configurable block size**  
hbase(main):002:0> create 'mytable',{NAME => 'colfam1', BLOCKSIZE => '65536'}  
  
**Block cache**  
workloads don’t benefit from putting data into a read cache—for instance, if a certain table or column family in a table is only accessed for sequential scans or isn’t  
accessed a lot and you don’t care if Gets or Scans take a little longer.By default, the block cache is enabled. You can disable it at the time of table creation  
or by altering the table:  
hbase(main):002:0> create 'mytable',{NAME => 'colfam1', BLOCKCACHE => 'false’}  
  
**Aggressive caching**  
You can choose some column families to have a higher priority in the block cache (LRU cache). This comes in handy if you expect more random reads on one column family compared to another. This configuration is also done at table-instantiation time:  
hbase(main):002:0> create 'mytable',  
{NAME => 'colfam1', IN\_MEMORY => 'true'}  
The default value for the IN\_MEMORY parameter is false.  
  
**Bloom filters**  
hbase(main):007:0> create 'mytable',{NAME => 'colfam1', BLOOMFILTER => 'ROWCOL'}  
The default value for the BLOOMFILTER parameter is NONE. A row-level bloom filter is enabled with ROW, and a qualifier-level bloom filter is enabled with ROWCOL. The rowlevel bloom filter checks for the non-existence of the particular rowkey in the block,and the qualifier-level bloom filter checks for the non-existence of the row and column qualifier combination. The overhead of the ROWCOL bloom filter is higher than that of the ROW bloom filter.  
  
**TTL (Time To Live)**  
can set the TTL while creating the table like this:  
hbase(main):002:0> create 'mytable', {NAME => 'colfam1', TTL => '18000'}  
This command sets the TTL on the column family colfam1 as 18,000 seconds = 5 hours. Data in colfam1 that is older than 5 hours is deleted during the next major compaction.  
  
**Compression**  
can enable compression on a column family when creating tables like this:  
hbase(main):002:0> create 'mytable',  
{NAME => 'colfam1', COMPRESSION => 'SNAPPY'}  
Note that data is compressed only on disk. It’s kept uncompressed in memory (Mem-Store or block cache) or while transferring over the network.  
  
**Cell versioning**  
Versions are also configurable at a column family level and can be specified at  
the time of table instantiation:  
hbase(main):002:0> create 'mytable', {NAME => 'colfam1', VERSIONS => 1}  
hbase(main):002:0> create 'mytable',  
{NAME => 'colfam1', VERSIONS => 1, TTL => '18000'}  
hbase(main):002:0> create 'mytable', {NAME => 'colfam1', VERSIONS => 5,  
MIN\_VERSIONS => '1'}  
  
**Description of a table**  
hbase(main):004:0> describe 'follows'  
DESCRIPTION ENABLED  
{NAME => 'follows', coprocessor$1 => 'file:///U true users/ndimiduk/repos/hbaseia twitbase/target/twitbase-1.0.0.jar|HBaseIA.TwitBase.coprocessors.FollowsObserver|1001|', FAMILIES => [{NAME => 'f', BLOOMFILTER => 'NONE', REPLICATION\_SCOPE =>'0', VERSIONS => '1', COMPRESSION => 'NONE', MIN\_VERSIONS => '0', TTL => '2147483647', BLOCKSIZE => '65536', IN\_MEMORY => 'false', BLOCKCACHE => 'true'}]}  
1 row(s) in 0.0330 seconds  
  
**Tuning HBase**  
hbase(main):003:0> help 'status'  
  
**SPLITTING TABLES**  
hbase(main):019:0> split 'mytable' , 'G'  
Alter table  
hbase(main):020:0> alter 't', NAME => 'f', VERSIONS => 1  
  
**TRUNCATING TABLES**  
hbase(main):023:0> truncate 't'  
Truncating 't' table (it may take a while):  
- Disabling table...  
- Dropping table...  
- Creating table...  
0 row(s) in 14.3190 seconds

Remember that data is never deleted by HBase until it does a compaction -- where it rewrites all of its data files. Once the data passes it TTL it will be invisible until a major compaction happens.

### As we all know Apache Hbase is an distributed data store optimized for read performance and the optimized read performance come from having only one file per Column Family.Hbase idea comes from Google File system and based on log structures merge tree.

### one file per Colum Family is not always possible during period of heavy writes,that is the reason Hbase try to combine the HFiles to reduce the maximum number of disk seeks needed for a read. This process is called compaction.

### 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 Compactions:**reads allthe Store files for a Region and writes to a singleStore file.

