Source :

1-<https://mapr.com/blog/in-depth-look-hbase-architecture/>

2- <https://www.edureka.co/blog/hbase-architecture/>

HBase vs Cassandra :

|  |  |  |
| --- | --- | --- |
|  | HBase | Cassandra |
| Single point of failure | HBase has several components which communicate together like HBase HMaster, ZooKeeper, NameNode, Region Severs. Can have single point of failure. | Cassandra is a single node type, in which all nodes are equal and performs all functions. Any node can be the coordinator; this removes Single Point of failure. |
| consistency | HBase is optimized for read and supports **single writes**, which leads to strict consistency. | Cassandra supports single row reads which maintains eventual consistency. |
| Fast Scan | HBase supports Range based scans, which makes scanning process faster. Due to RowKey partitioning the scanning process is faster in HBase as compared to Cassandra. | Cassandra does not support range based row scans, which slows the scanning process as compared to HBase. |
| Partioning | HBase supports ordered partitioning, in which rows of a Column Family are stored in RowKey order | whereas in Casandra ordered partitioning is a challenge |
| Read load balancing | HBase does not support read load balancing, one Region Server serves the read request and the replicas are only used in case of failure. | While Cassandra supports read load balancing and can read the same data from various nodes. This can compromise the **consistency**. |
| CAP (Consistency, Availability & Partition -Tolerance) | HBase maintains Consistency and Availability | Cassandra focuses on Availability and Partition -Tolerance. |

**Features of HBase**

* **Atomic read and write:**On a row level, HBase provides atomic read and write. It can be explained as, during one read or write process, all other processes are prevented from performing any read or write operations.
* **Consistent reads and writes:**HBase provides consistent reads and writes due to above feature.
* **Linear and modular scalability:** As data sets are distributed over HDFS, thus it is linearly scalable across various nodes, as well as modularly scalable, as it is divided across various nodes.
* **Automatic and configurable sharding of tables:**HBase tables are distributed across clusters and these clusters are distributed across regions. These regions and clusters split, and are redistributed as the data grows.
* **Easy to use Java API for client access:** It provides easy to use Java API for programmatic access.
* **Thrift gateway and a REST-ful Web services:** It also supports Thrift and REST API for non-Java front-ends.
* **Block Cache and Bloom Filters:**HBase supports a Block Cache and Bloom Filters for high volume query optimization .
* **Automatic failure support:**HBase with HDFS provides WAL (Write Ahead Log) across clusters which provides automatic failure support.
* **Sorted rowkeys:** As searching is done on range of rows, HBase stores rowkeys in a lexicographical order. Using these sorted rowkeys and timestamp, we can build an optimized request.

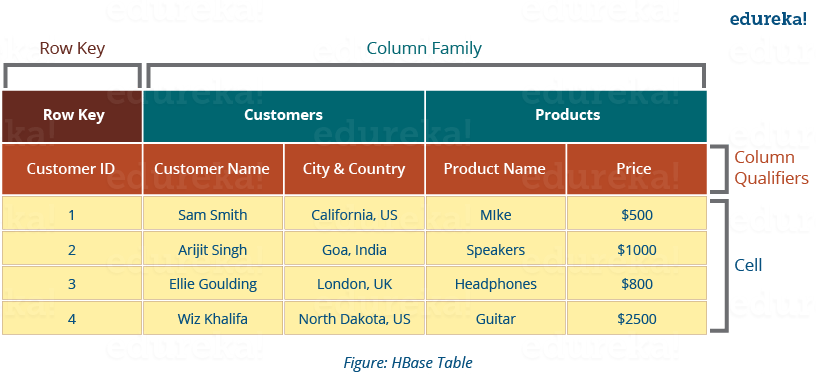
## ****Where we can use HBase?****

* We should use HBase where we have large data sets (millions or billions or rows and columns) and we require fast, random and real time, read and write access over the data.
* The data sets are distributed across various clusters and we need high scalability to handle data.
* The data is gathered from various data sources and it is either semi structured or unstructured data or a combination of all. It could be handled easily with HBase.
* You want to store column oriented data.
* You have lots of versions of the data sets and you need to store all of them.

