NoSQL & HBASE

NoSQL means Not Only SQL, implying that when designing a software solution or product, there are more than one storage mechanism that could be used based on the needs NoSQL is a whole new way of thinking about a database. Though NoSQL is not a relational database, the reality is that a relational database model may not be the best solution for all situations. Next Generation Databases mostly addressing some of the points: *being non-relational, distributed, open-source and horizontally scalable*.

a. Problems with RDBMS for **high volume** systems
   Performance problems with huge data set JOINs, Transactions, Normalization, handling different data formats from structured to un structured, huge data loads etc.
b. Solutions that partially solves issue
   Adding in memory cache, splitting read/write between different servers [1 master takes all writes and slaves cater to reads], Vertical scaling up, Sharding

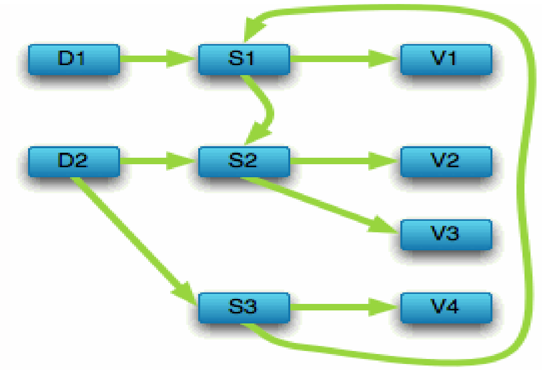
Types of NoSQL

- **Document databases** are a type of NoSQL database that store data in 'documents.' Each bit of data, such as a user record or commerce product, and stored in an individual document. These documents come in multiple formats ranging from HTML, XML and much more commonly JSON. Document databases often run in a Key-Value pattern, in which each document's ID is the key; and the document body is the value. Querying document databases is often extremely time-efficient depending on how the database is architected. Many support a Map/Reduce ideology and use simple JavaScript to query data. Some even have fully HTTP/REST APIs..

```
        {officeName:"3Pillar Noida",
        {Street: "B-25, City:"Noida", State:"UP", Pincode:"201301"}
        }
        {officeName:"3Pillar Timisoara",
        {Boulevard:"Coriolan Brediceanu No. 10", Block:"B, Ist Floor", City: "Timisoara",
Pincode: 300011"}
        }
```

- **Graph stores** allow you to store entities and relationships between these entities. Entities are also known as nodes, which have properties. Think of a node as an instance of an object in the application. Relations are known as edges that can have properties. Edges have directional significance; nodes are organized by relationships which allow you to find interesting patterns between the nodes. The organization of the graph lets the data to be stored once and then interpreted in different ways based on relationships.

  Usually, when we store a graph-like structure in RDBMS, it's for a single type of relationship ("who is my manager" is a common example). Adding another relationship to the mix usually means a lot of schema changes and data movement, which is not the case when we are using graph databases. Similarly, in relational databases we model the graph beforehand based on the Traversal we want; if the Traversal changes, the data will have to change.

  In graph databases, traversing the joins or relationships is very fast. The relationship between nodes is not calculated at query time but is actually persisted as a relationship. Traversing persisted relationships is faster than calculating them for every query. Nodes can have different types of relationships between them, allowing you to both represent relationships between the domain entities and to have secondary relationships for things like category, path, time-trees, quad-trees for spatial indexing, or linked lists for sorted access. Since there is no limit to the number and kind of relationships a node can have, they all can be represented in the same graph database.
- **Key-value stores** are the simplest NoSQL databases. The client can either get the value for the key, put a value for a key, or delete a key from the data store. The value is a blob that the data store just stores, without caring or knowing what's inside; it's the responsibility of the application to understand what was stored. Since key-value stores always use primary-key access, they generally have great performance and can be easily scaled. Examples of key-value stores are Riak, Voldemort, Redis.
- **Column family stores** Column-family databases store data in column families as rows that have many columns associated with a row key. Column families are groups of related data that is often accessed together. For a Customer, we would often access their Profile information at the same time, but not their Orders. Each column family can be compared to a container of rows in an RDBMS table where the key identifies the row and the row consists of multiple columns. The difference is that various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows. When a column consists of a map of columns, then we have a super column. A super column consists of a name and a value

which is a map of columns. Think of a super column as a container of columns. DBs like Cassandra and HBase are optimized for queries over large datasets, and store columns of data together, instead of rows

Characteristics of NoSQL

- **Dynamic Schemas – No Relational model**
- **Auto Sharding**
- **Integrated Caching**
- **Performance, Flexible, No joins, low cost.**

Dynamic Schemas – No Relational model : A flexible data structure means that there is no need to define a structure as a database schema. Traditional RDBMSs require pre-defined schemas, and redefining them carries a high cost. NoSQL, on the other hand, does not require defined schemas, so users can store data with various different structures in the same database table. However, most NoSQL databases do not support high-level query languages such as SQL, which is used by RDBMSs, so products that support either simple relational operations or indexing have been released. This feature is evaluated qualitatively.

Replication is the copying of data to achieve data redundancy and load distribution. Even if data consistency has been lost among the replicas, it is eventually achieved: this is known as eventual consistency. Replication is evaluated in terms of consistency and availability.

Scalable refers to achieving high performance by using many general-purpose machines in a distributed manner. Distributing the data over a large number of machines enables scaling of the data set and distribution of the processing load. A common feature of many NoSQL databases is that data is automatically distributed to new machines when they are added to the cluster, so the performance is also improved. Scale-out is evaluated in terms of scalability and elasticity.

The term sharding describes the logical separation of records into horizontal partitions. The idea is to spread data across multiple storage files—or servers—as opposed to having each stored contiguously. The separation of values into those partitions is performed on fixed boundaries: you have to set fixed rules ahead of time to route values to their appropriate store. With it comes the inherent difficulty of having to reshard the data when one of the horizontal partitions exceeds its capacity.

CAP theorem

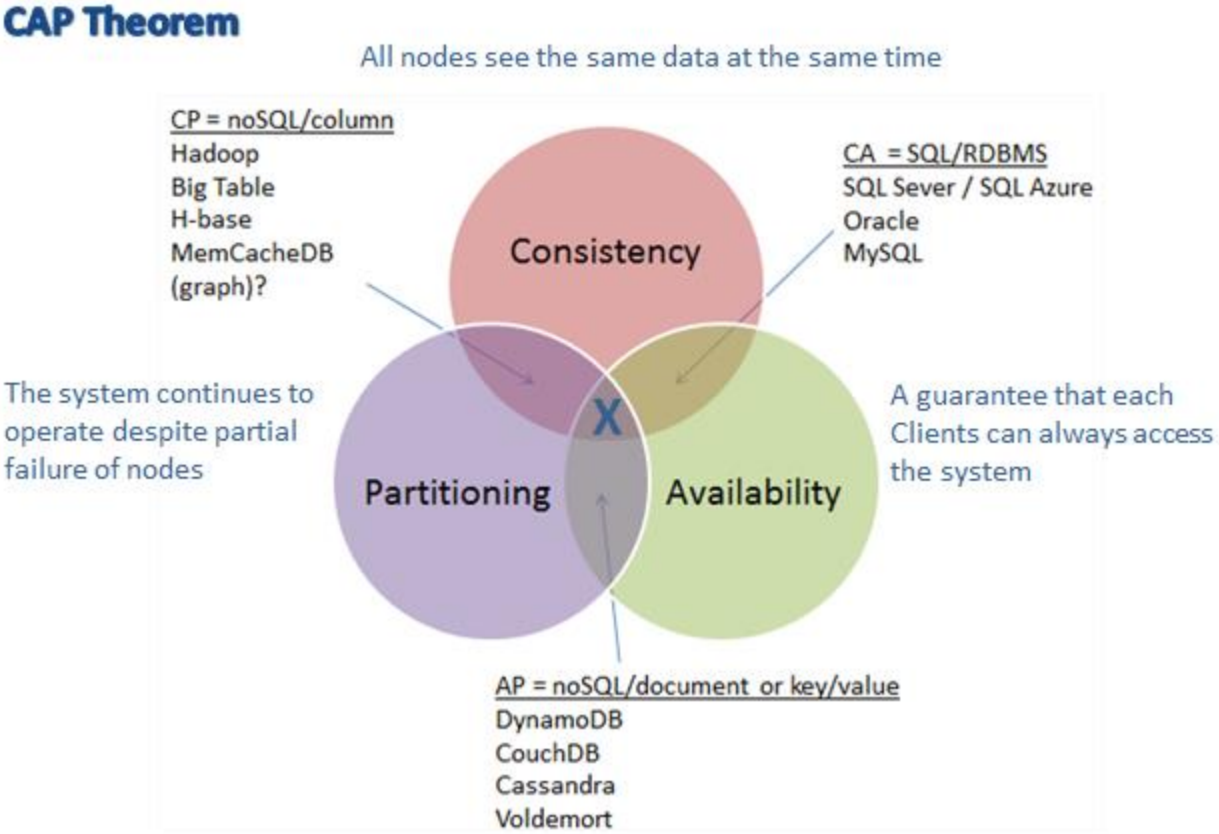
In a distributed system, managing consistency(C), availability(A) and partition toleration(P) is important, Eric Brewer put forth the CAP theorem which states that in any distributed system we can choose only two of consistency, availability or partition tolerance. Many NoSQL databases try to provide options where the developer has choices where they can tune the database as per their needs.

Consistency: all nodes see the same data at the same time

Availability: a guarantee that every request receives a response about whether it was successful or failed

Partition tolerance: the system continues to operate despite arbitrary message loss

The CAP theorem states that if you get a network partition, you have to trade off availability of data versus consistency of data. Durability can also be traded off against latency, particularly if you want to survive failures with replicated data.


CAP Theorem
All nodes see the same data at the same time
CP = noSQL/column
Hadoop
Big Table
H-base
MemCacheDB
(graph)?
CA = SQL/RDBMS
SQL Sever / SQL Azure
Oracle
MySQL
Consistency
X
The system continues to operate despite partial failure of nodes
Partitioning
Availability
A guarantee that each Clients can always access the system
AP = noSQL/document or key/value
DynamoDB
CouchDB
Cassandra
Voldemort

HBASE

What is HBase

HBase is a columnar data store provides random, realtime read/write access to your Big Data which is otherwise called as hadoop database that stores data in hdfs in the form of hfiles and uses java api's to process or compute data sets. Unlike relational database systems, HBase does not support a structured query language like SQL; in fact, HBase isn't a relational data store at all.

Need of HBASE

- **OLAP need with random read/write**
- **Schema less**
- **I/O efficient**
- **Aggregation and compression**

Brief History

- 2006: BigTable paper published by Google. End of year HBase development started.
- 2008: HBase becomes Hadoop sub-project.
- 2010: HBase becomes Apache top-level project.

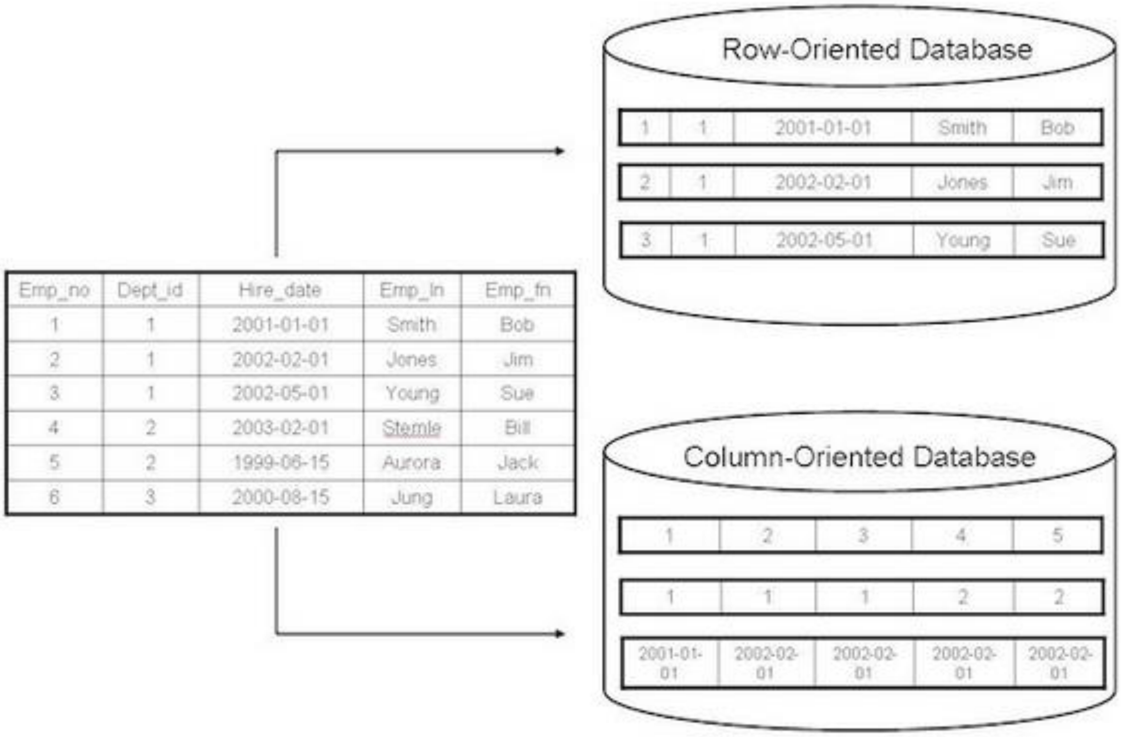
Traditional vs Column oriented storage

Row-oriented data stores –

- Data is stored and retrieved one row at a time and hence could read all row data if only some of the data in a row is required.
- Easy to read and write records
- Well suited for OLTP systems
- Not efficient in performing operations applicable to the entire dataset and hence aggregation is an expensive operation
- Typical compression mechanisms provide less effective results than those on column-oriented data stores

Column-oriented data stores –

- Data is stored and retrieved in columns and hence can read only relevant data if only some data is required
- Read and Write are typically slower operations
- Well suited for OLAP systems
- Can efficiently perform operations applicable to the entire dataset and hence enables aggregation over many rows and columns
- Permits high compression rates due to few distinct values in columns



Emp_no	Dept_id	Hire_date	Emp_ln	Emp_fn
1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue
4	2	2003-02-01	Stemle	Bill
5	2	1999-06-15	Aurora	Jack
6	3	2000-08-15	Jung	Laura

Row-Oriented Database

1	1	2001-01-01	Smith	Bob
2	1	2002-02-01	Jones	Jim
3	1	2002-05-01	Young	Sue

Column-Oriented Database

1	2	3	4	5
1	1	1	2	2
2001-01-01	2002-02-01	2002-02-01	2002-02-01	2002-02-01

Relational vs. HBase

Relational Database

- Is Based on a Fixed Schema
- Is a Row-oriented datastore
- Is designed to store Normalized Data
- Contains thin tables
- Has no built-in support for partitioning.

HBase

- Is Schema-less
- Is a Column-oriented datastore
- Is designed to store Denormalized Data
- Contains wide and sparsely populated tables
- Supports Automatic Partitioning

HDFS vs. HBase

HDFS is a distributed file system that is well suited for storing large files. It's designed to support batch processing of data but doesn't provide fast individual record lookups. HBase is built on top of HDFS and is designed to provide access to single rows of data in large tables. Overall, the differences between HDFS and HBase are

HDFS –

- Is suited for High Latency operations batch processing
- Data is primarily accessed through MapReduce
- Is designed for batch processing and hence doesn't have a concept of random reads/writes

HBase –

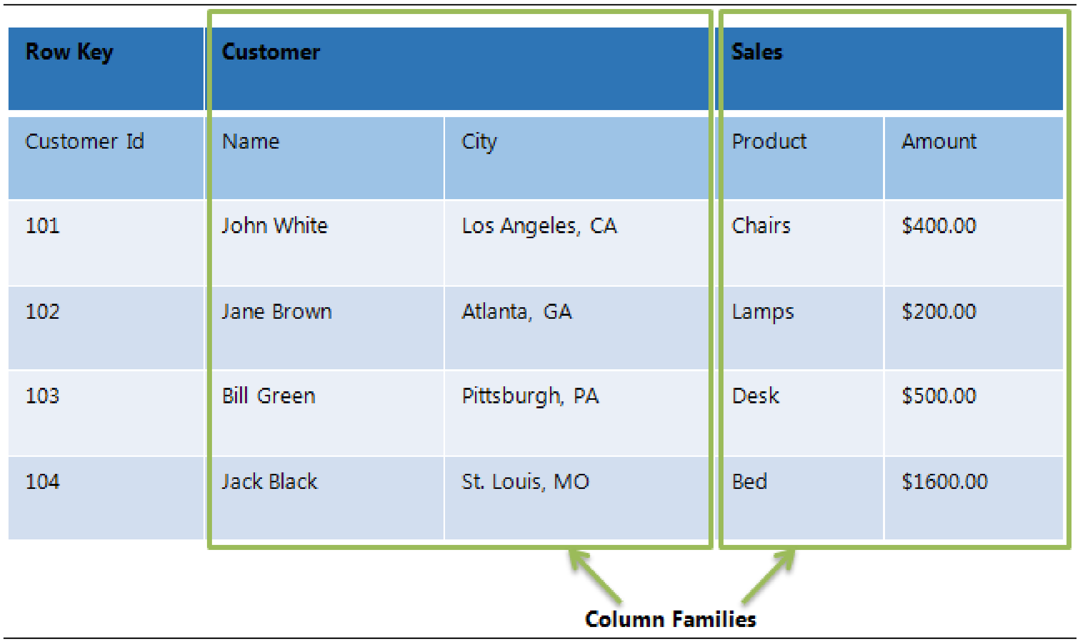
- Is built for Low Latency operations
- Provides access to single rows from billions of records
- Data is accessed through shell commands, Client APIs in Java, REST, Avro or Thrift

HBASE Storage Hierarchy

Most basic unit is a column. One or more columns form a row that is addressed uniquely by a row key. A number of rows, in turn, form a table, and there can be many of them. Each column may have multiple versions, with each distinct value contained in a separate cell. This sounds like a reasonable description for a typical database, but with the extra dimension of allowing multiple versions of each cells.

- **Table -** Collection of rows.
- **Row -** Collection of column families**.**

- **Row Key -** Rows identified by unique ID.
- **Column family -** Collection of columns.
- **Column -** Collection of key value pairs.
- **Cell –** Each value of the column.
- **Timestamp –** Versions of a cell.



Row Key	Customer		Sales	
Customer Id	Name	City	Product	Amount
101	John White	Los Angeles, CA	Chairs	$400.00
102	Jane Brown	Atlanta, GA	Lamps	$200.00
103	Bill Green	Pittsburgh, PA	Desk	$500.00
104	Jack Black	St. Louis, MO	Bed	$1600.00

Column Families

Characteristics

- Isolation – Row level
- Lexographic Sorting (1,10,11,2,20,21)
- Versioning
- Sharding

Table Design

- What is the Row key (Mailbox:UserId:Year:Month)
- How many CF, Columns, Data for CF
- Column names, Cell data
- How many Versions

Row level atomicity

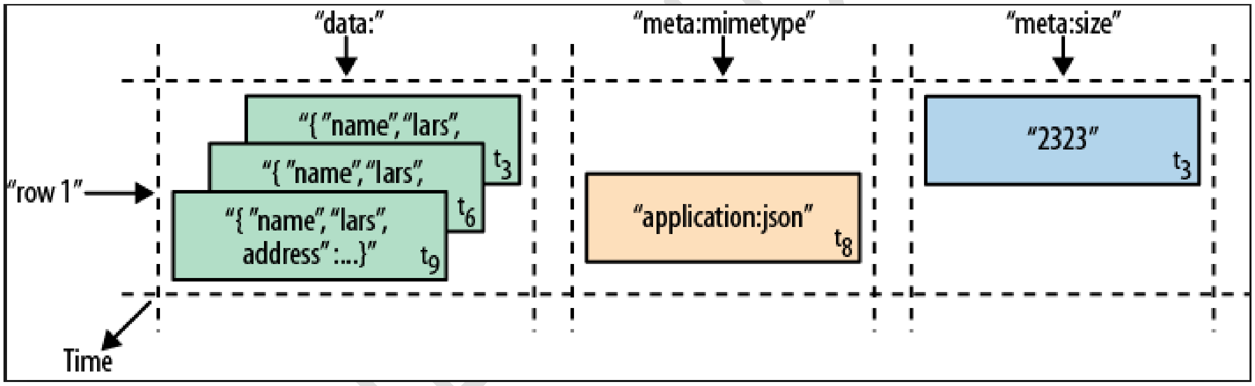
Atomicity is guaranteed by the client taking out exclusive locks on the entire row. When the client goes away during the locked phase the server has to rely on lease recovery mechanisms ensuring that these rows are eventually unlocked again

Versioning

A special feature of HBase is the possibility to store multiple versions of each cell (the value of a particular column). This is achieved by using timestamps for each of the versions and storing them in descending order. Each timestamp is a long integer value measured in milliseconds.

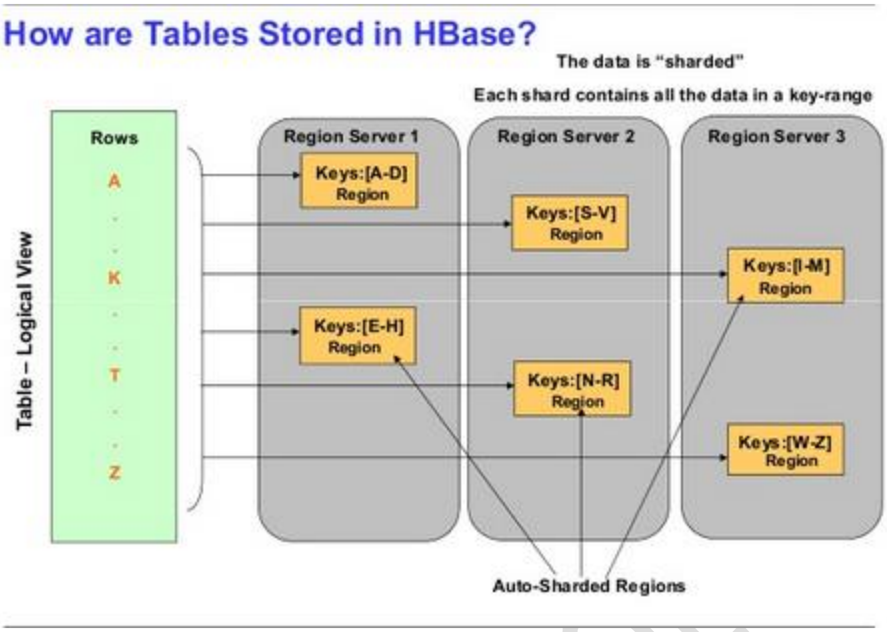
When you put a value into HBase, you have the choice of either explicitly providing a timestamp or omitting that value, which in turn is then filled in by the RegionServer when the put operation is performed.

Time oriented view: Multi dimensional



Auto sharding

The basic unit of scalability and load balancing in HBase is called a region. Regions are essentially contiguous ranges of rows stored together. They are dynamically split by the system when they become too large. Initially there is only one region for a table, and as you start adding data to it, the system is monitoring it to ensure that you do not exceed a configured maximum size. If you exceed the limit, the region is split into two at the middle key—the row key in the middle of the region—creating two roughly equal halves. Each region is served by exactly one region server, and each of these servers can serve many regions at any time.

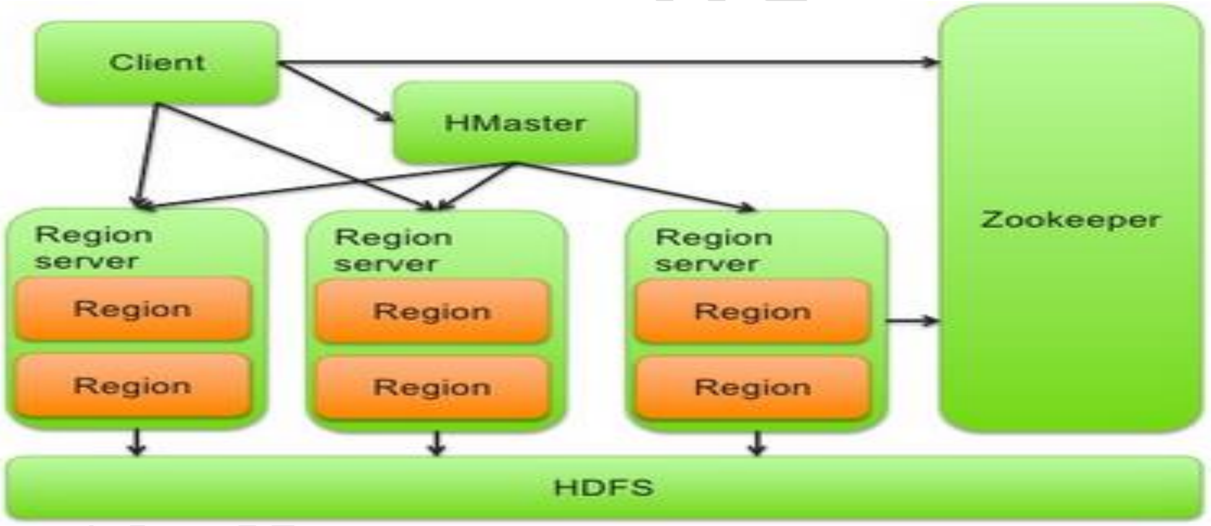


How are Tables Stored in HBase?

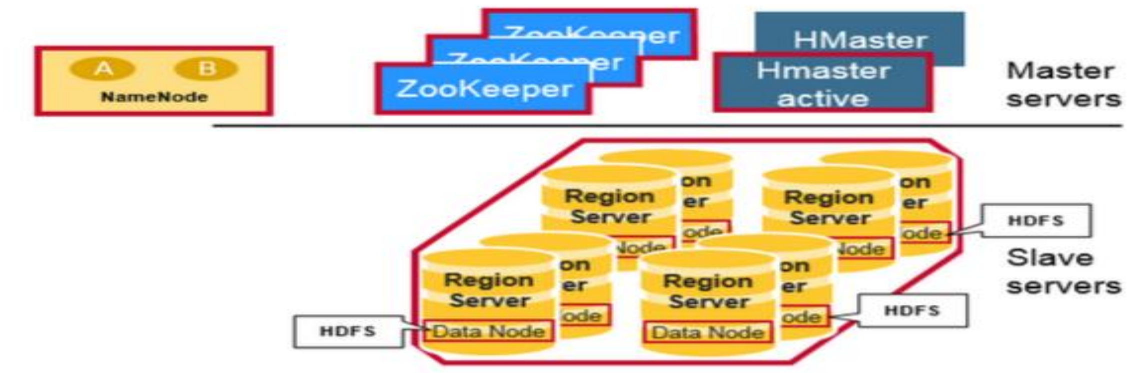
The data is "sharded"

Each shard contains all the data in a key-range

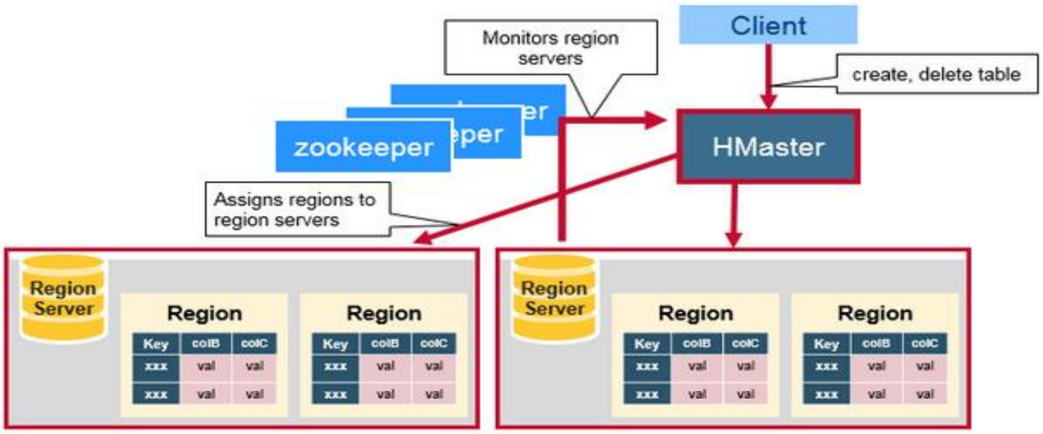
HBASE Components & Architecture



HBase Internals

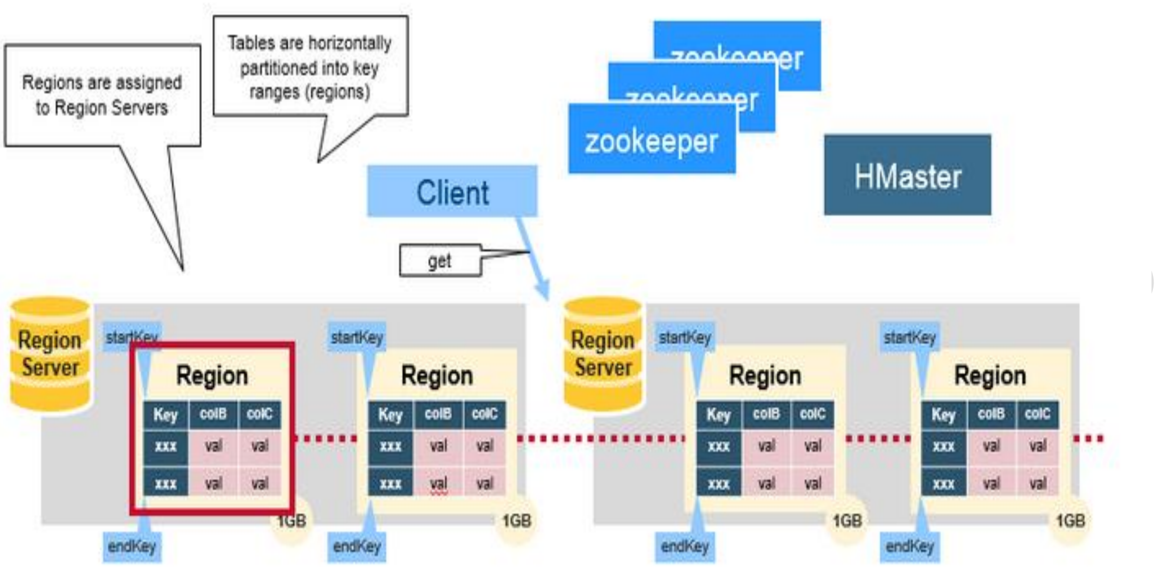


HMaster



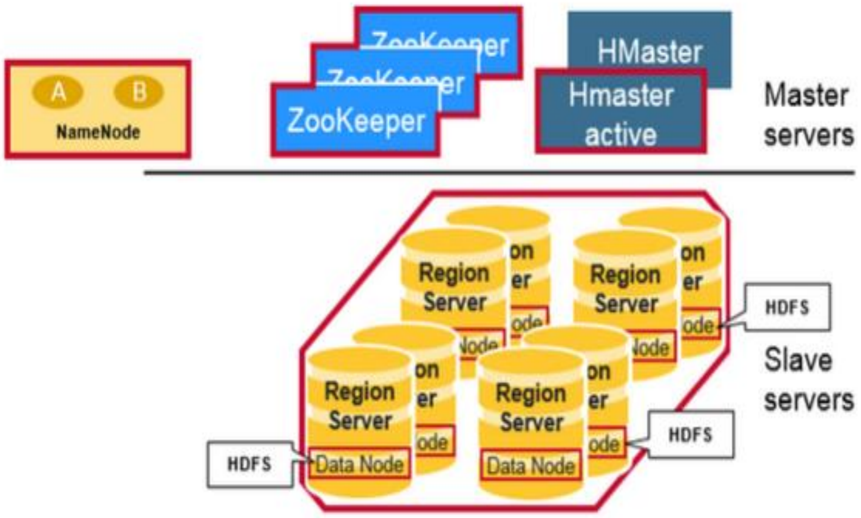
- Assigns regions to the region servers and takes the help of Apache ZooKeeper for this task.
- Handles load balancing of the regions across region servers. It unloads the busy servers and shifts the regions to less occupied servers.
- Maintains the state of the cluster by negotiating the load balancing.
- Is responsible for schema changes and other metadata operations such as creation of tables and column families.

Regions



Regions are nothing but tables that are split up and spread across the region servers. 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.

Region server



The region servers have regions that -

- Communicate with the client and handle data-related operations.
- Handle read and write requests for all the regions under it.
- Decide the size of the region by following the region size thresholds.

Region Server Components

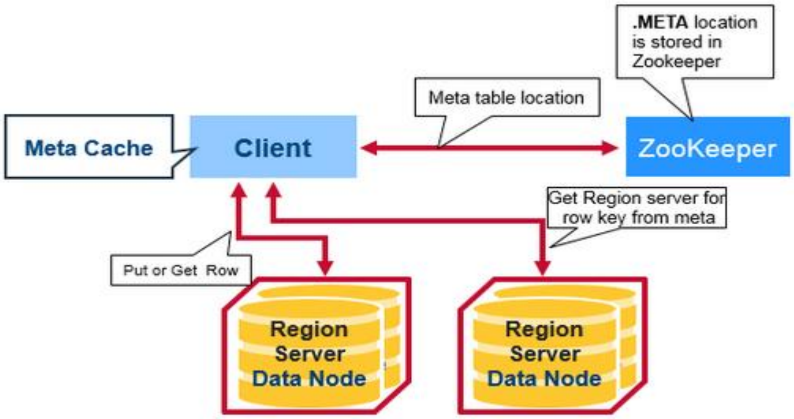
- **WAL**: Write Ahead Log is a file on the distributed file system. The WAL is used to store new data that hasn't yet been persisted to permanent storage; it is used for recovery in the case of failure.
- **BlockCache**: is the read cache. It stores frequently read data in memory. Least Recently Used 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.

Zookeeper

- Zookeeper is an open-source project that provides services like maintaining configuration information, naming, providing distributed synchronization, etc.
- Zookeeper has ephemeral nodes representing different region servers. Master servers use these nodes to discover available servers.
- In addition to availability, the nodes are also used to track server failures or network partitions.
- Clients communicate with region servers via zookeeper.
- In pseudo and standalone modes, HBase itself will take care of zookeeper.

HBase Read Operation

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.



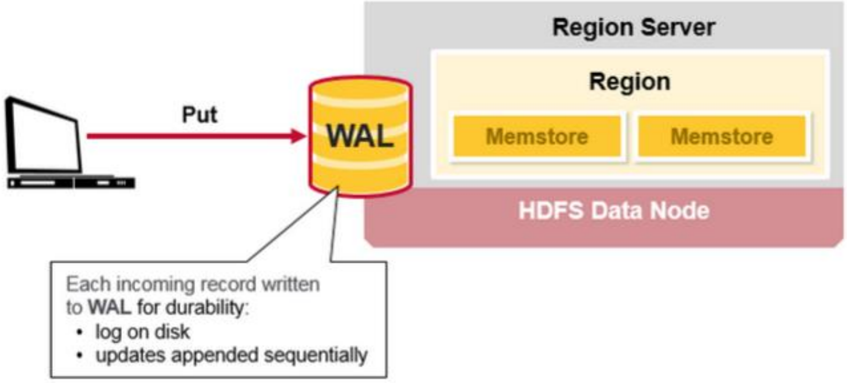
1. The client gets the Region server that hosts the META table from ZooKeeper.
2. The client will query the .META. server to get the region server corresponding to the row key it wants to access. The client caches this information along with the META table location.
3. It will get the Row from the corresponding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

For future reads, the client uses the **Block 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 Write Operation

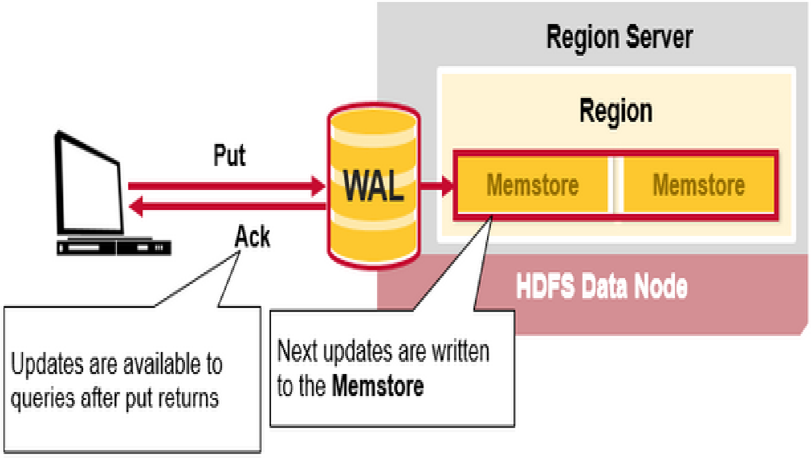
1. When the client issues a Put request, the first step is to write the data to the write-ahead log, the WAL:

- 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.



Each incoming record written to WAL for durability:
- log on disk
- updates appended sequentially

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

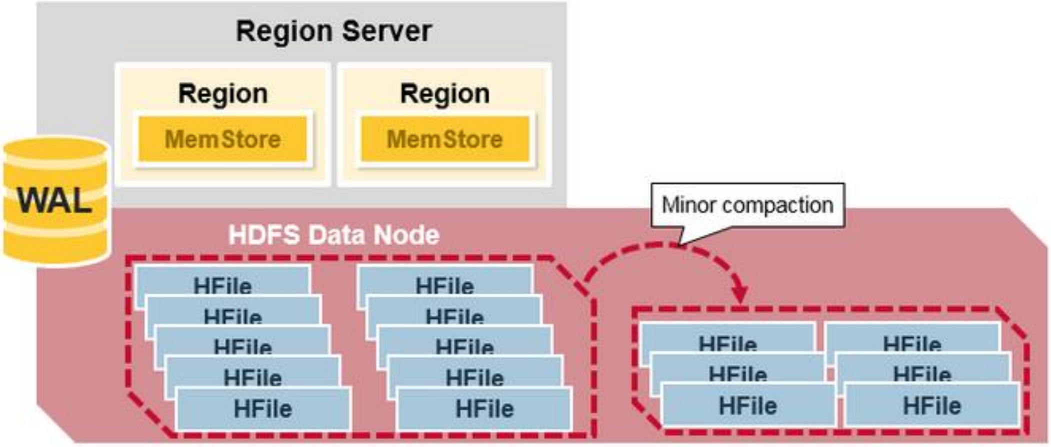


Updates are available to queries after put returns

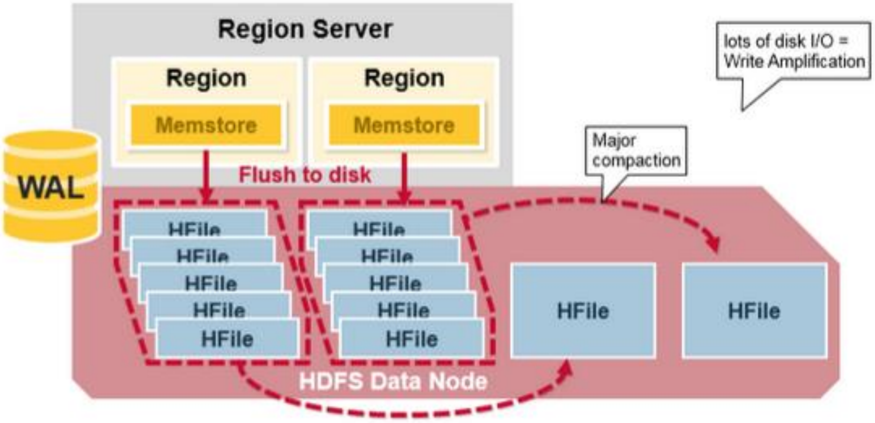
Next updates are written to the **Memstore**

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.



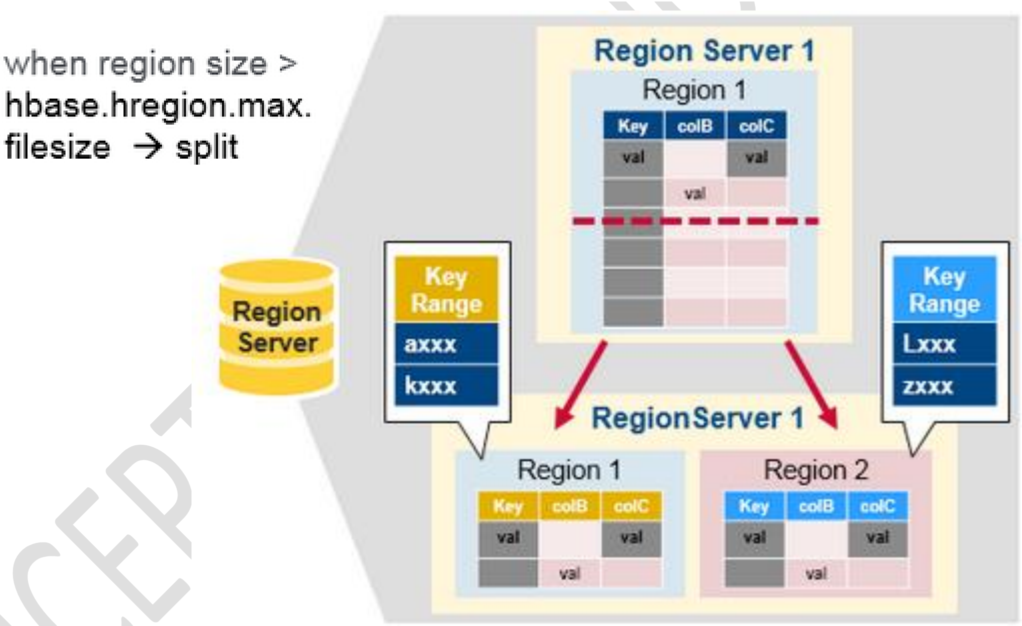
HBase Major Compaction



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 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.


when region size >
hbase.hregion.max.
filesize → split
Region Server
Region Server 1
Region 1
Key
colB
colC
val
Key Range
axxx
kxxx
Key Range
Lxxx
zxxx
RegionServer 1
Region 1
Region 2