### Each  Hbase table has following

### 1. One or more column families: group columns and specify the physical layout of data storage

### 2. one or more Regions: same as shards that means set of rows belonging to a table specified by startKey and EndKey.

### For every column family of a table in a region we have a store which has

### 1.MemStore – a buffer that holds in-memory modifications (till it is flushed to store files)

### 2.0 or More Store files  – that get created when MemStore fills up.These are called Hfiles.

### These store files are immutable and HBase creates a new file on every MemStore flush i.e. it does not write to an existing HFile.Compaction combines all these Store files for a Region into fewer Store files to optimize performance

### The minor compactions not remove the delete flags and the deleted cells. It only merge the small files into a bigger one. Only the major compaction (in 0.94) will deal with the delete cells. There is also some more compaction mechanism coming in trunk with nice features.

### Minor compactions are promoted to major compactions when the compaction policy decide to compact all the files. If all the files need to be merged, then we can run a major compaction which will do the same thing as the minor one, but with the bonus of deleting the required marked cells.

### One can manually trigger the major compaction using below command

### major\_compact "table name"

### If you let HBase accumulate many HFiles without compacting them, you'll achieve better **write performance** (the data is rewritten less frequently). If on the other hand you instruct HBase to compact many HFiles sooner you'll have better **read performance**, but now the same data is read and rewritten more often.

### HBase allows to tweak when to start compacting HFiles and what is considered the maximum limit of HFiles to ensure acceptable read performance.

### Generally flushes and compaction can commence in parallel. A scenario of particular interest is when clients write to HBase faster than the  can absorb, i.e. faster than compactions can reduce the number of HFiles - manifested in an ever larger number of HFiles, eventually reaching the specified limit. When this happens the memstores can continue to buffer the incoming data, but they cannot grow indefinitely - RAM is limited. What should HBase do in this case? What can it do? The only option is to **disallow writes**, and that is exactly what HBase does.Looks at Hbase-site.xml for many parameters which control this phenomena.

------------------------------------------

NoSql supported by column oriented databases where RDBMS is row oriented database... And say for example we have a Employee table with Name, Age, Salery, Address, EmployeeId etc... we put same table in MySql (RDBMS support) and HBase (NoSQL support). If a customer/client writes a query to get the average Age or Salery details from 1Lakh employees records... what happens?

In RDBMS it will go around each row and collects the value and sum & divide for resulting. When it comes to Columnar database no need to worry about all one lakh row iterations. But deal with only one Row which is faster to compute. So this way sometimes NoSQL is faster than SQL. This case NoSQL don't care about ACID complaints are worth!

-------------------------------------------------

For example take this example, u have a customer model with many orders and many items associated with each order, then they also have many saved items for later purchases... if you are a large ecommerce store with let's say 10 million customers and 50 million orders. And that customer logs into their dashboard which displays this exact data, how much work is a sql database going to need to do to find the customer, join the orders and each line item and saved items. In a sql database all this data will likely need to be joined in some way... or u can create a collection in ur database called usercache and save this data exactly how you use it in real life. So this can truely be a single query on a single field [id] to get all of this data back. On top of that the nosql database doesn't need to send the query to all 40 database servers to retrieve the information just the ones that actually have the data.

So can a sql db query a single Id field just as fast if not faster than nosql? Yes but can a sql database return all of the data you need by querying one table and one field? No, unless you do something like save the data in Json inside a large text field. But now that data is not query-able for potential future use.  
----------------------------------

The question itself is severely flawed in several ways:

*"What makes nosql type databases fast .. compared to relational sql based databases?"*

For ad hoc queries, joins, updates, etc., relational databases will tend to be faster than "nosql type databases" for *most* use cases. The reason that NoSQL is useful is that many applications can be built avoiding those particular use cases, and can instead focus on using a very small set of database functionality; for example, applications can perform all data access and modification using primary key-based operations in order to optimize for a NoSQL K/V store.

*What makes nosql type databases .. scalable compared to relational sql based databases?"*

Contrary to the assumption in your question, *most* of the operations that one can perform on a relational (SQL) database are either impossible or impossibly-slow using a NoSQL database, and tend to get *worse* as the NoSQL database is scaled out. As in the above example, applications can be optimized to avoid these particular use cases and instead focus on a very small set of functionality that does scale extremely well, by relying on features that enable partitioning, replication, and routing. (These terms are from a presentation I've given on "The Five Patterns of Stateful Scale-Out".)

*I understand that they distribute data into separate data files which removes the bottle neck of concurrency*

This is an incorrect statement. Some relational databases use a single file, while others use many files. The same goes for NoSQL databases.  
  
