

# CS 523 – BDT

# Big Data Technology

Lesson 10

## NoSQL DB - Apache HBase

*Spontaneous fulfilment of desires*



Maharishi International  
University

# WHOLENESS OF THE LESSON

Apache HBase is a distributed, NoSQL, column-oriented database which runs on top of HDFS and is a good choice when random, real-time, read/write/update access to very large datasets is required.

**Science & Technology of Consciousness:** Real time querying was not possible with Hadoop earlier but with the advent of HBase, all our desires of finding insights in real time and making random updates to the underlying data are going to be spontaneously fulfilled by HBase, just like TM which supports spontaneous fulfilment of desires.

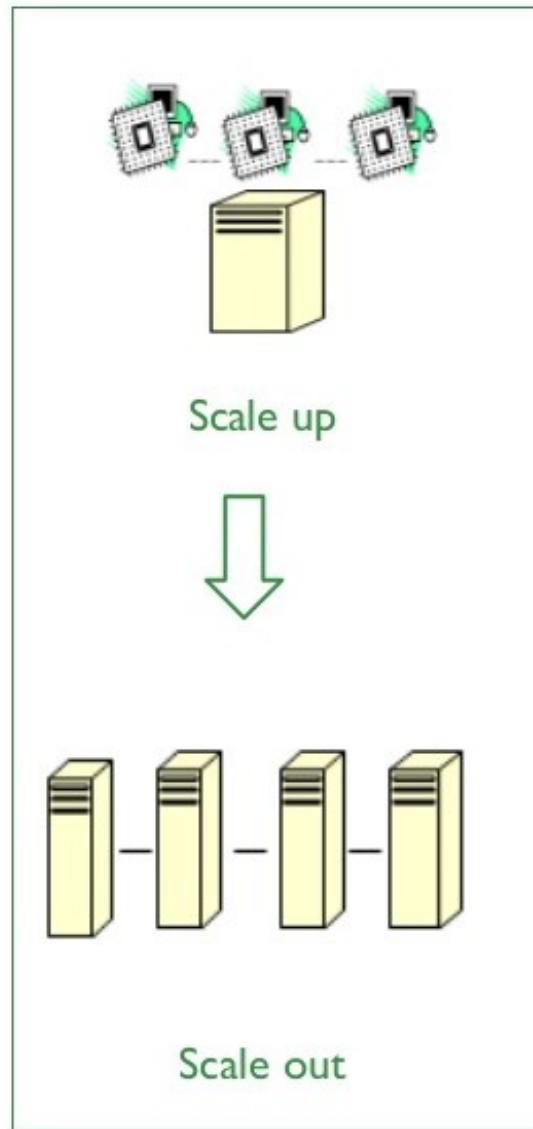
# Need for a Database

- Data storage comes in different forms that have different features.
- **Data storage as files** - fast to scan, therefore well suited for batch processing.
  - But it's difficult to find a particular record in large input files or to find a number of records that pertain to a single individual.
  - Updates to files can be difficult to coordinate.
- These actions are much better supported by some sort of database and so databases are preferred over flat files.
  - But DBs have a significant cost in “scanning performance” compared to flat files.

# Need for NoSQL Databases

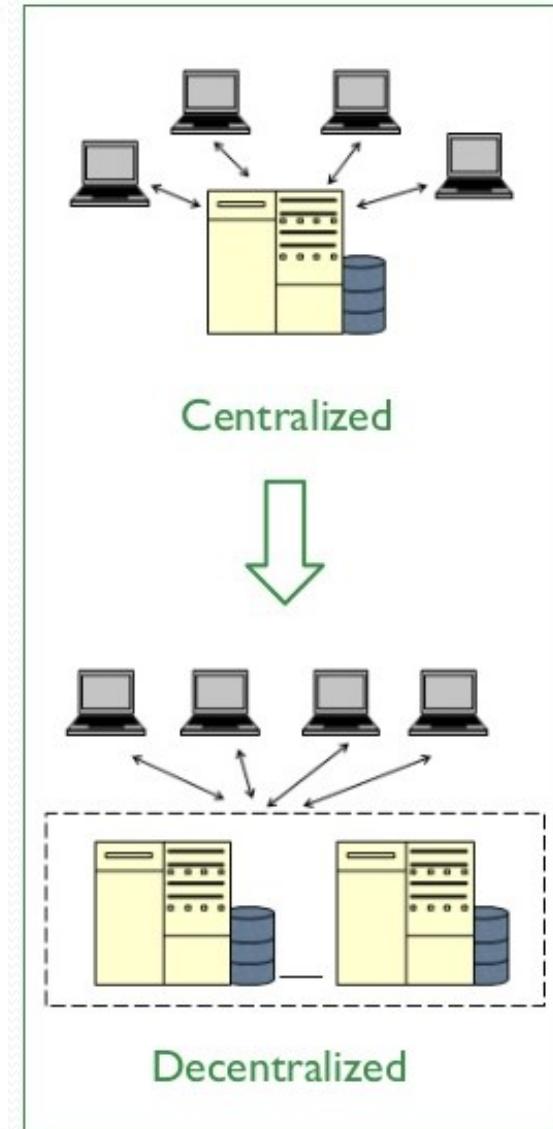
- **RDBMS** - the structure of the data should be known beforehand. Changing structure and adding new data would leave lots of NULLs in your table which is not good.
  - Lot of resources are spent on referential integrity and normalization (not a disadvantage but still takes up lot of resources)
  - Typically RDBMS are built for single server, so they are good up to a few TBs of data but they cannot handle more than 500TB of data.
  - Not able to sustain and maintain Big Data with ACID properties.
- Usage profile of applications changed - A new type of data increasingly became large that doesn't need ACID.
  - **WORM** is the property of this Data which calls for a new type of thinking for storing big data.
- **NoSQL - Not Only SQL** – is an approach for storing and retrieving high volumes of data with horizontal scaling, simple design, high flexibility and high availability.

# Non-RDBMS way of Thinking



The diagram illustrates two types of schema:

- Static schema:** Shows a table with a fixed structure. The columns are **Userid**, **Fname**, and **Blog**. The rows are User1 (Chhavi, www.bigdata.imitus.com), User2 (John, www.imitus.co.in), and User3 (Larry, www.imitus.com/webinar).
- Dynamic schema:** Shows a table where the columns can change. The first row is **User1 Chhavi**. The second row has three columns: **User3**, **Larry**, and **Pearson**. The third row has four columns: **User2**, **John**, **www.imitus.co.in**, and **www.imitus.co.in/webinar**.



# Comparison of NoSQL with RDBMS

- **SQL databases** - very rigid, inflexible data models made up of rows & columns in the form of tables, linked together through foreign keys.
- **NoSQL** - family of databases which do not conform to the SQL standard of data organization
  - Revolutionary form of data storage in modern web applications.
- **NoSQL databases often have no fixed schema whatsoever.**
  - For instance, in Document-orientated Databases, data is stored in document format i.e. JSON, and each record in the database could look entirely different from each other.

# Comparison of NoSQL with RDBMS

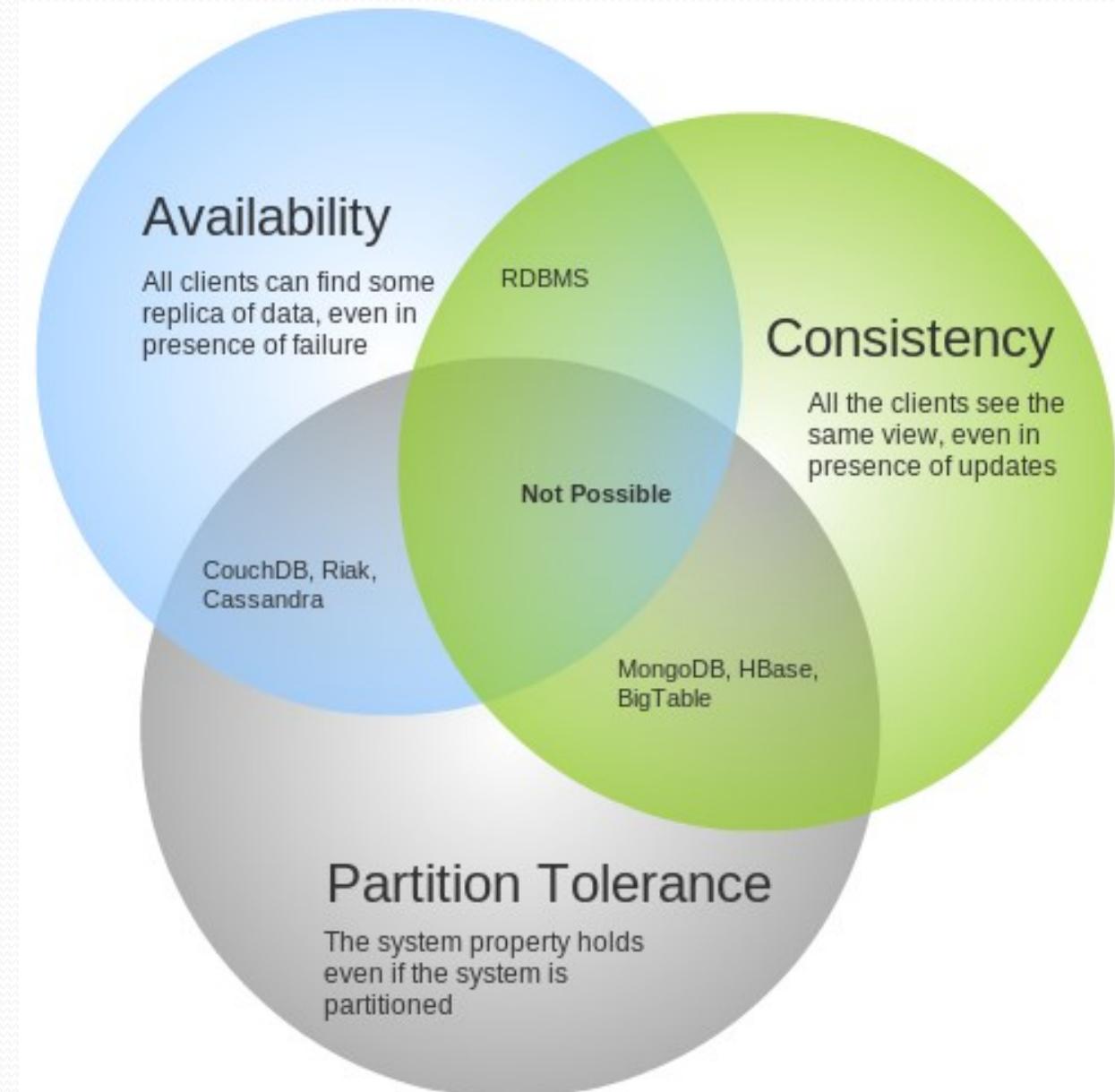
- NoSQL databases have given up some of the capabilities of RDBMSs such as advanced indexing and transactions in order to allow them to scale to much higher throughput and data sizes that are not possible with relational systems.
- So, NoSQL DBs do not support:
  - Joins
  - Transactions
  - ACID
  - Referential integrity
  - Heavy checks on data
- NoSQL databases are not intended to completely replace relational databases. Each has its own strengths and should be used for what it does best.

# Types of NoSQL Databases

Graph	Column	Document	Persistent Key/Value	Volatile Key/Value
<a href="#"><u>neo4j</u></a>	<a href="#"><u>BigTable</u></a> (Google)	<a href="#"><u>MongoDB</u></a> (~BigTable)	<a href="#"><u>Dynamo</u></a> (Amazon)	<a href="#"><u>memcached</u></a>
<a href="#"><u>FlockDB</u></a> (Twitter)	<a href="#"><u>HBase</u></a> (BigTable)	<a href="#"><u>CouchDB</u></a>	<a href="#"><u>Voldemort</u></a> (Dynamo)	<a href="#"><u>Hazelcast</u></a>
<a href="#"><u>InfiniteGraph</u></a>	<a href="#"><u>Cassandra</u></a> (Dynamo + BigTable)	<a href="#"><u>Riak</u></a> (Dynamo)	<a href="#"><u>Redis</u></a>	
	<a href="#"><u>Hypertable</u></a> (BigTable)		<a href="#"><u>Membase</u></a> (memcached)	
	<a href="#"><u>SimpleDB</u></a> (AmazonAWS)		<a href="#"><u>Tokyo Cabinet</u></a>	

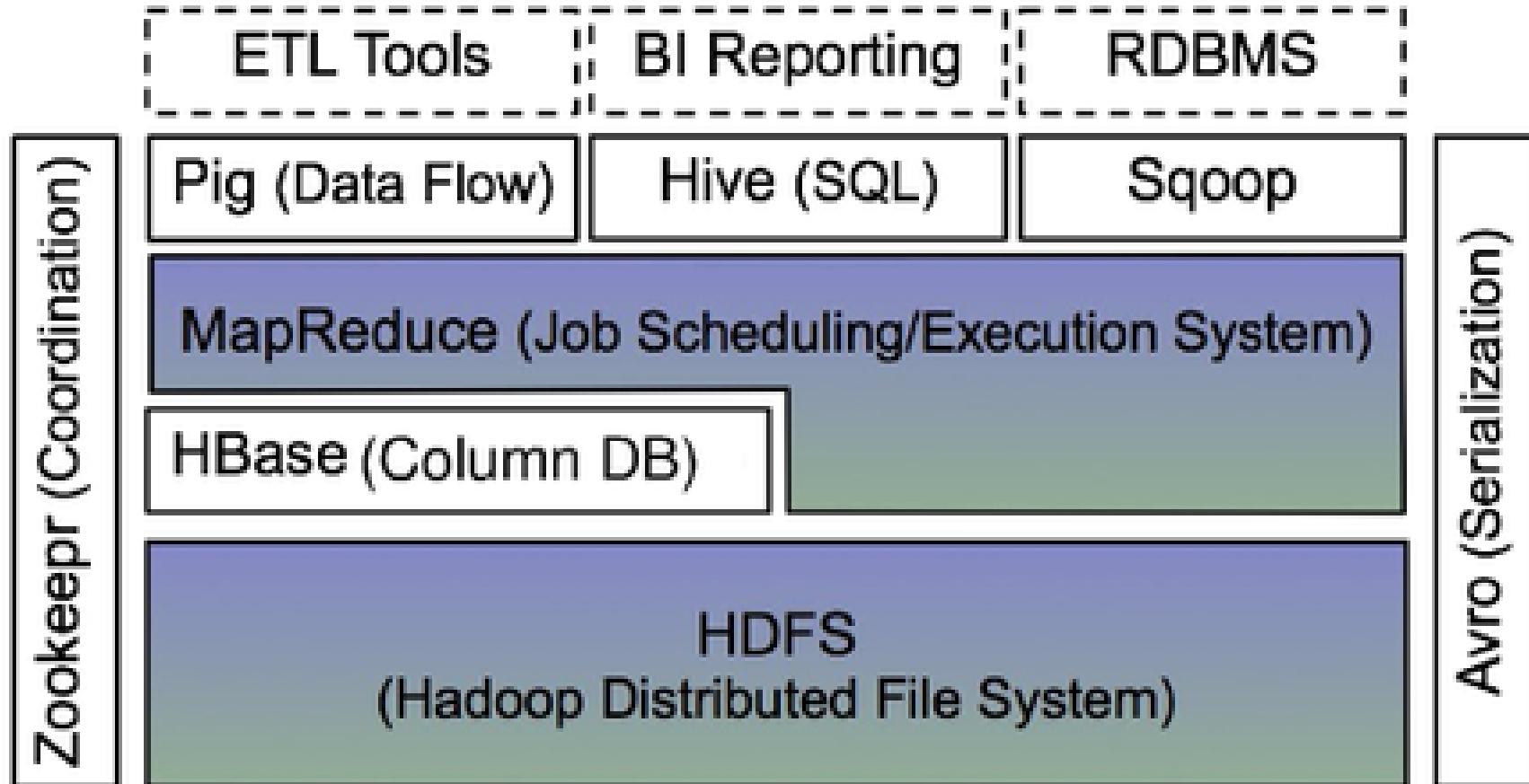
# CAP Theorem

- The CAP theorem states that any networked shared-data system can have at most two of the three desirable properties:
- **Consistency (C)** equivalent to having a single up-to-date copy of the data;
- **High Availability (A)** of that data (for queries); and
- **Partition (P) Tolerance** even if the connections between nodes are down, the other two (A & C) promises, are kept.



- **Horizontally scalable, distributed, NoSQL DB**
  - Automatic sharding (horizontal partitioning)
- **Strongly consistent reads and writes**
- **Automatic fail-over**
- **Simple Java API**
- **Integration with MapReduce/Spark frameworks**
- **Quick access to data**
- **Variable schema support**

# Hadoop Ecosystem



# Apache HBase

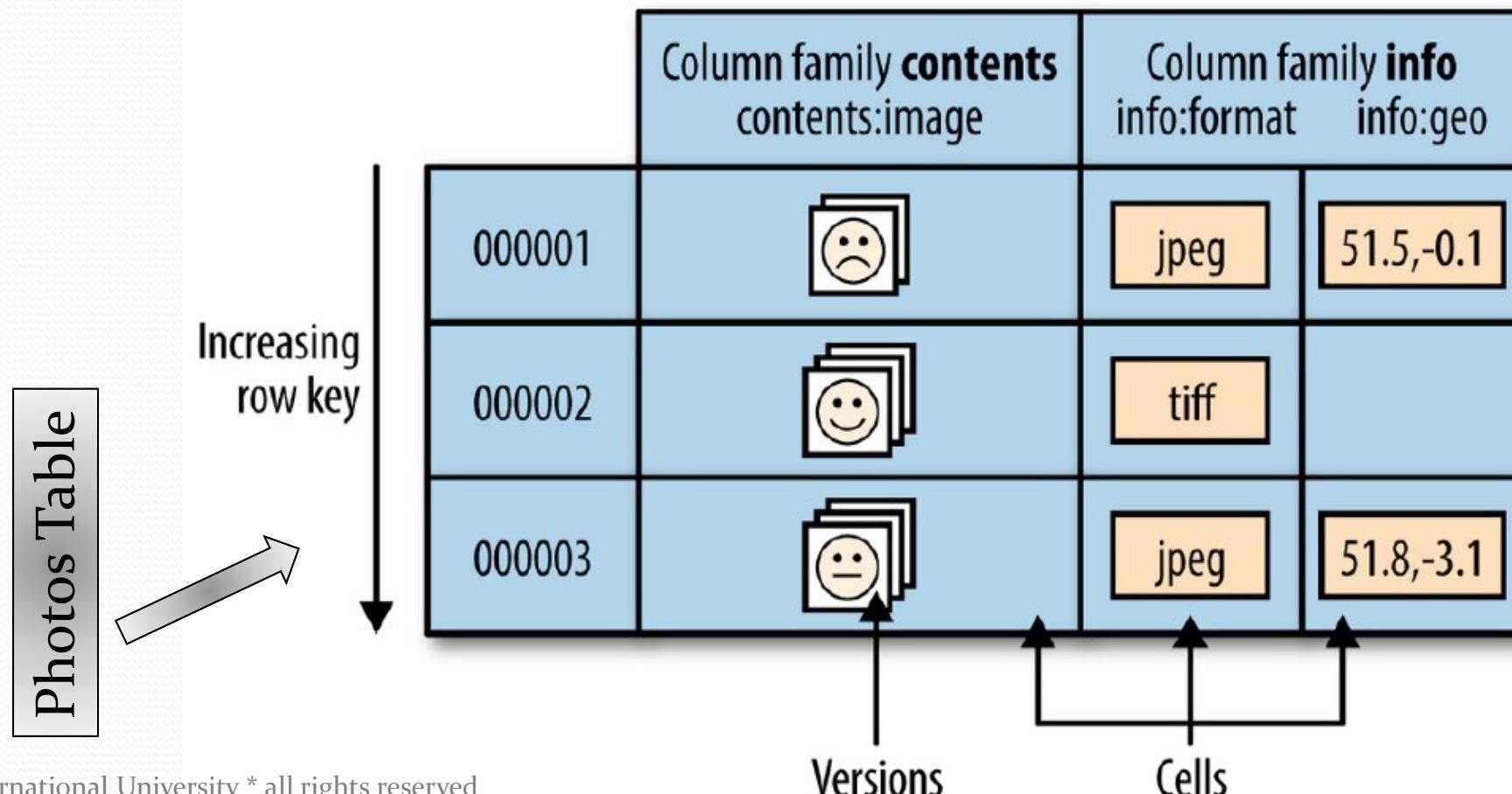
- **Apache HBase** - distributed, NoSQL column-oriented database which runs on top of HDFS and is a good choice when random real-time read/write/update access to very large datasets is required.
- Based on [Google's BigTable](#).
- HBase is not relational and does not support SQL. But it can host very large, sparsely populated tables on clusters made from commodity hardware.
- An example of HBase use case is the webtable, a table of crawled web pages and their attributes (such as language and MIME types) keyed by the web page URL. The webtable is large, with row counts that run into the billions.
  - Batch analytic and parsing mapreduce jobs are continuously run against the webtable, deriving statistics and adding new columns of verified MIME-type and parsed text content for later indexing by a search engine.
  - Concurrently, the table is randomly accessed by crawlers running at various rates and updating random rows while random web pages are served in real time to users.

# Supported Operations in HBase

- HBase supports five primary operations:
  - **create** to create tables
  - **put** to add or update rows
  - **scan** to retrieve a range of cells
  - **get** to return cells for a specified row
  - **delete** to remove rows, columns or versions from the table.
- Versioning is available so that previous values of the data can be fetched (the history can be deleted every now and then to clear space via HBase compactions).
- No support for “typed columns” like RDBMS – everything is byte array - so developer is responsible to keep track of column data types.
- In order to run HBase, *ZooKeeper* is required - a server for distributed coordination such as configuration, maintenance, and naming.

# HBase Concepts

- Applications store data into HBase as labeled tables, and tables are further split into **rows** and **column families**.
- Column families, which must be declared in the schema, group together a certain set of columns (columns don't require schema definition).



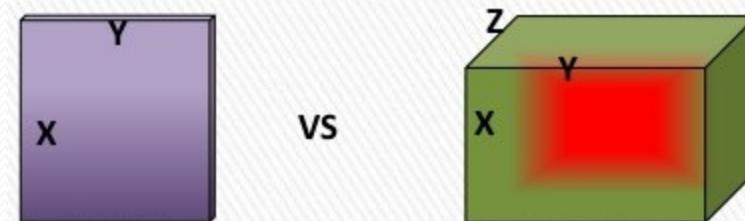
# HBase Concepts: Row-Key

- A row in HBase is a grouping of key/value mappings identified by the row-key.
- Row-keys are also byte arrays, so theoretically anything can serve as a row-key, from strings to binary representations of long or even serialized data structures.
- Table rows are sorted by row key, which is also called as the table's Primary Key. The sort is byte-ordered. All table accesses are via the PK.
- **All the data pertaining to a single row-key is stored on one single node.**

Row Key	Data
'login_2012-03-01 00:09:17'	d:{'user':'alex'}
...	...
'login_2012-03-01 23:59:35'	d:{'user':'otis'}
'login_2012-03-02.00:00:21'	d:{'user':'david'}

# HBase Concepts-Cells

- HBase stores cells individually.
  - Great for “sparse” data
- A cell’s content is an uninterpreted array of bytes.
- Each cell in Hbase is versioned. By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion.
- Each cell in HBase is defined as a key/value pair, and each key to identify a cell is made up of row-key, CF name, column name and time-stamp.
  - So, it is advised to keep all these names short
  - And serialize and store many values into single cell



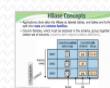
# Concepts: Column-Family

- Row columns are grouped into column families and **column families are internally stored in HDFS as files**.
- All column family members have a common prefix, so, for example, the columns *info:format* and *info:geo* are both members of the *info* column family, whereas *contents:image* belongs to the *contents* family.
- Typically a table will have small number of column families which shouldn't change frequently.
- Column families are used to store frequently accessed data together. Settings can be customized on a per column family basis.
- A table's column families must be specified up front as part of the table schema definition, but new column family members (new columns) can be added on demand.



# Concepts: Column-Family contd...

- Physically, all column family members are stored together on the filesystem.
- Tuning and storage specifications are done at the column family level, it is advised that all CF members have the same general access pattern and size characteristics.
  - For the photos table, the image data, which is large (MBs), is stored in a separate column family from the metadata, which is much smaller in size (KBs).
- Separate column families are useful for:
  - Data that is not frequently accessed together
  - Data that uses different column family options (compression)



HBase tables are like those in an RDBMS, only cells are versioned, rows are sorted, and columns can be added on the fly as long as the CF they belong to pre-exists.

Types can differ between rows

	fname	lname	picture
aputrell@apache.org	"Andrew"	"Putrell"	
jdcryans@apache.org	"Jean-Daniel"	1337	downfall.jpg
stack@apache.org		"Stack"	dancing_stack.jpg
todd@apache.org	"Todd"	"Lipcon"	turbo.jpg
			speedy.jpg
			fast.jpg

Columns without data added to them don't exist

Previous versions of data can be accessed

The diagram illustrates a row in a column-oriented database. The row key is 'todd@apache.org'. It contains four columns: 'fname' (value "Todd"), 'lname' (value "Lipcon"), and 'picture' (value "turbo.jpg"). The fourth column is empty. A callout box points to this empty column with the text "Columns without data added to them don't exist". Another callout box points to the 'picture' column with the text "Previous versions of data can be accessed", and it shows three previous versions of the 'picture' column: "speedy.jpg", "fast.jpg", and the current value "turbo.jpg". A callout box at the top points to the row with the text "Types can differ between rows", which is true because the 'lname' column contains a string while the 'picture' column contains a file path.

## Things to remember:

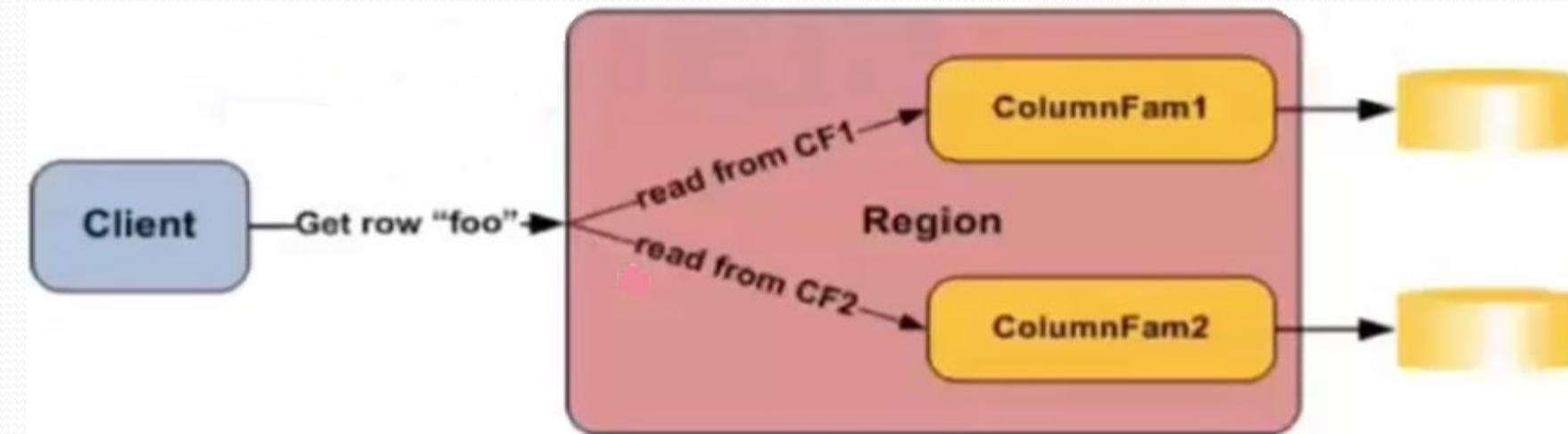
- All the data pertaining to a single row-key is stored on one single node
- Rows are stored sorted by row key for fast lookups
- Each column family is internally stored in HDFS as a single file

# Main Point

Each cell in HBase is defined as a key/value pair, and each key to identify a cell is made up of row-key, CF name, column name and time-stamp. So we can say HBase is like a distributed hash map where each cell of a table is stored independently. **Science & Technology of Consciousness:** Even though each HBase cell is stored independently, it is still the part of the same underlying big dataset in HDFS. Similarly, even though all the things in the relative world look different, they are nothing but the distributed impulses of the underlying pure field of intelligence which is the absolute source of all relative creation and can be directly experienced through the human nervous system.

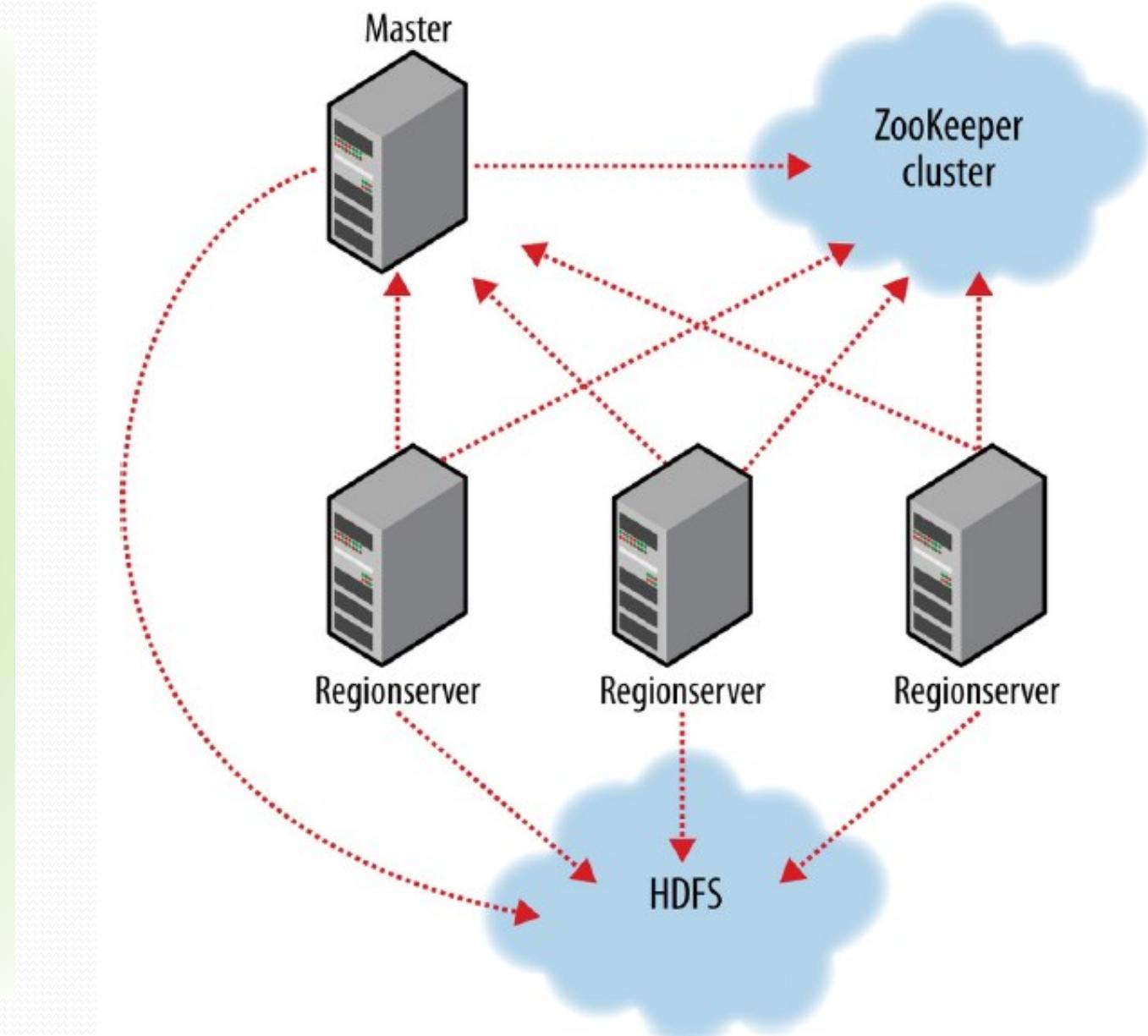
# Designing Tables

- **Same row-key = same node**
  - Lookups by row-key talk to a single node or “region server” (discussed later)
- **Same column-family = same set of physical files**
  - Retrieving data from a column family is sequential I/O

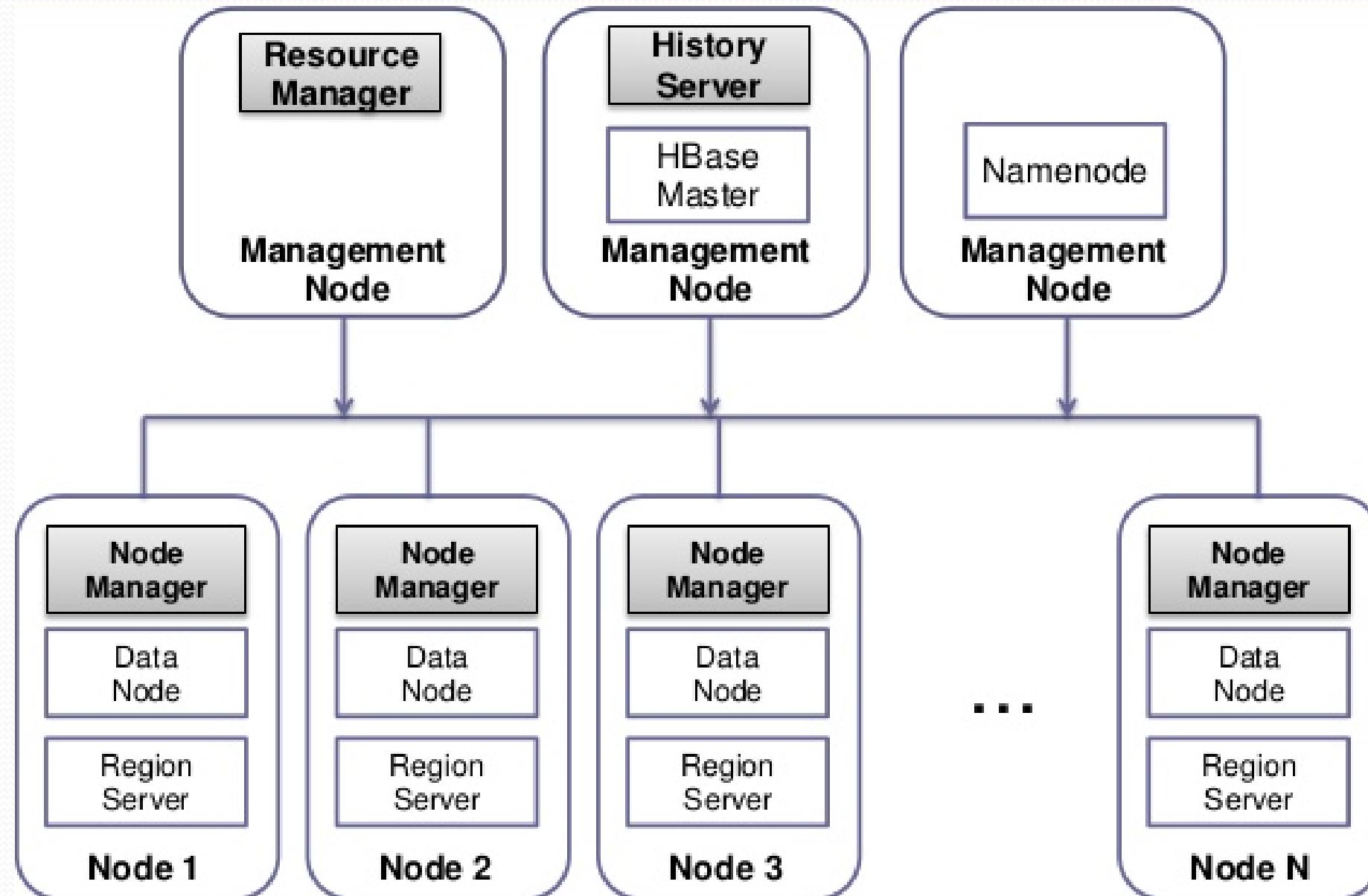


# HBase Cluster

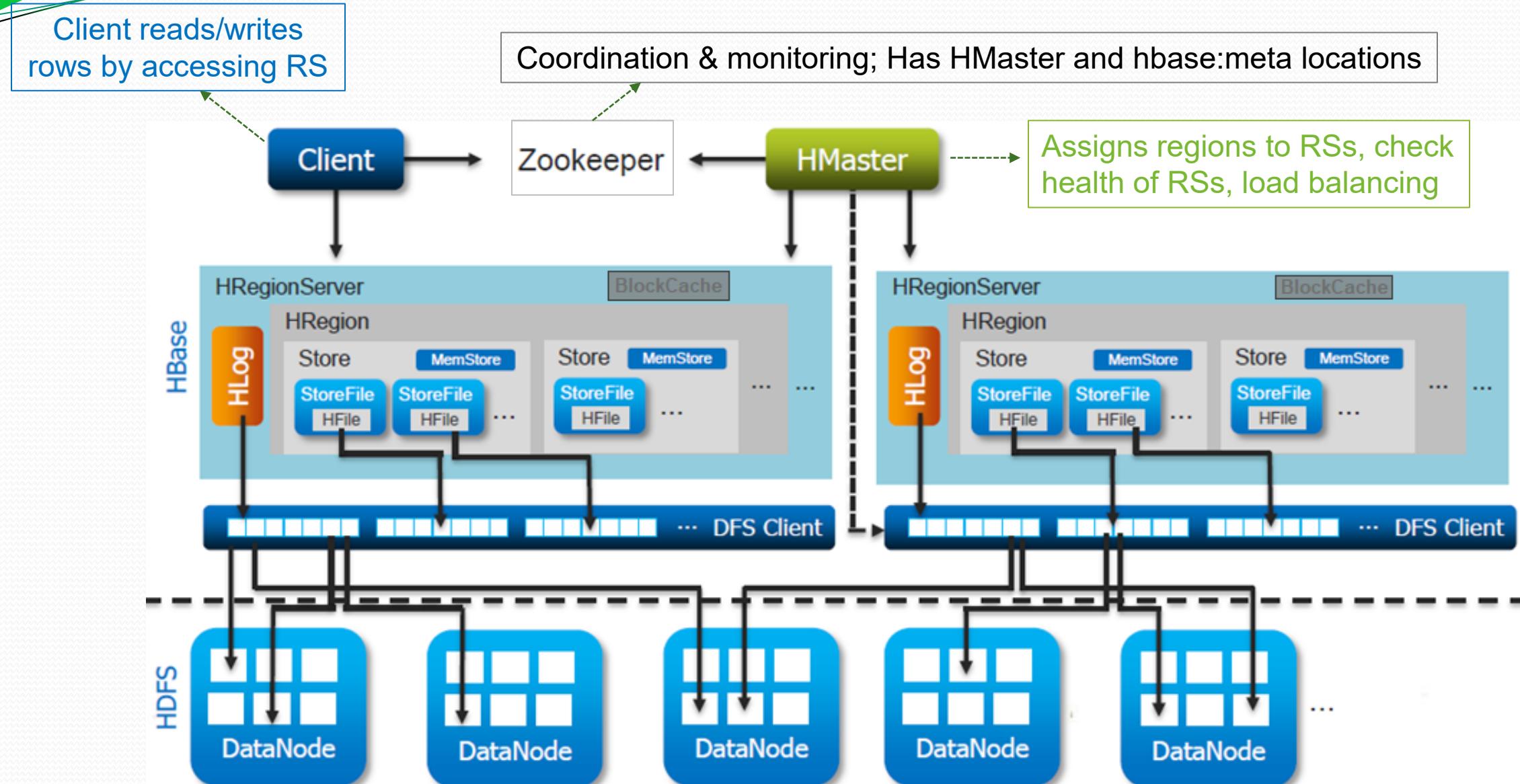
- **Master-Slave architecture**
- Master is **HMaster** and each slave has **RegionServer** process running.
- Clients do not talk directly with HMaster; instead all client requests first go to **Zookeeper** and then to RegionServers directly.



# HDFS, YARN, HBase Daemons



# HBase Architecture



# Major Components of HBase

- 1. Zookeeper**
- 2. HMaster**
- 3. HRegionServer**
- 4. HRegion**
- 5. Catalog table**
  - **hbase:meta**

# ZooKeeper



- Distributed applications require coordination.
  - Develop your own service (lot of work!) or use robust pre-existing one (Zookeeper)
- Zookeeper is Distributed open source Centralized coordination service for:
  - Maintaining configuration information
    - E.g. Sharing “config” info across all nodes.
  - Naming
    - E.g. Find a machine in a cluster of 1,000s of servers – Naming Service
  - Providing distributed synchronization
    - E.g. Locks, barriers, queues, etc.
  - Providing group services
    - E.g. Leader election and more

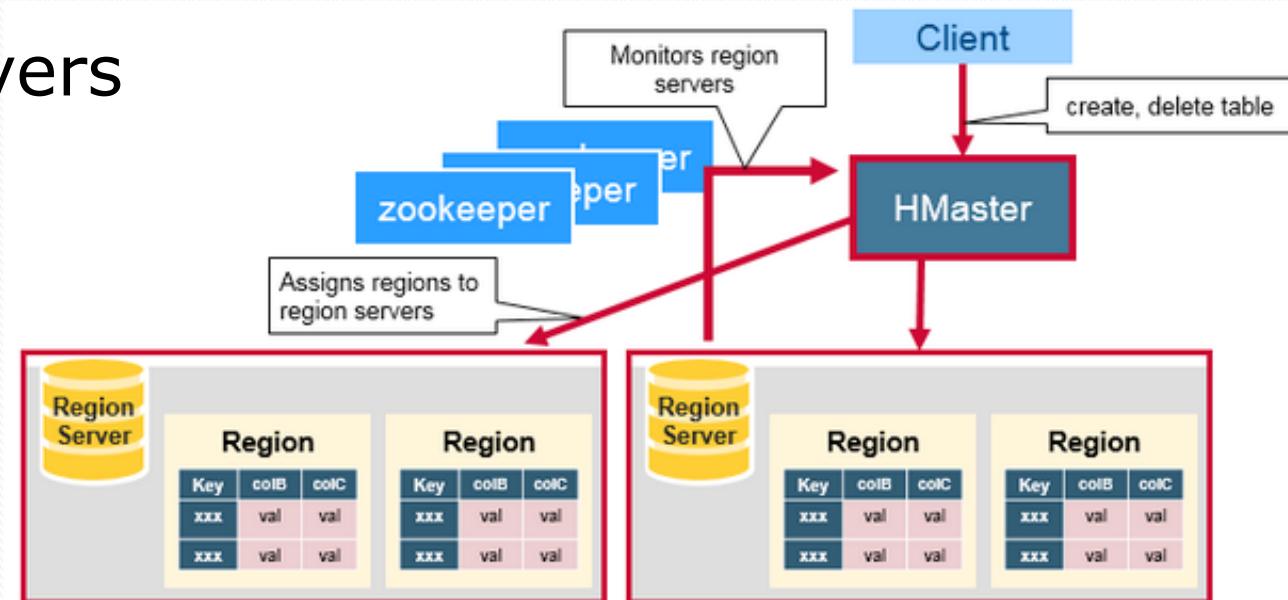
# ZooKeeper contd...

- If client wants to communicate with regions, the client has to approach ZooKeeper first.
  - Zookeeper is responsible for Client Communication establishment with region servers.
- It also tracks server failure and network partitions.
- HMaster and HRegionServers register themselves with ZooKeeper.
- During a failure of nodes that are present in HBase cluster, ZooKeeper will trigger error messages, and it starts to repair the failed nodes.
- The Zookeeper service is Distributed, Reliable, Fast and Simple!

# HMaster

- Assigns Regions to RegionServers
- Handles DDL
- Splits region if necessary
- Moves region to balance load
- Single point of failure

- There can be multiple HMaster, just in case of backup in an inactive state.
- The active HMaster sends heartbeats to ZooKeeper, and the inactive one waits.
- Master server crash cannot cause data loss.



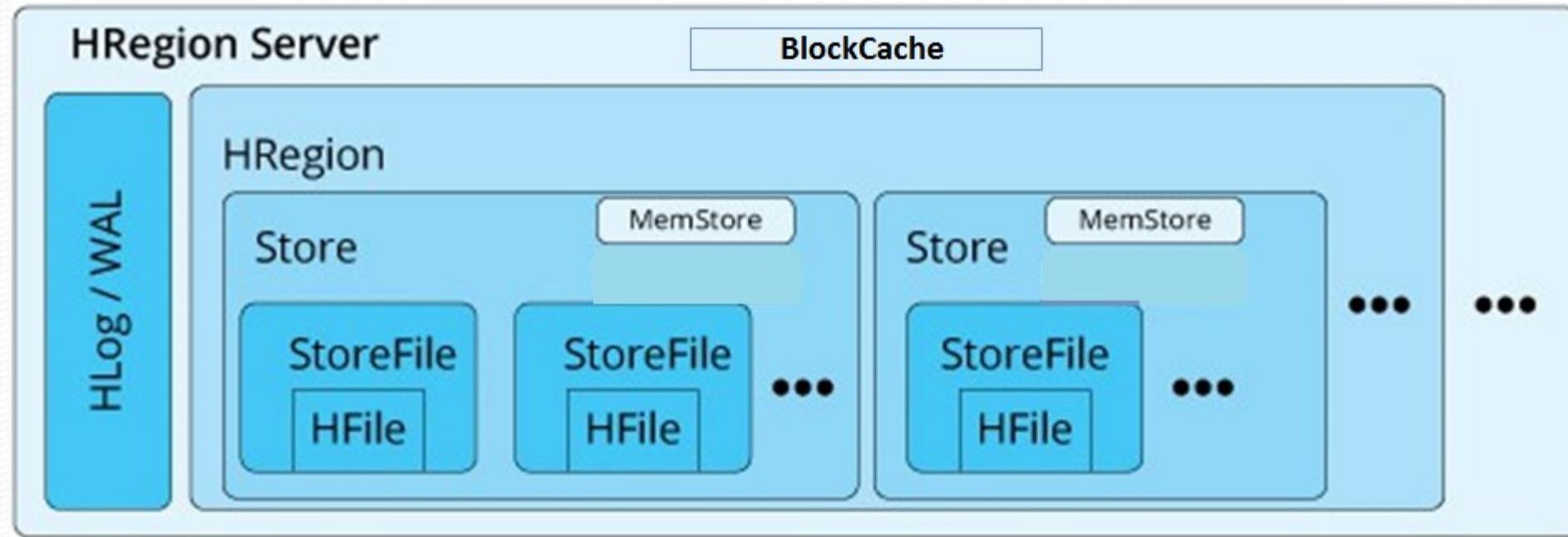
# RegionServer

A Region Server runs on an HDFS data node and has the following components:

- **Multiple HRegions**
- **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**: It is the read cache. It stores frequently read data in memory. Least Recently Used data is evicted when full.
- **MemStore**: It 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.

# RegionServer

contd...



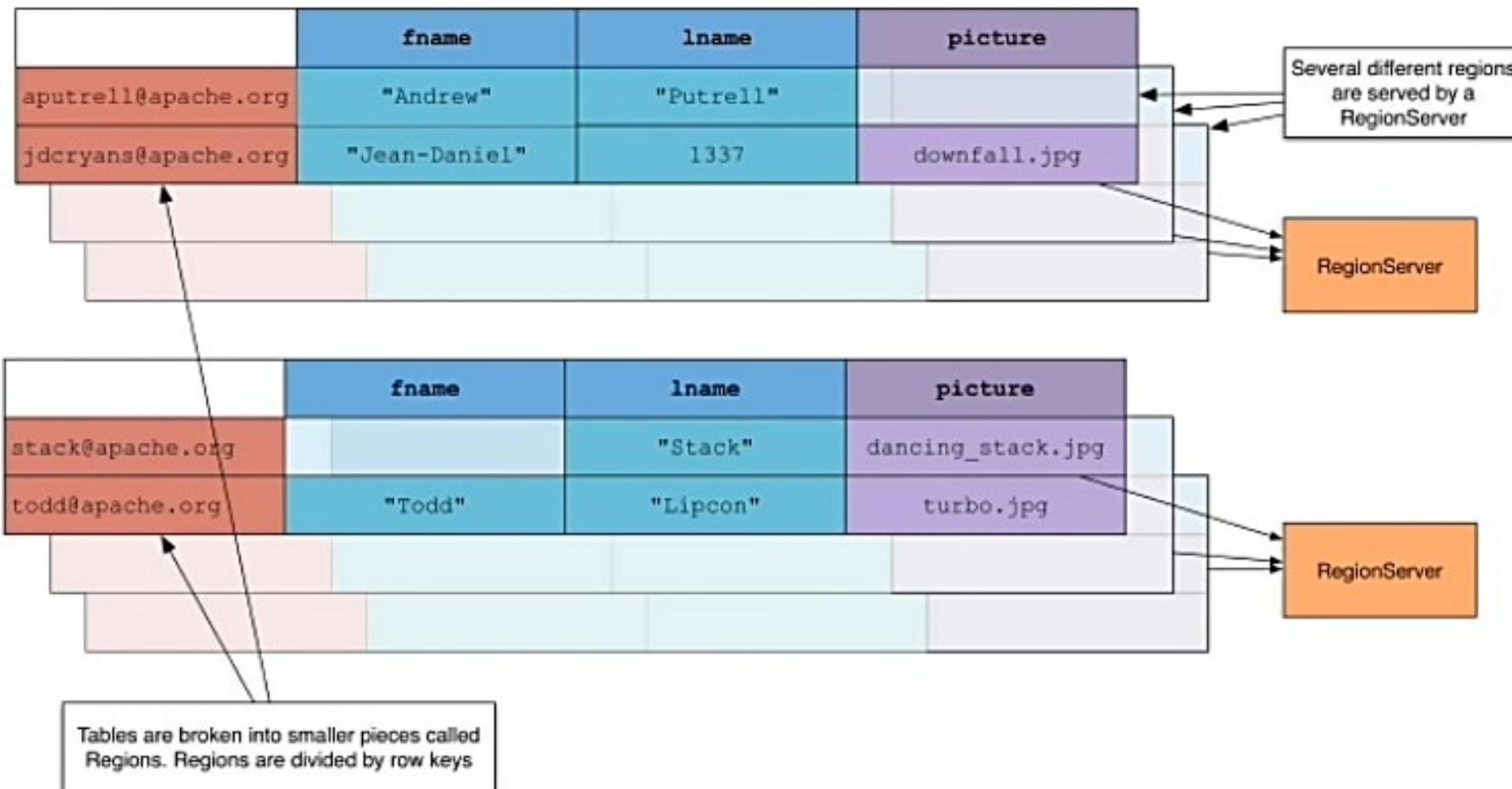
- A RegionServer contains a single WAL (HLog), single BlockCache and multiple Regions.
- A Region consists of multiple Stores for each column family.
- A Store consists of multiple StoreFiles and one MemStore.
- A StoreFile corresponds to a single HFile.
- HFiles and HLog are persisted on HDFS.

# Regions (HRegions)

- Regions are the units that get distributed over an HBase cluster (Regions are fundamental partitioning object).
- Initially, a table comprises a single region, but as the region grows it eventually crosses a configurable size threshold (256 MB), at which point it splits at a row boundary into two new regions of approximately equal size.
- Until this first split happens, all loading will be against the single server hosting the original region. As the table grows, the number of its regions grows.
- In this way, a table that is too big for any one server can be carried by a cluster of servers, with each node hosting a subset of the table's total regions.

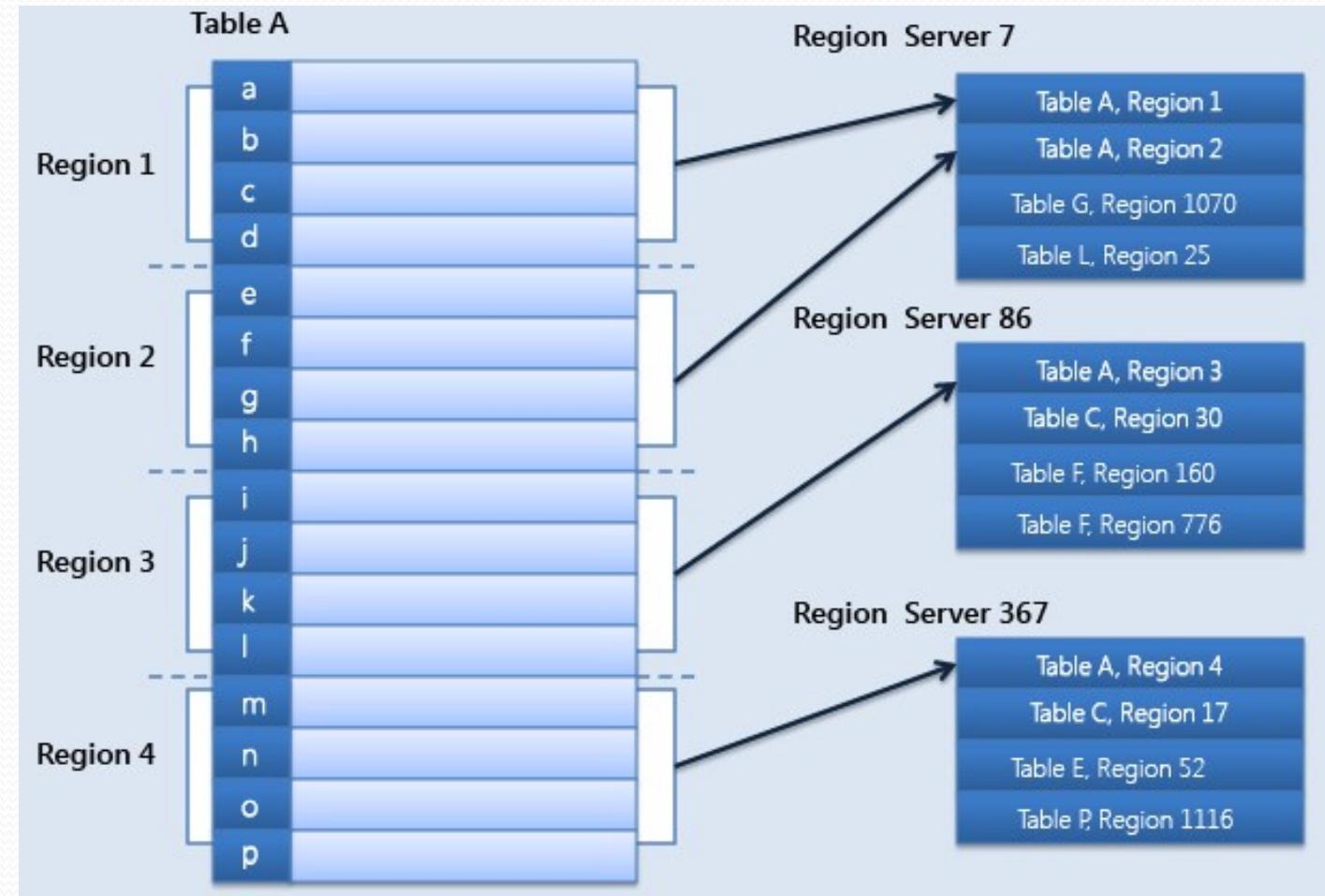
# Regions (HRegions) contd...

- Tables are automatically partitioned horizontally by HBase into regions. Each region comprises a subset of a table's rows.



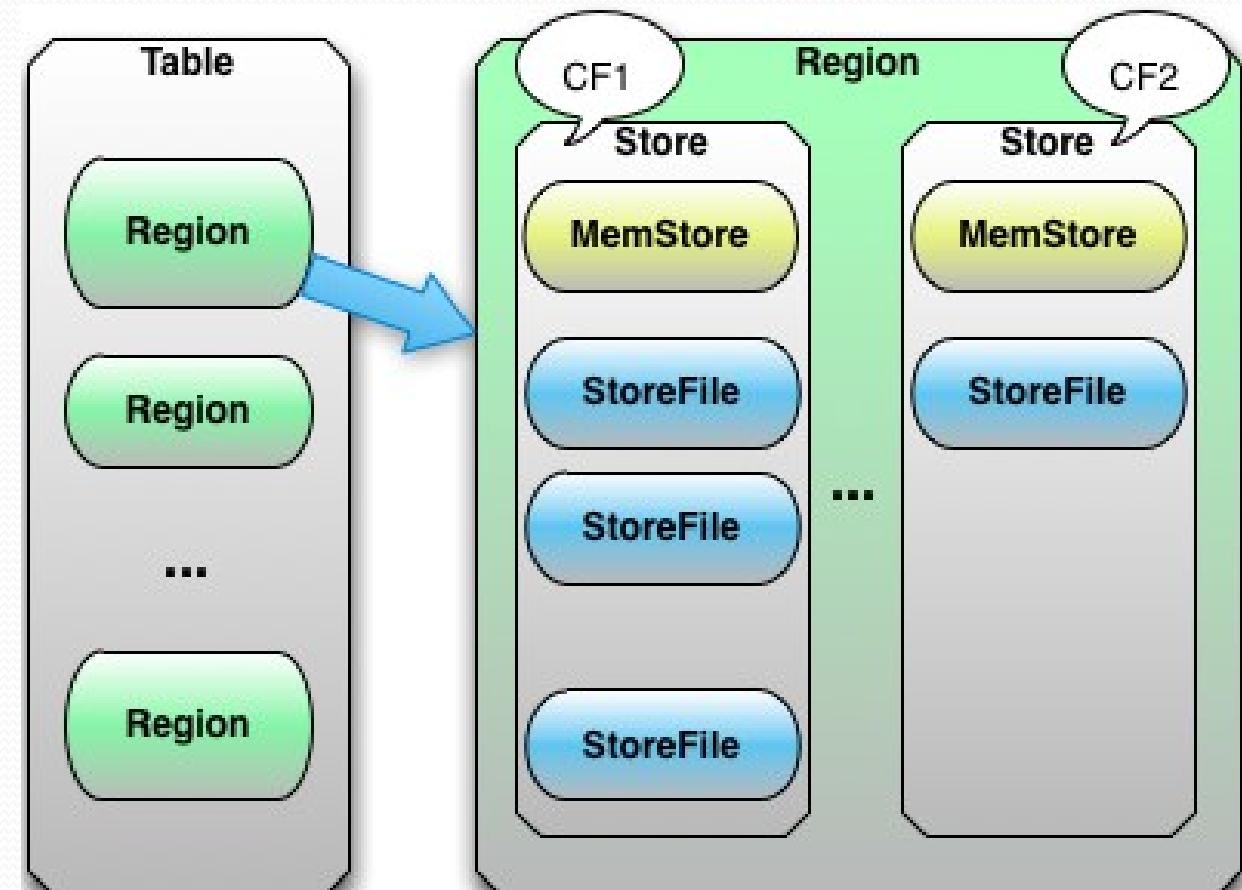
# Regions (HRegions) contd...

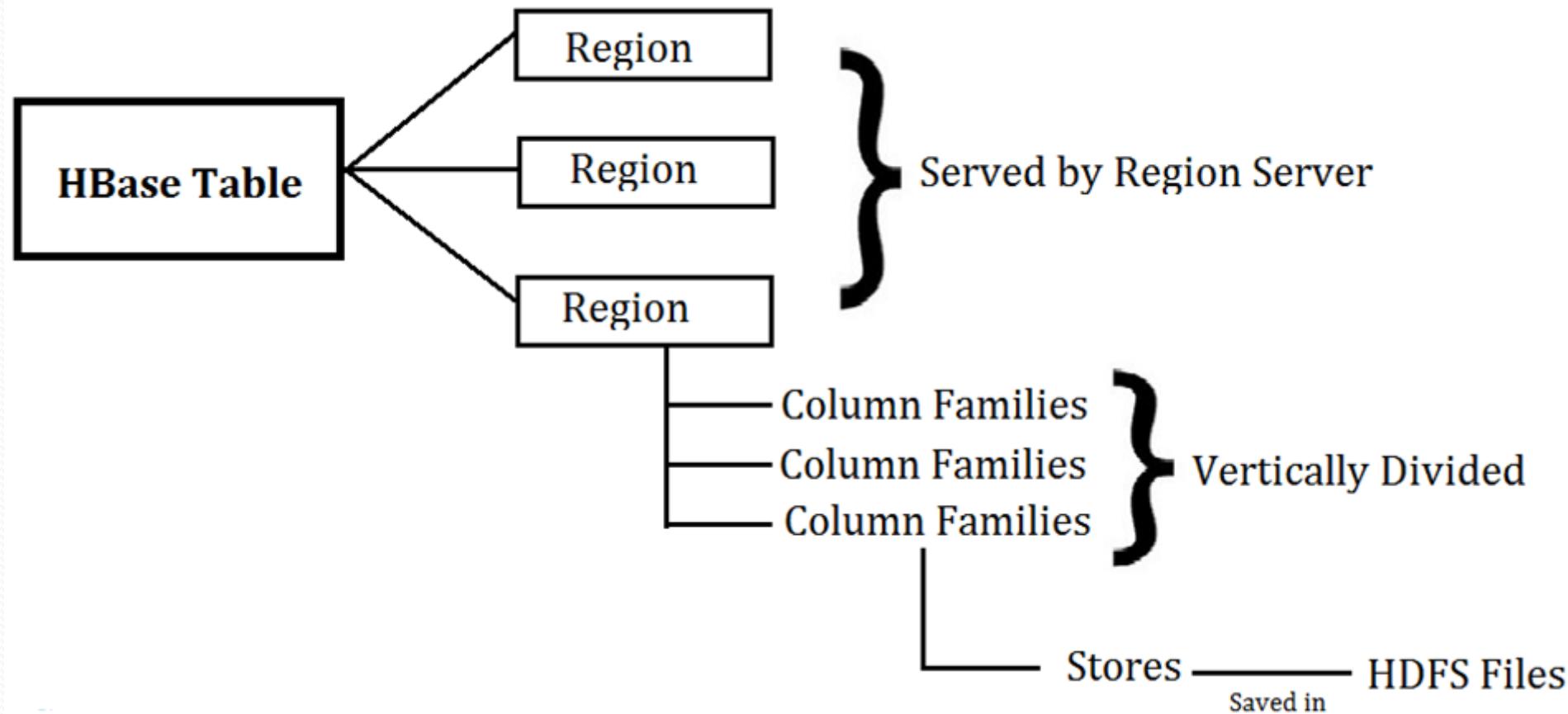
- A region contains a contiguous, sorted range of rows between a start key and an end key.
- Recommended max region size is 10-50Gb
- Default is 256Mb.



# Regions (HRegions) contd...

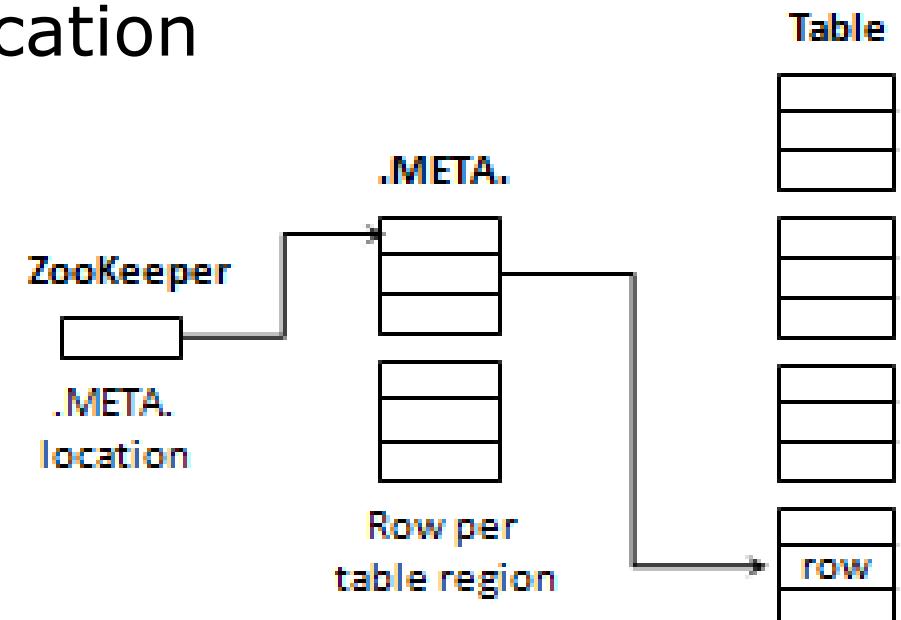
- A Region contains multiple Stores, one for each Column Family.
- Each Store has a MemStore and multiple StoreFiles.
- A region of a table is served to the client by a RegionServer.
- A RegionServer can serve about 1,000 regions (which may belong to the same table or different tables)





# Catalog Table: hbase:meta

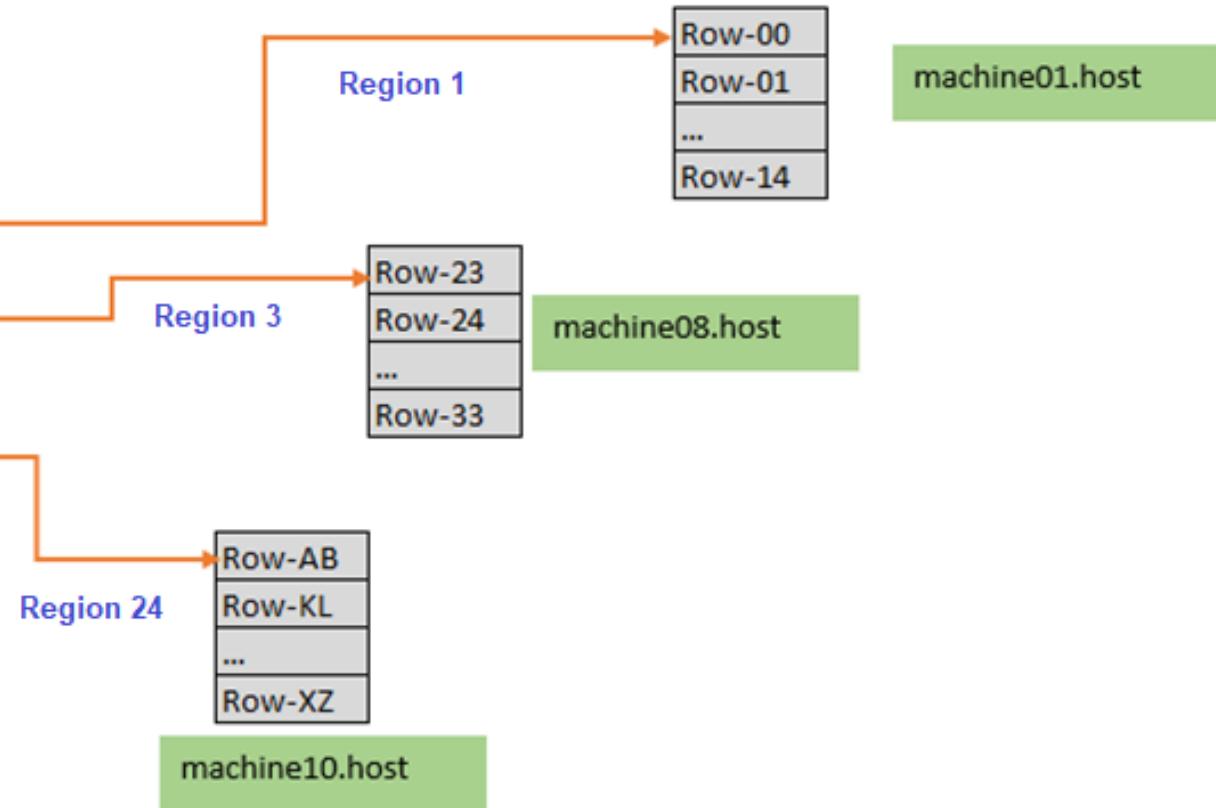
- Since tables are partitioned and stored across the cluster, how can the client find which region is hosting a specific row key range?
- **hbase:meta** table is used for this purpose.
- Zookeeper stores the location of hbase:meta table.
- hbase:meta table hosts the region location info for a specific row key range.
  - This meta table is stored on RSs, which can be split into as many regions as required.



# hbase:meta Table Structure

## META TABLE

Table	Start Key	Region ID	RegionServer
table1	Row-00	1	machine01.host
table1	Row-15	2	machine05.host
table1	Row-23	3	machine08.host
table1	Row-34	4	machine05.host
	...	...	...
table2	Row-AB	24	machine10.host
table2	Row-KL	25	machine13.host
	...	...	...



# Main Point

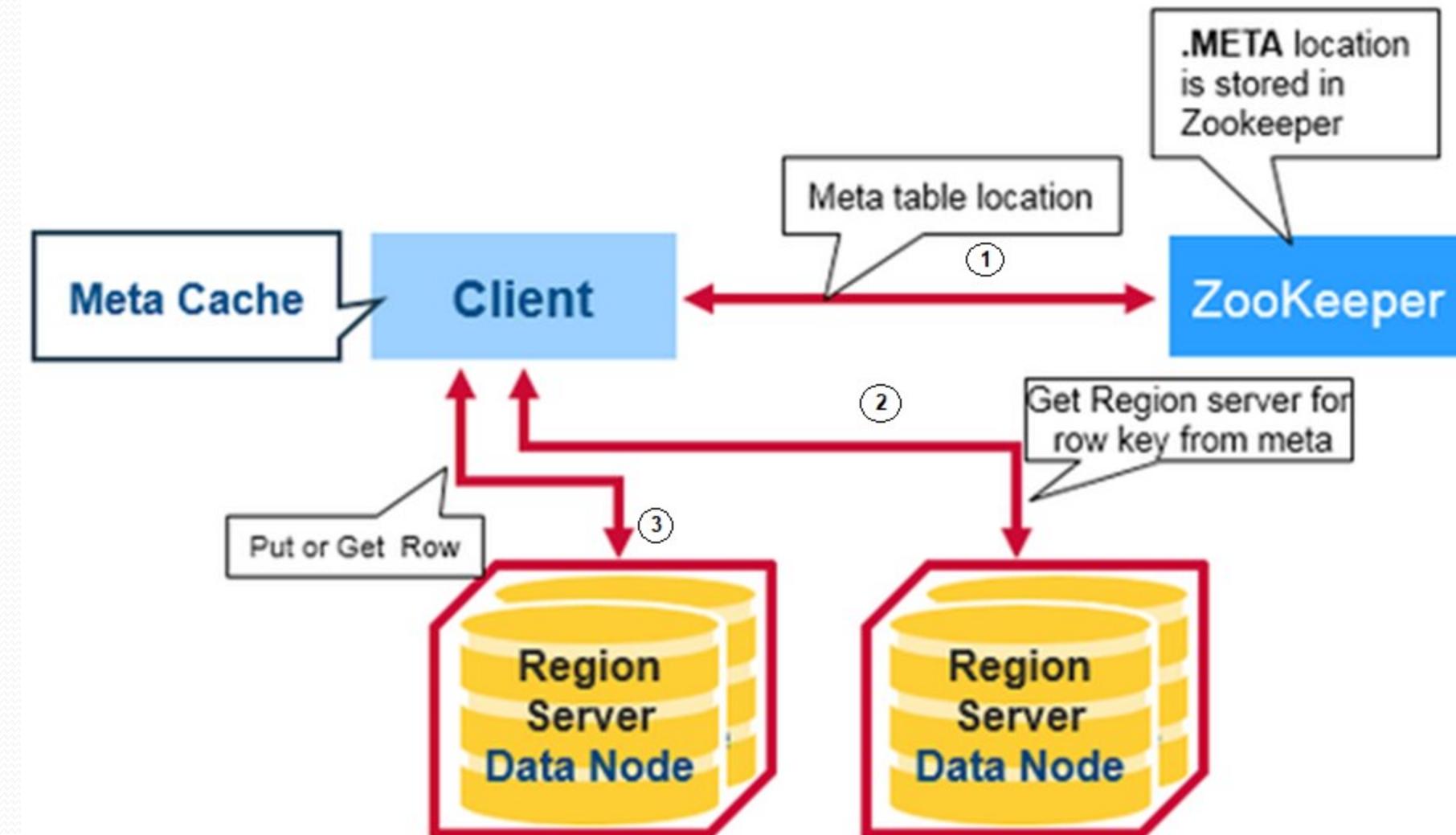
There are 5 major components of HBase: Zookeeper, HMaster, RegionServer, Regions and Meta table. But to get a complete knowledge of HBase, not only these 5 components but the others like MetaStore, BlockCache, WAL, HFiles and the ultimate underlying HDFS needs to be properly understood.

**Science & Technology of Consciousness:** The level of life that is open to the 5 senses of perception is merely the most obvious and surface level of creation. Most of the layers of existence are not available to sensory perception—the cellular, the molecular, the atomic, sub-atomic, quantum force fields, and ultimately the Unified Field of all the laws of nature are all part of life's inherent holistic structure. In order to know and live life fully, the entire range of creation—from gross, to subtle, to subtlest, to life's ultimate, unified transcendental source—must be open to one's awareness.

# Region Lookup – Read/Write Path

- Whenever a client wants to read/write data related to a particular row key, it's a 2-step process.
  1. How to find the region where the target row is?
    - This information is with the hbase:meta Table.
  2. How to find where the hbase:meta table is?
    - This information can be obtained from ZooKeeper.
- Once Client gets the region info, it caches it along with the META table location.
- For future reads/writes, the client uses the cache to retrieve the region info. Over time, it does not need to query the META table again, unless there is a miss because a region has moved; then it will re-query and update the cache.

# Locating RS for Read/Write Path



# Use of BlockCache in Read Path

- BlockCache is a LRU priority cache for data reading.
  - BlockCache is instantiated at region server startup and is retained for the entire lifetime of the process.
  - The BlockCache keeps data blocks resident in memory after they're read.
  - Servicing reads from the BlockCache is the primary mechanism through which HBase is able to serve random reads with millisecond latency.
  - When a data block is read from HDFS, it is cached in the BlockCache. Subsequent reads of neighbouring data – data from the same block – do not suffer the I/O penalty of again retrieving that data from disk.
  - There is a single BlockCache instance in a region server, which means all data from all regions hosted by that server share the same cache pool.



# Write Path

- The **write path** is how an HBase completes **put** or **delete** operations.
- HBase data is organized like a sorted map, with the sorted key space partitioned into different regions.
- An HBase client updates a table by invoking *put* or *delete* commands. When a client requests a change, that request is routed to a region server right away by default.
- Since the row key is sorted, it is easy to determine which region server manages which key.
- So the write path begins at a client, moves to a region server, and ends when data eventually is written to an HBase data file called an **HFile**.

# Write Path contd...

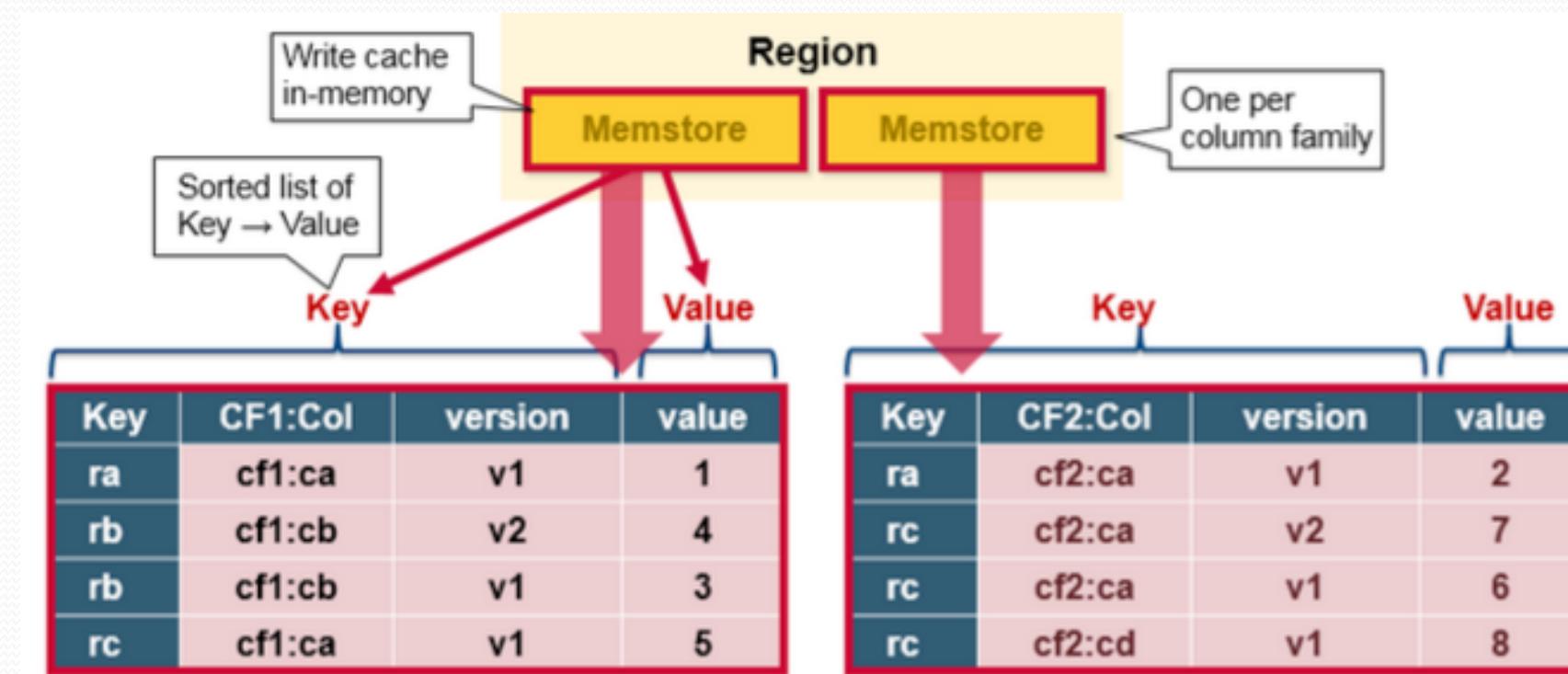
- A change request is for a specific row. Each row key belongs to a specific region which is served by a region server.
- So based on the put or delete's key, an HBase client can locate a proper region server using the 2-step process discussed earlier.
- The region location is cached to avoid this expensive series of operations.
  - If the cached location is invalid (for example, we get some unknown region exception), it's time to re-locate the region and update the cache.
- After the request is received by the right region server, the change cannot be written to HFile immediately because the data in HFile must be sorted by the row key which allows for efficient searching of random rows when reading the data.

# Write Path contd...

- As data cannot be randomly inserted into the HFile, the change must be written to a new file.
- If each update were written to a new HFile, many small HFiles would be created. Such a solution would not be scalable nor efficient to merge or read at a later time. Therefore, changes are not immediately written to a new HFile!
- Instead, each change is stored in a place in memory called the **MemStore**, which cheaply and efficiently supports random writes.
- Data in the MemStore is sorted in the same manner as data in HFile.
- When the MemStore accumulates enough data, the entire sorted set is written to a new HFile in HDFS. Completing one large write task is efficient and takes advantage to HDFS's strengths.

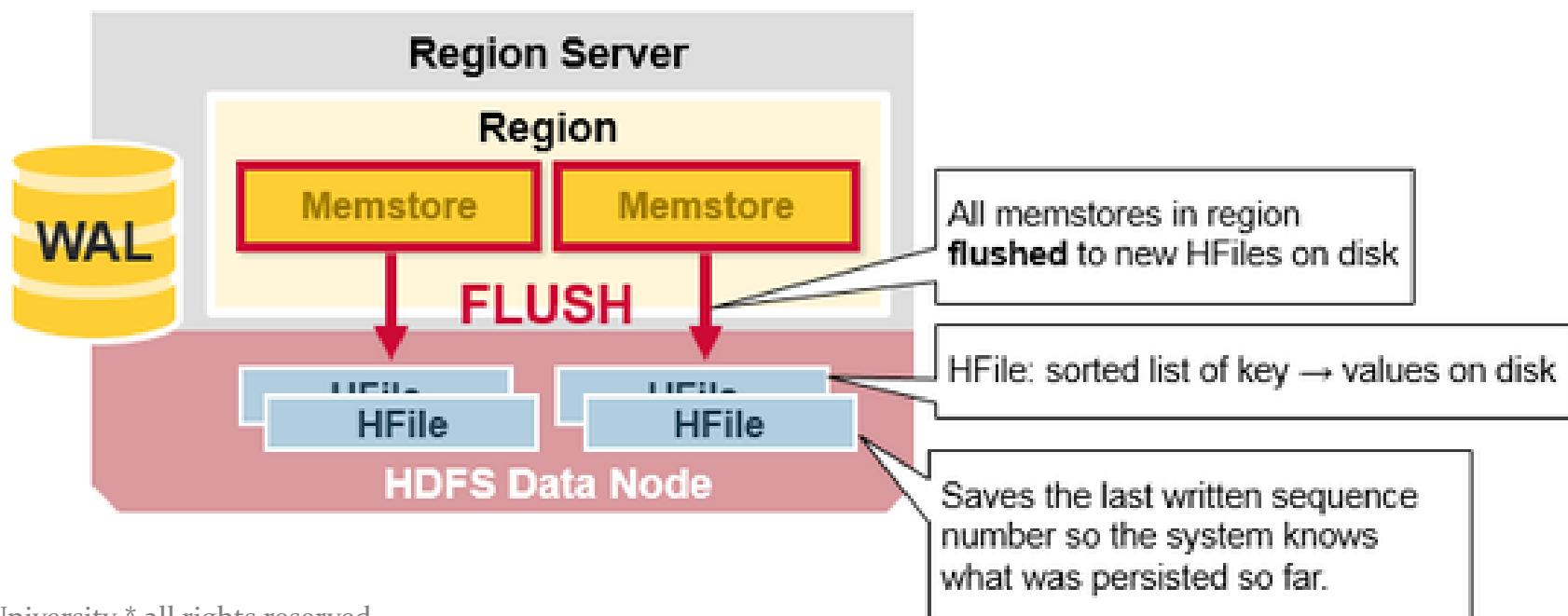
# MemStore

- MemStore is “write” cache – sorted list of key->value.
- When something is written to HBase, it is first written to this in-memory store (*MemStore*) and so the MemStore is important for accessing recent edits.



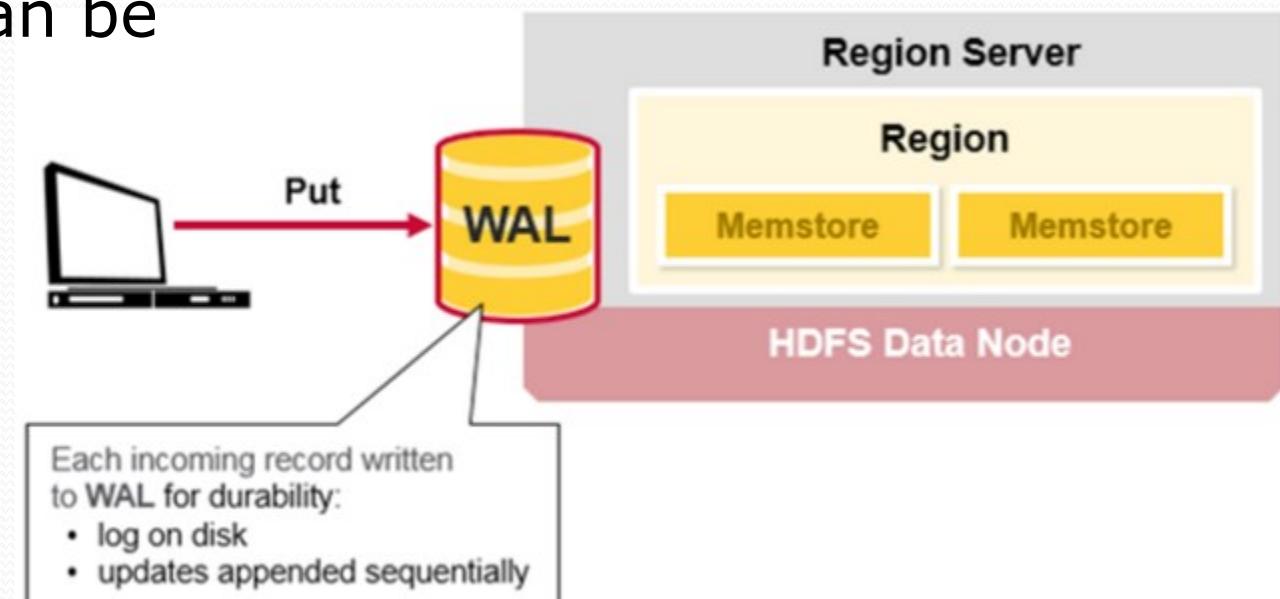
# MemStore Flush

- Once the MemStore reaches a certain size, it is flushed to disk into a new *store file* (the entire sorted set is written to a new HFile in HDFS).
- HBase uses multiple HFiles (StoreFiles) per column family, which contain the actual cells, or KeyValue instances. These files are created over time as KeyValue edits stored in the MemStore are flushed as files to disk.



# Use of WAL(HLog) in Write Path

- Although writing data to the MemStore is efficient, it also introduces an element of risk: Information stored in MemStore is stored in volatile memory, so if the system fails, all MemStore information is lost.
- To help mitigate this risk, HBase saves updates in a write-ahead-log (**WAL**) before writing the information to MemStore.
- In this way, if a region server fails, information that was stored in that server's MemStore can be recovered from its WAL (HLog).
- HDFS replicates WAL automatically.

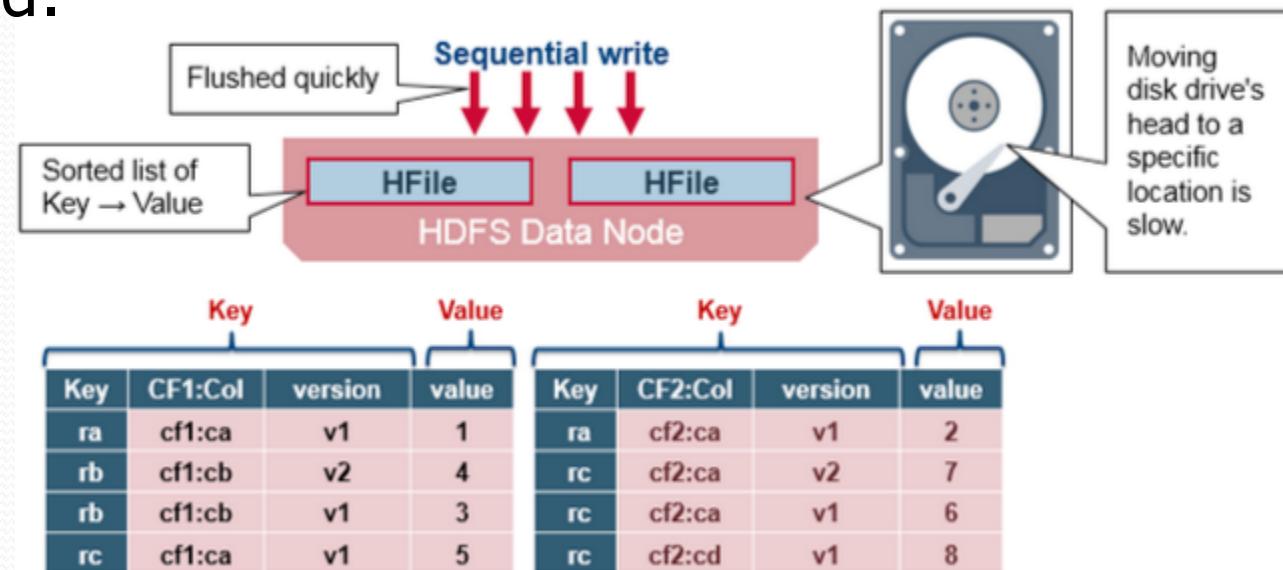


# Use of WAL(HLog) in Write Path contd...

- Because WAL files are ordered chronologically, there is never a need to write to a random place within the file.
- As WALs grow, they are eventually closed and a new, active WAL file is created to accept additional edits. This is called “rolling” the WAL file. Once a WAL file is rolled, no additional changes are made to the old file.
- Note: By default, WAL is enabled, but the process of writing the WAL file to disk does consume some resources. WAL may be disabled, but this should only be done if the risk of data loss is not a concern. If you choose to disable WAL, consider implementing your own disaster recovery solution or be prepared for the possibility of data loss.

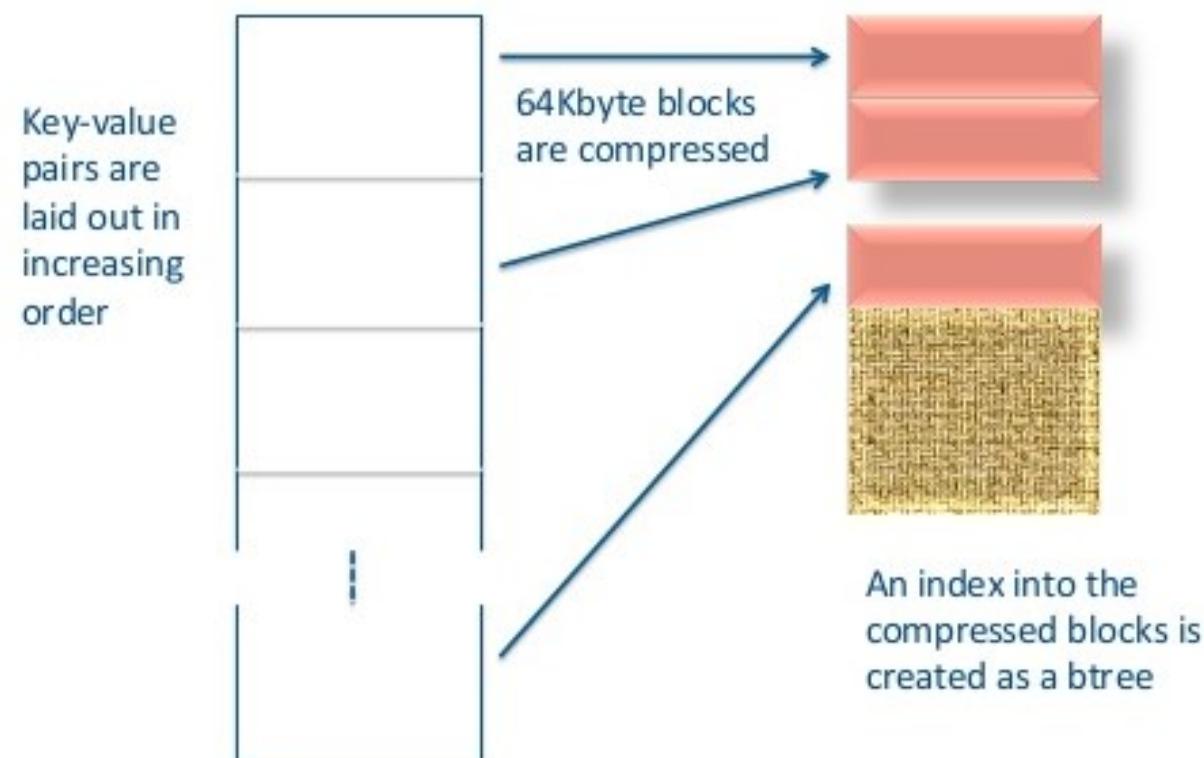
# HFile

- HBase stores its data in HDFS in a format called HFile.
- HDFS replicates HFiles automatically.
- HFiles store the rows as sorted Key-Values on disk and according to this order, data is stored and split across the nodes.
- When the MemStore accumulates data more than its limit, the entire sorted 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.