**Tall-Narrow and Flat-Wide** are the two HBase table design approaches that can be used. However, which approach should be used when merely depends on what you want to achieve and how you want to use the data. The performance of HBase completely depends on the RowKey and hence on directly on how data is accessed.  
  
On a high level, the major difference between flat-wide and tall-narrow approach is similar to the difference between get and scan. Full scans are costly in HBase because of ordered RowKey storage policy. Tall-narrow approach can be used when there is a complex RowKey so that focused scans can be performed on a logical group of entries.  
  
Ideally, the tall-narrow approach is used when there are less number of rows and a large number of columns whereas flat-wide approach is used when there are less number of columns and a large number of rows.

## ****HBase Data Model****

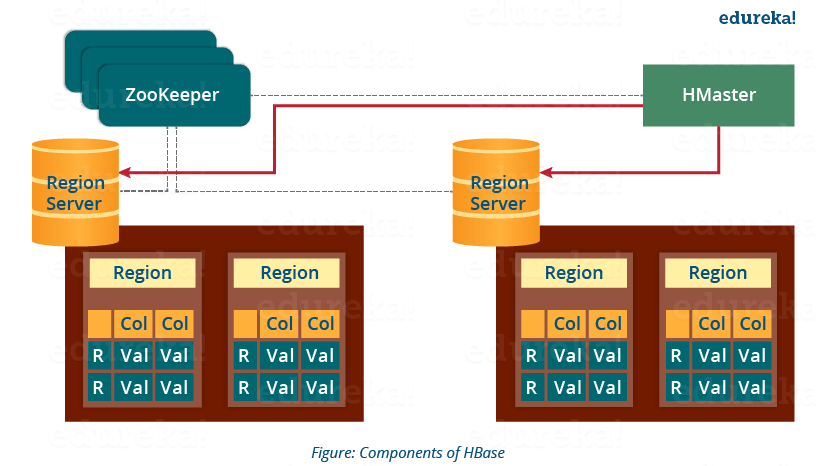
## ****Column oriented****



* **Tables**: Data is stored in a table format in HBase. But here tables are in column-oriented format.
* **Row** **Key**: Row keys are used to search records which make searches fast. You would be curious to know how? I will explain it in the architecture part moving ahead in this blog.
* **Column** **Families**: Various columns are combined in a column family. These column families are stored together which makes the searching process faster because data belonging to same column family can be accessed together in a single seek.
* **Column** **Qualifiers**: Each column’s name is known as its column qualifier.
* **Cell**: Data is stored in cells. The data is dumped into cells which are specifically identified by rowkey and column qualifiers.
* **Timestamp**: Timestamp is a combination of date and time. Whenever data is stored, it is stored with its timestamp. This makes easy to search for a particular version of data.

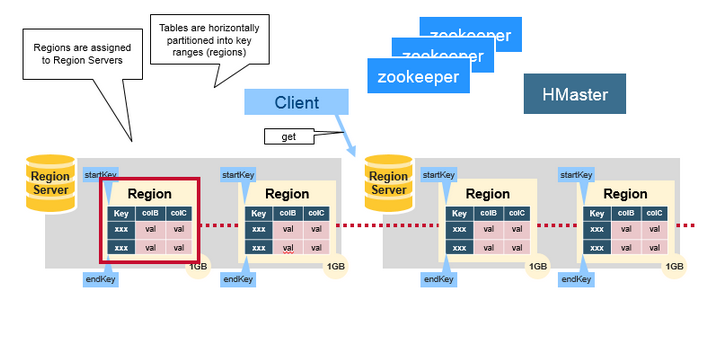
## ****HBase Architecture: Components of HBase Architecture****

HBase has three major components i.e., **HMaster Server**, **HBase Region Server, Regions** and **Zookeeper**.



## ****Region****

A **region** contains all the rows between the start key and the end key assigned to that region. HBase tables can be divided into a number of regions in such a way that all the columns of a column family is stored in one region. Each region contains the rows in a sorted order.



Many regions are assigned to a **Region Server(up to 1000 region per region server)**, which is responsible for handling, managing, executing reads and writes operations on that set of regions.

* A table can be divided into a number of regions. A Region is a sorted range of rows storing data between a start key and an end key.
* A Region has a default size of 256MB which can be configured according to the need.
* A Group of regions is served to the clients by a Region Server.
* A Region Server can serve approximately 1000 regions to the client.

## ****HMaster****

It is similar to NameNode in HDFS, it handles multiple region servers. HBase HMaster performs DDL operations (create and delete tables).

HMaster is responsible for:

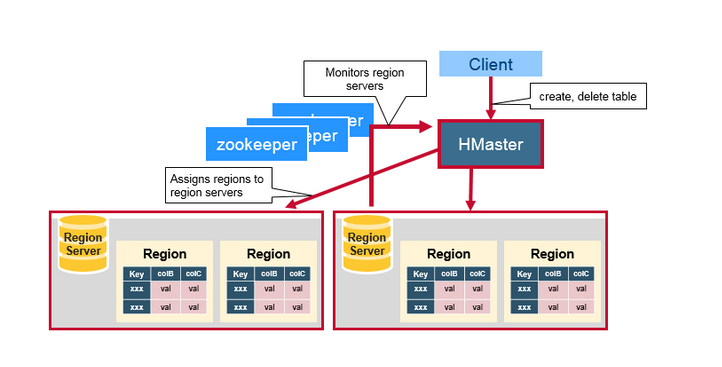
* Coordinating the region servers

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

- Monitoring all RegionServer instances in the cluster (listens for notifications from zookeeper) and performs recovery activities whenever any Region Server is down.

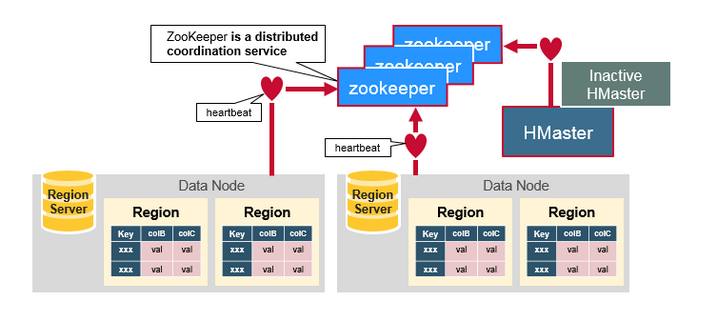
* Admin functions

- 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.

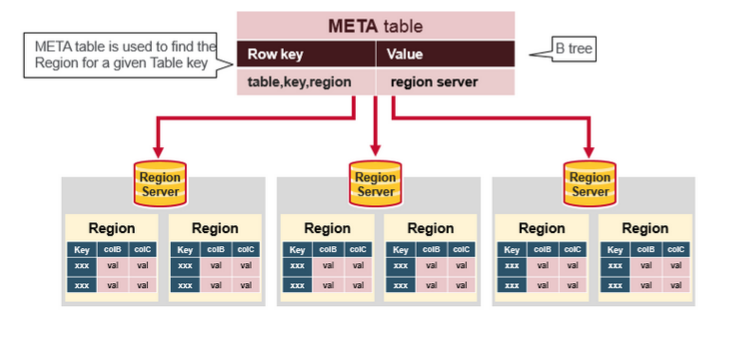


* It helps in maintaining server state inside the cluster by communicating through sessions.
* Every Region Server along with HMaster Server sends continuous heartbeat at regular interval to Zookeeper and it checks which server is alive and available. It also provides server failure notifications so that, recovery measures can be executed.
* There is an inactive server, which acts as a backup for active server. If the active server fails, it comes for the rescue.
* The active HMaster sends heartbeats to the Zookeeper while the inactive HMaster listens for the notification send by active HMaster. If the active HMaster fails to send a heartbeat the session is deleted and the inactive HMaster becomes active.
* If a Region Server fails to send a heartbeat, the session is expired and all listeners are notified about it. Then HMaster performs suitable recovery actions.
* Zookeeper also maintains the .META Server’s path, which helps any client in searching for any region. The Client first has to check with .META Server in which Region Server a region belongs, and it gets the path of that Region Server.

Each Region Server creates an ephemeral (temporary) node. The HMaster monitors these nodes to discover available region servers, and it also monitors these nodes for server failures. HMasters vie (compete eagerly with someone in order to do/achieve something) 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.

## ****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

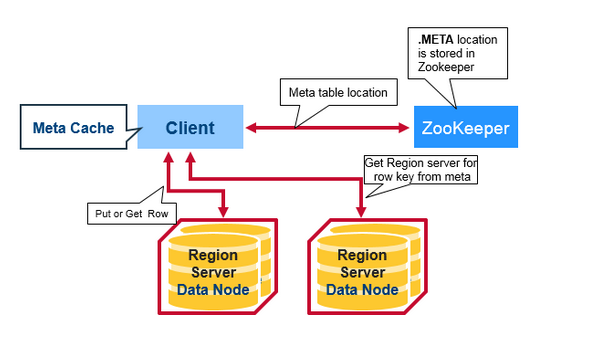
## HBase First Read or Write

ZooKeeper stores the location of the META table.

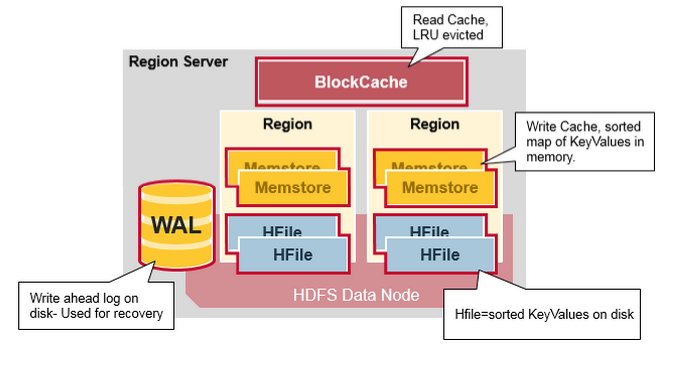
This is what happens the first time a client reads or writes to HBase:

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 Architecture: Components of Region Server****



A Region Server maintains various regions running on the top of [***HDFS***](https://www.edureka.co/blog/apache-hadoop-hdfs-architecture/). Components of a Region Server are:

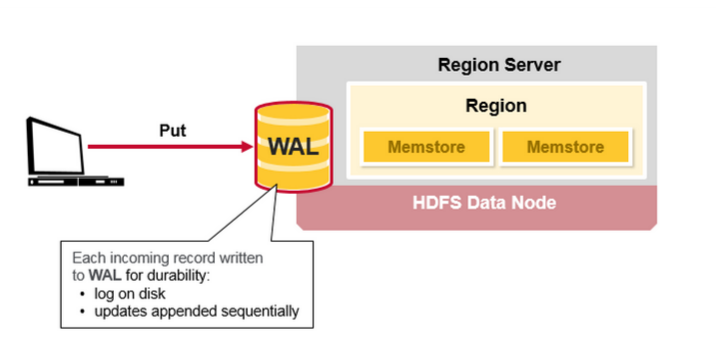
* **WAL (WriteAheadLog):** Write Ahead Log (WAL) is a file attached to every Region Server inside the distributed environment. The WAL stores the new data that hasn’t been persisted or committed to the permanent storage. It is used in case of failure to recover the data sets.
* **Block Cache:**is the 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 (lexicographically) before writing to disk. There is one MemStore per column family per region.
* **HFile:**it stores the rows as sorted KeyValues on disk. MemStore commits the data to HFile when the size of MemStore exceeds.

**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:

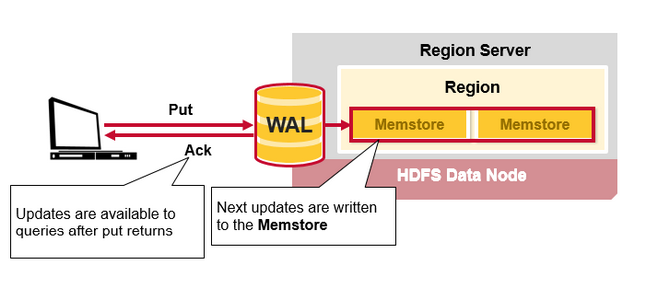
- Edits are appended to the end of the WAL file that is stored on disk.

- 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.