However, I will assume that you are referring to partitioning across servers, which would have the side effect of scaling out I/O across multiple servers, with each server reading and writing only a subset of the partitions. In that case, the solution is likely to scale better with the addition of servers.

*Whats the core difference in the systems that make them scalable to millions of operations per second.*

Just to be clear, relational databases *can* scale to millions of operations per second. I just did a quick Google search and found one of the Oracle Exadata references (US Customs & Border Patrol) claiming 32 billion queries per day on their database (on average 1/2 million per second, but obviously peak times would have much higher rates).  
  
However, the *cost* of scaling NoSQL operations is significantly lower, because (in the general case) NoSQL implementations tend to relax the requirements around Consistency and Durability (the "C" and the "D" from ACID).  
  
For an example regarding Durability (the "D" in ACID), MongoDB was a popular NoSQL database early on because from a developer's perspective it seemed very fast for some typical use cases. Behind the scenes, though, it wasn't actually recording the data to disk as the "transactions" completed, so when web sites built with MongoDB had a server crash, they would lose important data. Even worse, it was completely single-threaded for transactions (a global write lock), so when web sites actually saw production load, it neither scaled nor was fast -- no matter how many servers there were. (FWIW - I think that they got rid of the global write lock in 2.2.)  
  
Regarding Consistency (the "C" in ACID), real-time consistency is particularly difficult to implement in a partitioned system, because transactions that occur across (or draw data from) multiple partitions need to be globally ordered ("serializable" in transaction terminology) in order to ensure that operations are drawing from a consistent view of the system. NoSQL systems that use partitioning tend to solve this by relaxing the Consistency requirement such that it applies at a per-partition level, and not at a system level. This is a very reasonable engineering trade-off for most systems, and removes the greatest barrier for scalability that exists in distributed systems.

----------------------

 if you need to do several Inserts to store an order and you only need one simple Insert in a database like djondb then the performance will be 10x faster in the NoSQL world, just because you're using 10 times less calls to the database layer, that's it.

-----------------------

Here I'm assuming that you want to optimize one particular query, which is simply looking up a record by key. One example of this might be looking up a userinfo record by username. For some systems a query like that has to be incredibly fast and all other queries are unimportant.

The biggest factor in database performance will be the number of I/O operation required to read/write data. Most database systems use similar data structures (i.e. b-trees) which can retieve uncached data in O(log(n)) I/Os. In order to give durable updates the data will have to be written to disk: most systems do that sequentially, which is the fastest way.

So, where can a Key-Value store get efficiencies?

1. Non-normalized data. Putting all the data in one row means no joins.
2. Low CPU overhead. A key-value store avoids the CPU cost of query processing/optimization, security checks, constraint checks, etc.
3. It is easier to have the store be in-process (as opposed to a SQL server running as a separate service) this eliminate IPC overhead.

Most RDBMS systems are built on top of something which looks like a key-value store so you could view this as cutting out the middleman.

**Suppose I have three tables BaseCategory,Category and products. If i am thinking in terms of RDBMS then the relationship amoung these tables are**

**1- One BaseCategory has Many categories 2- One Category has Many Products.**

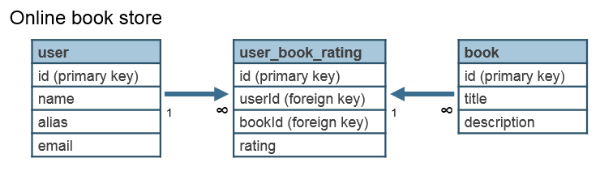
**Now i am thinking to convert it into HBase. can anybody help me how to map these relations into HBase?**

|  |  |
| --- | --- |
| up vote0down vote | You'd probably have each row represent a supercategory/category pair (encoded with a separator, e.g. MySuperCategory:MyCategory, and a column family named "products" with a column for each product in that category.  This would allow you to very quickly retrieve all of the items in a given supercategory/category pair, and with some de-duplication all of the items in a supercategory. |

**Many-to-Many Relationship in an RDBMS**

Here is an example of a many-to-many relationship in a relational database. These are the query requirements:

* Get name for user x
* Get title for book x
* Get books and corresponding ratings for userID x
* Get all userIDs and corresponding ratings for book y



**Many-to-Many Relationship in HBase**

The queries that we are interested in are:

* Get books and corresponding ratings for userID x
* Get all userIDs and corresponding ratings for book y

For an entity table, it is pretty common to have one column family storing all the entity attributes, and column families to store the links to other entities.

The entity tables are as shown below:

