



This is Database Management System which provides a mechanism for [storage](https://en.wikipedia.org/wiki/Computer_data_storage) and [retrieval](https://en.wikipedia.org/wiki/Data_retrieval) of data that is modeled in means other than the tabular relations used in [relational databases](https://en.wikipedia.org/wiki/Relational_database). Such databases have existed since the late 1960s, but did not obtain the "NoSQL" moniker until a surge of popularity in the early twenty-first century,[[2]](https://en.wikipedia.org/wiki/NoSQL#cite_note-leavitt-2) triggered by the needs of [Web 2.0](https://en.wikipedia.org/wiki/Web_2.0) companies such as [Facebook](https://en.wikipedia.org/wiki/Facebook), [Google](https://en.wikipedia.org/wiki/Google), and [Amazon.com](https://en.wikipedia.org/wiki/Amazon.com).[[3]](https://en.wikipedia.org/wiki/NoSQL#cite_note-3)[[4]](https://en.wikipedia.org/wiki/NoSQL#cite_note-4)[[5]](https://en.wikipedia.org/wiki/NoSQL#cite_note-5) NoSQL databases are increasingly used in [big data](https://en.wikipedia.org/wiki/Big_data) and [real-time web](https://en.wikipedia.org/wiki/Real-time_web) applications.[[6]](https://en.wikipedia.org/wiki/NoSQL#cite_note-6) NoSQL systems are also sometimes called "Not only SQL" to emphasize that they may support [SQL](https://en.wikipedia.org/wiki/SQL)-like query languages.[[7]](https://en.wikipedia.org/wiki/NoSQL#cite_note-7)[[8]](https://en.wikipedia.org/wiki/NoSQL#cite_note-8)

Motivations for this approach include: simplicity of design, simpler ["horizontal" scaling](https://en.wikipedia.org/wiki/Horizontal_scaling#Horizontal_and_vertical_scaling) to [clusters](https://en.wikipedia.org/wiki/Cluster_computing) of machines (which is a problem for relational databases),[[2]](https://en.wikipedia.org/wiki/NoSQL#cite_note-leavitt-2)and finer control over availability. The data structures used by NoSQL databases (e.g. key-value, wide column, graph, or document) are different from those used by default in relational databases, making some operations faster in NoSQL. The particular suitability of a given NoSQL database depends on the problem it must solve. Sometimes the data structures used by NoSQL databases are also viewed as "more flexible" than relational database tables.[[9]](https://en.wikipedia.org/wiki/NoSQL#cite_note-9)

Many NoSQL stores compromise [consistency](https://en.wikipedia.org/wiki/Consistency_(database_systems)) (in the sense of the [CAP theorem](https://en.wikipedia.org/wiki/CAP_theorem)) in favor of availability, partition tolerance, and speed. Barriers to the greater adoption of NoSQL stores include the use of low-level query languages (instead of SQL, for instance the lack of ability to perform ad-hoc joins across tables), lack of standardized interfaces, and huge previous investments in existing relational databases.[[10]](https://en.wikipedia.org/wiki/NoSQL#cite_note-10) Most NoSQL stores lack true [ACID](https://en.wikipedia.org/wiki/ACID) transactions, although a few databases, such as [MarkLogic](https://en.wikipedia.org/wiki/MarkLogic), [Aerospike](https://en.wikipedia.org/wiki/Aerospike_database), FairCom [c-treeACE](https://en.wikipedia.org/wiki/C-treeACE), Google [Spanner](https://en.wikipedia.org/wiki/Spanner_(database)) (though technically a [NewSQL](https://en.wikipedia.org/wiki/NewSQL) database), Symas [LMDB](https://en.wikipedia.org/wiki/Lightning_Memory-Mapped_Database), and [OrientDB](https://en.wikipedia.org/wiki/OrientDB)have made them central to their designs. (See [ACID and join support](https://en.wikipedia.org/wiki/NoSQL#ACID_and_join_support).)

Instead, most NoSQL databases offer a concept of "eventual consistency" in which database changes are propagated to all nodes "eventually" (typically within milliseconds) so queries for data might not return updated data immediately or might result in reading data that is not accurate, a problem known as stale reads.[[11]](https://en.wikipedia.org/wiki/NoSQL#cite_note-11) Additionally, some NoSQL systems may exhibit lost writes and other forms of [data loss](https://en.wikipedia.org/wiki/Data_loss).[[12]](https://en.wikipedia.org/wiki/NoSQL#cite_note-12) Fortunately, some NoSQL systems provide concepts such as [write-ahead logging](https://en.wikipedia.org/wiki/Write-ahead_logging) to avoid data loss.[[13]](https://en.wikipedia.org/wiki/NoSQL#cite_note-13) For [distributed transaction processing](https://en.wikipedia.org/wiki/Distributed_transaction_processing) across multiple databases, data consistency is an even bigger challenge that is difficult for both NoSQL and relational databases. Even current relational databases "do not allow referential integrity constraints to span databases."[[14]](https://en.wikipedia.org/wiki/NoSQL#cite_note-14)There are few systems that maintain both [ACID](https://en.wikipedia.org/wiki/ACID) transactions and [X/Open XA](https://en.wikipedia.org/wiki/X/Open_XA) standards for distributed transaction processing.



There are various NoSQL Databases. Each one uses a different method to store data. Some might use column store, some document, some graph, etc., Each database has its own unique characteristics.

**In the in-memory databases** like Redis/CouchBase/Tarantool/Aerospike everything is stored in RAM in balanced trees like RB-Tree or in hash tables. All the writes are applied on both RAM and disk, but on disk it goes in an append-only way. A file append can be done as fast as 100 Mbytes per second on a normal magnetic disk. If a record size is, say, 1K, then the data will be written at 100 krps.

**In the on-disk NoSQL databases and db-engines** like Cassandra/HBase/RocksDB/LevelDB/Sophia the main idea is that you have a snapshot file and a write ahead log (WAL) file. Snapshot contains already prepared data in a form of B-Tree with upper levels of that tree being permanently in RAM, that can be accesses for reading by doing only one disk seek. A WAL contains all the new changes on top of a current snapshot. A snapshot file is being totally rebuilt on a regular basis using current snapshot and a WAL. All the writes are done nearly as fast as with in-memory databases. "Nearly" because disk is partially busy by doing regular snapshot converting that was described earlier. Reads are significantly slower than that are in in-memory databases, because they take at least one disk seek, but good news is that they can be cached in optimized in-memory structures like RB-Trees/hash tables.

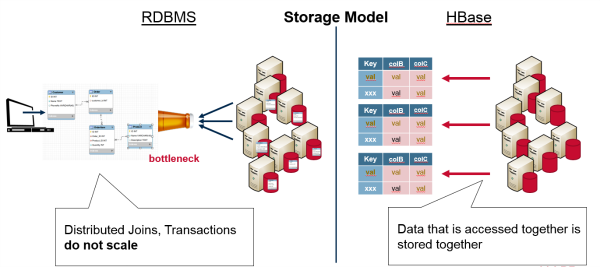
That's the way it works in a rough approximation.

If we talking about precisely HBase database then the short answer will be :

HBase is a columnar database, so all data is stored into tables with rows and columns similar to relational database management systems (RDBMSs). The intersection of a row and a column is called a cell. One important difference between HBase tables and RDBMS tables is versioning.

Each cell value includes a “version” attribute, which is nothing more than a timestamp uniquely identifying the cell. Versioning tracks changes in the cell and makes it possible to retrieve any version of the contents should it become necessary. HBase stores the data in cells in decreasing order (using the timestamp), so a read will always find the most recent values first.

Columns in HBase belong to a column family. The column family name is used as a prefix to identify members of its family. For example, fruits:apple and fruits:banana are members of the fruits column family. HBase implementations are tuned at the column family level, so it is important to be mindful of how you are going to access the data and how big you expect the columns to be.

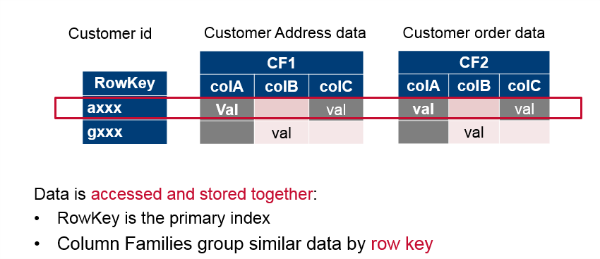


The rows in HBase tables also have a key associated with them. The structure of the key is very flexible. It can be a computed value, a string, or even another data structure. The key is used to control access to the cells in the row, and they are stored in order from low value to high value.

All of these features together make up the schema. The schema is defined and created before any data can be stored. Even so, tables can be altered and new column families can be added after the database is up and running. This extensibility is extremely useful when dealing with big data because you don’t always know about the variety of your data streams.

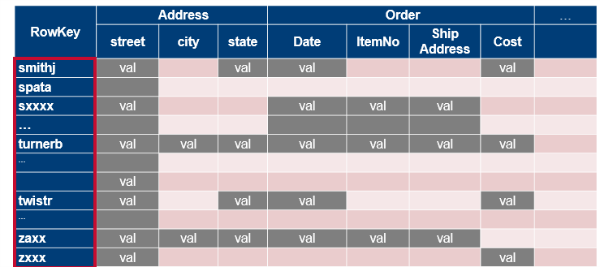


A column family is a NoSQL object that contains columns of related data. It is a tuple (pair) that consists of a key-value pair, where the key is mapped to a value that is a set of columns. In analogy with relational databases, a column family is as a "table", each key-value pair being a "row".

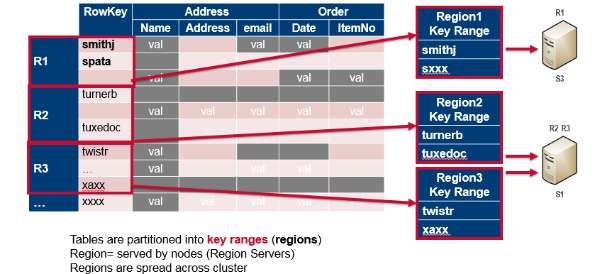


Columns in Apache HBase are grouped into column families. All column members of a column family have the same prefix. For example, the columns courses:history and courses:math are both members of the courses column family. The colon character (:) delimits the column family from the . The column family prefix must be composed of printable characters. The qualifying tail, the column family qualifier, can be made of any arbitrary bytes. Column families must be declared up front at schema definition time whereas columns do not need to be defined at schema time but can be conjured on the fly while the table is up an running.

Physically, all column family members are stored together on the filesystem. Because tunings and storage specifications are done at the column family level, it is advised that all column family members have the same general access pattern and size characteristics.



Tables are divided into sequences of rows, by key range, called regions. These regions are then assigned to the data nodes in the cluster called “RegionServers.” This scales read and write capacity by spreading regions across the cluster. This is done automatically and is how HBase was designed for horizontal sharding.





There is no special limit of number of columns per row declared. However, the potential issue could be with 'too wide' rows. If you don't specify exact qualifiers any scan will result whole rows so you could get much more data than you actually need. Think about ranges like 'this month'. Again, do you really want to use slow intra-row scanning to get needed column inside row?



Basically, it isn’t necessary at all because Columns are essentially just an additional label for the value ( arbitrary names (or labels) assigned by the application ) . The principle operations supported by HBase are [Put](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html#put%28org.apache.hadoop.hbase.client.Put%29) (add some data), [Delete](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html#delete%28org.apache.hadoop.hbase.client.Delete%29) ("delete" some data), [Scan](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html#getScanner%28org.apache.hadoop.hbase.client.Scan%29) (retrieve some cells), [Get](http://hbase.apache.org/apidocs/org/apache/hadoop/hbase/client/HTable.html#get%28org.apache.hadoop.hbase.client.Get%29) (which is just a special case of Scan). No queries, no secondary indexes.

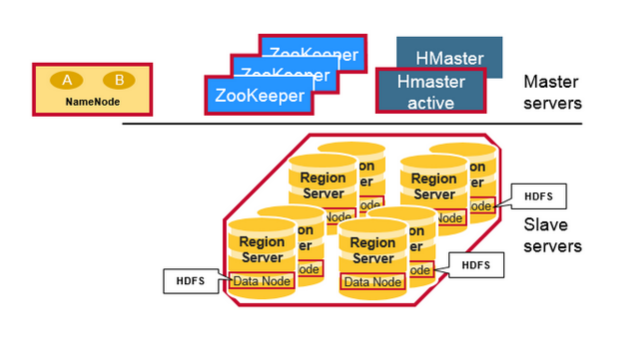


To answer on this question in the best way we should first of all describe the architecture of components of HBase.

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.

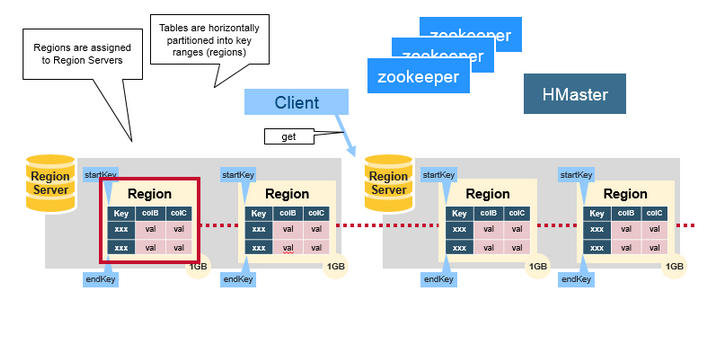
The Hadoop DataNode stores the data that the Region Server is managing. All HBase data is stored in HDFS files. Region Servers are collocated with the HDFS DataNodes, which enable data locality (putting the data close to where it is needed) for the data served by the RegionServers. HBase data is local 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.

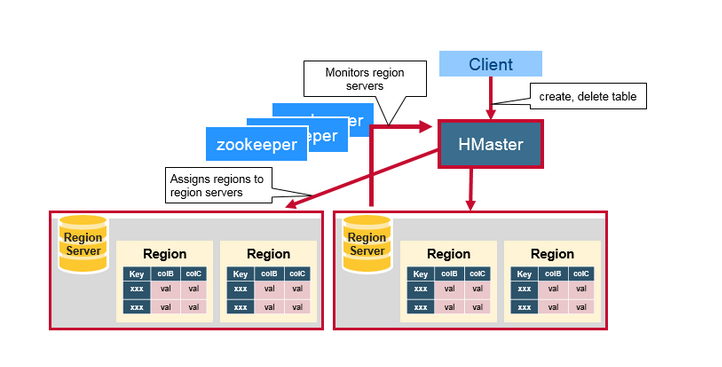


## **HBase HMaster**

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

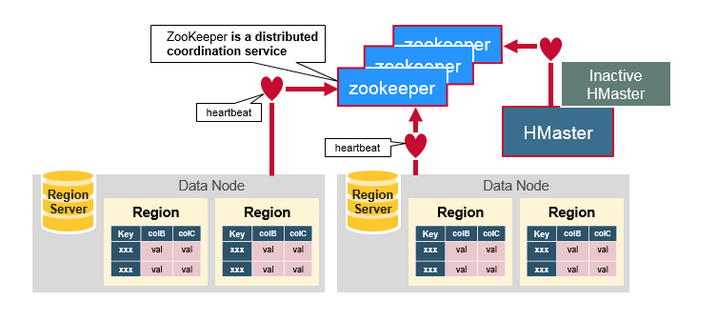
A master is responsible for:

* 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, updating tables



## **ZooKeeper: The Coordinator**

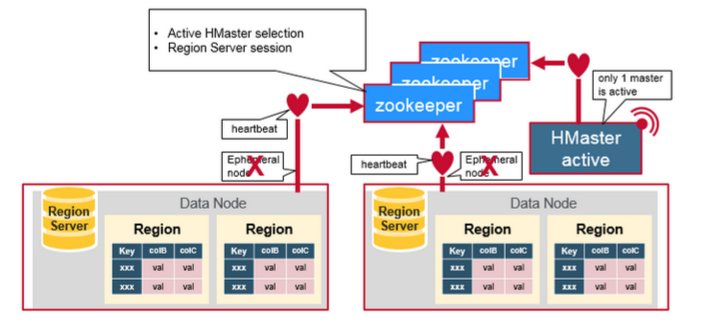
HBase uses ZooKeeper as a distributed coordination service to maintain server state in the cluster. Zookeeper maintains which servers are alive and available, and provides server failure notification. Zookeeper uses consensus to guarantee common shared state. Note that there should be three or five machines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



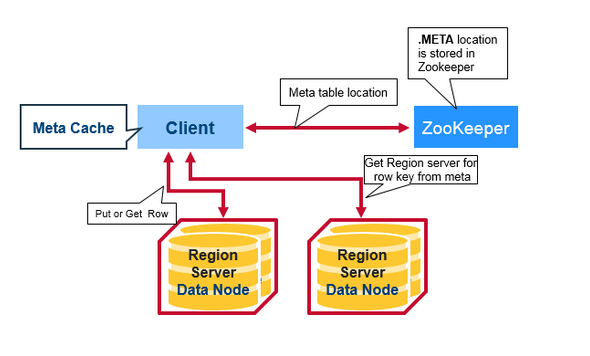
First of all we should start with meta table

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:

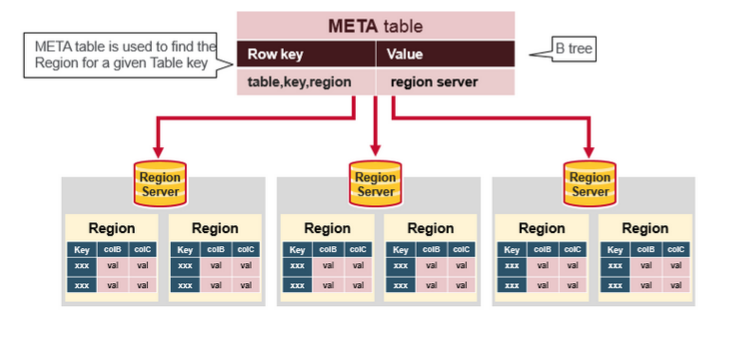
1. The client gets the Region server that hosts the META table from ZooKeeper.
2. The client will query the .META. server to get the region server corresponding to the row key it wants to access. The client caches this information along with the META table location.
3. It will get the Row from the corresponding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



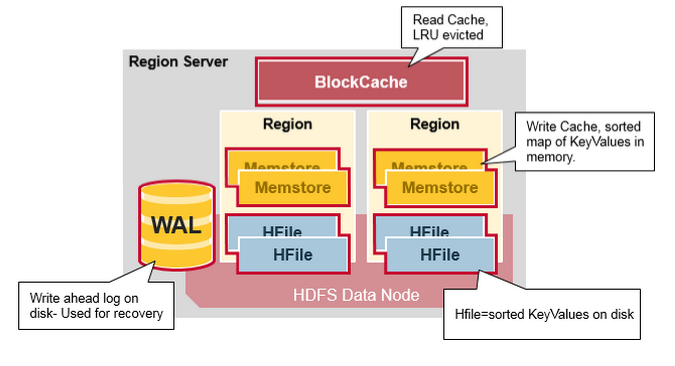
**HBase Meta Table**

* 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.

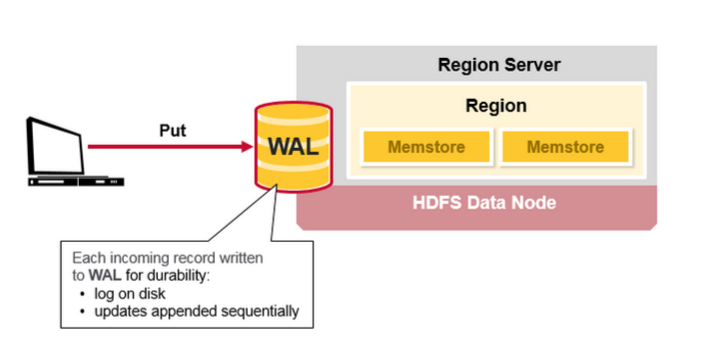


**HBase Write Steps (1)**

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.

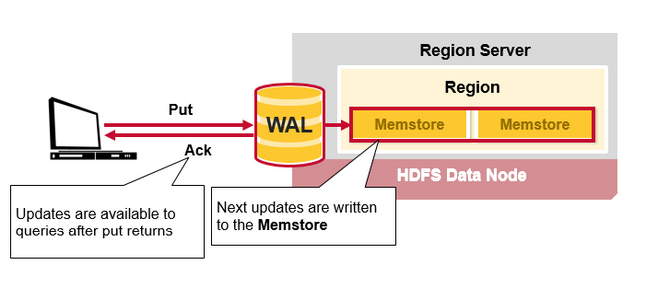
- The WAL is used to recover not-yet-persisted data in case a server crashes.



## 

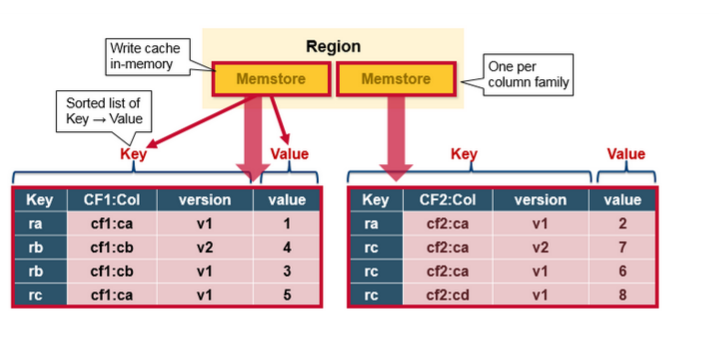
## **HBase Write Steps (2)**

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.



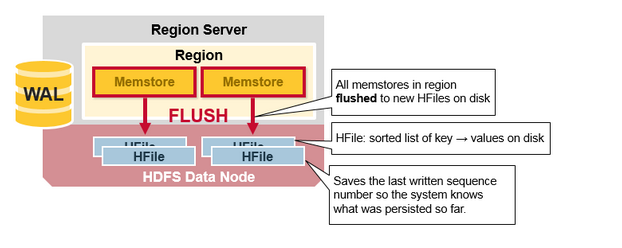
## 

## **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.