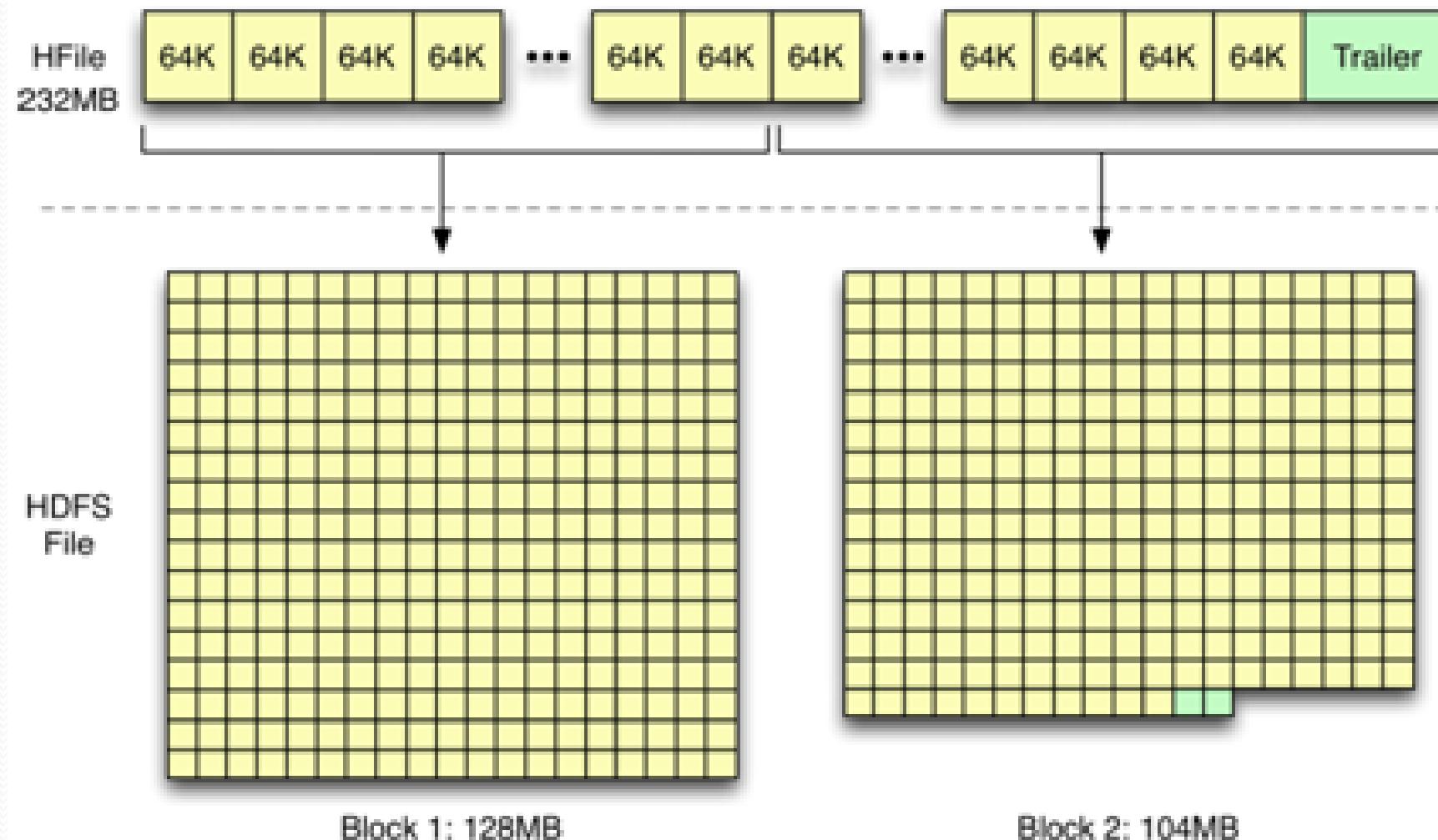


# HFile Structure

- 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.
- Each cell is an individual key + value

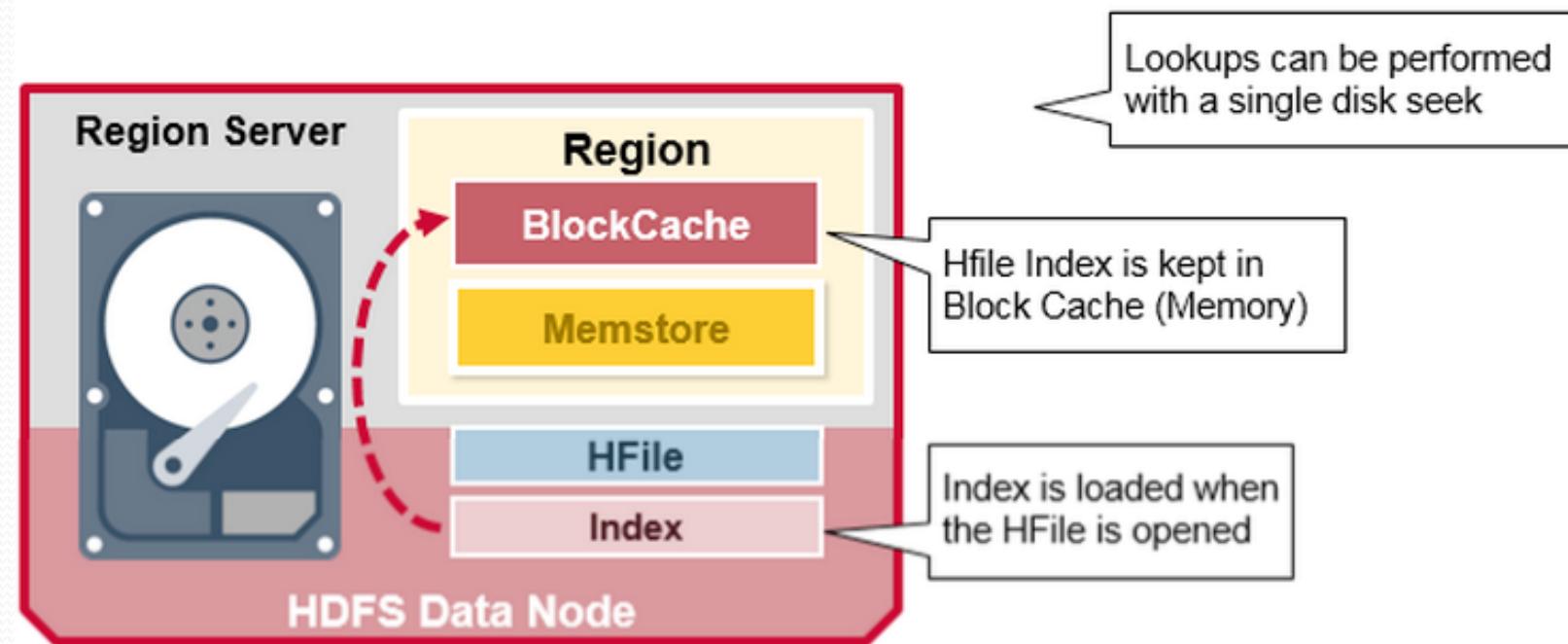


# HFile Structure contd..



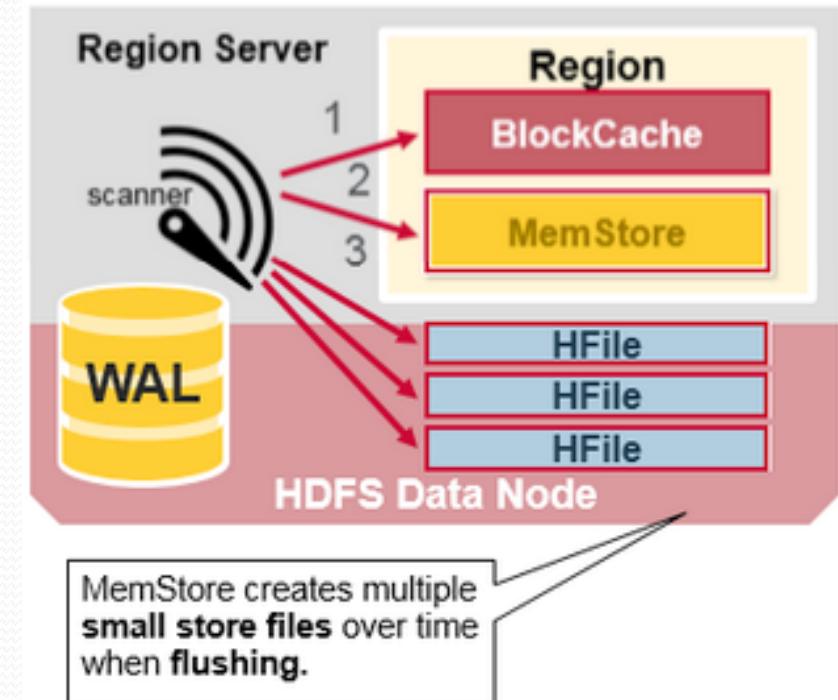
# HFile contd...

- No two column families can exist in single HFile.
- The index in HFile is loaded when the HFile is opened and is kept in memory (BlockCache). This allows lookups to be performed with a single disk seek.
- The highest sequence number is stored as a meta field in each HFile, to better state where it has ended previously and where to continue next.



# HBase Read Merge

- KeyValue cells corresponding to one row can be in multiple places :- **row** cells that are already persisted are in HFiles, recently updated cells are in MemStore, and recently read cells are in BlockCache.
- So when you read a row, how does the system get the corresponding cells to return?



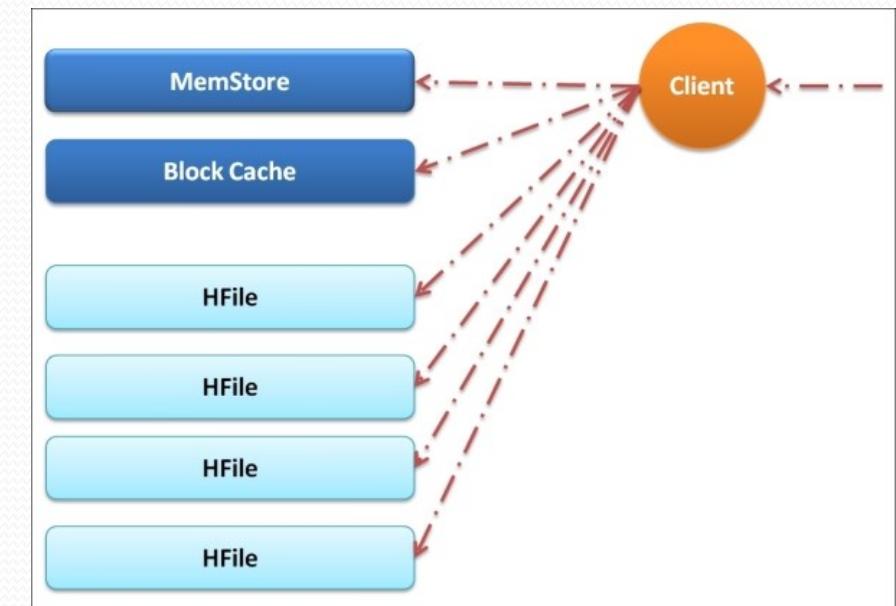
# HBase Read Merge contd...

- A *Read* merges Key Values from the BlockCache, MemStore, and HFiles in the following steps:
  - Scanner looks for the Row cells in the BlockCache and MemStore.
  - The latest value is returned.
  - If the scanner does not find all of the row cells in the MemStore and BlockCache, then HBase will use the BlockCache indexes and bloom filters to load HFiles into memory, which may contain the target row cells.

HBase Bloom Filter is a space-efficient mechanism to test whether an HFile contains a specific row or row-col cell.

# Read Amplification

- There may be many HFiles per MemStore, which means for a read operation, multiple Hfiles may need to be examined, which can affect the performance.
- **This is called read amplification.**
- To avoid this, there is a need to merge these smaller HFiles into bigger ones and reduce the number of Hfiles per CF.
- Also, in HBase if we try to update the column values, it does not overwrite the existing value but rather stores different values per row by time (and qualifier) – excess versions!
- These excess versions mean excess store files!

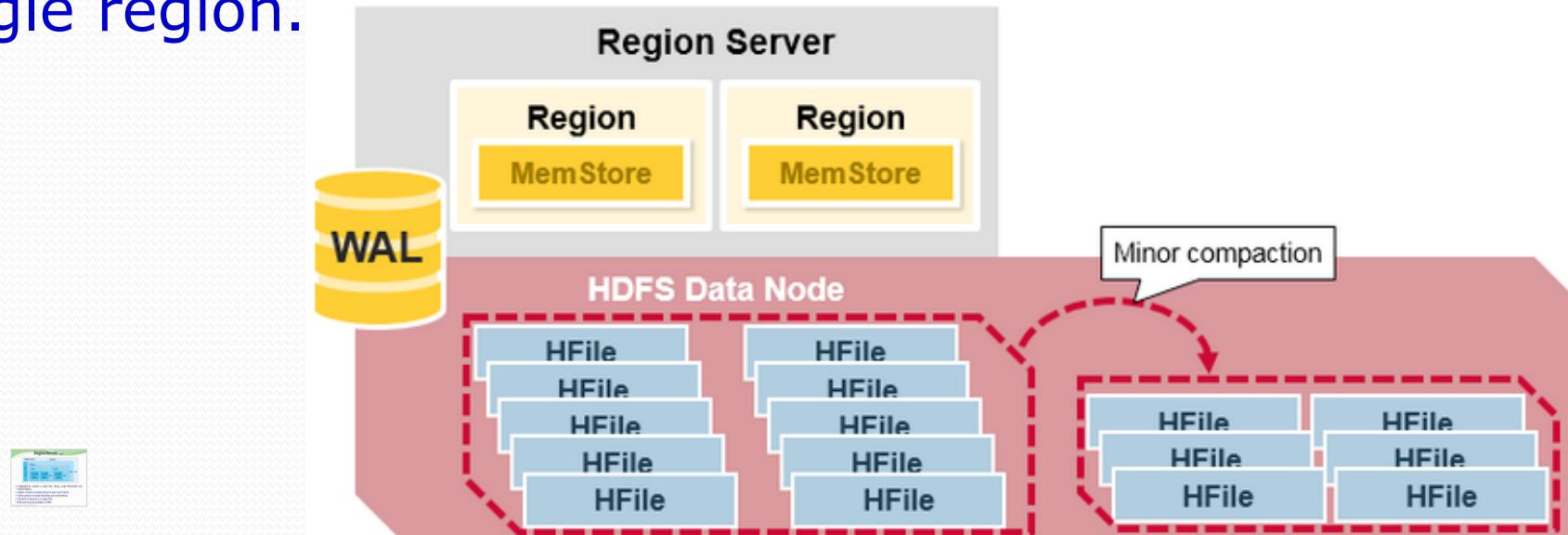


# Compactions

- Excess store files or HFiles are merged together by a process called *compaction*.
- **Two types of compactions:**
  - **Minor compactions**
  - **Major compactions**

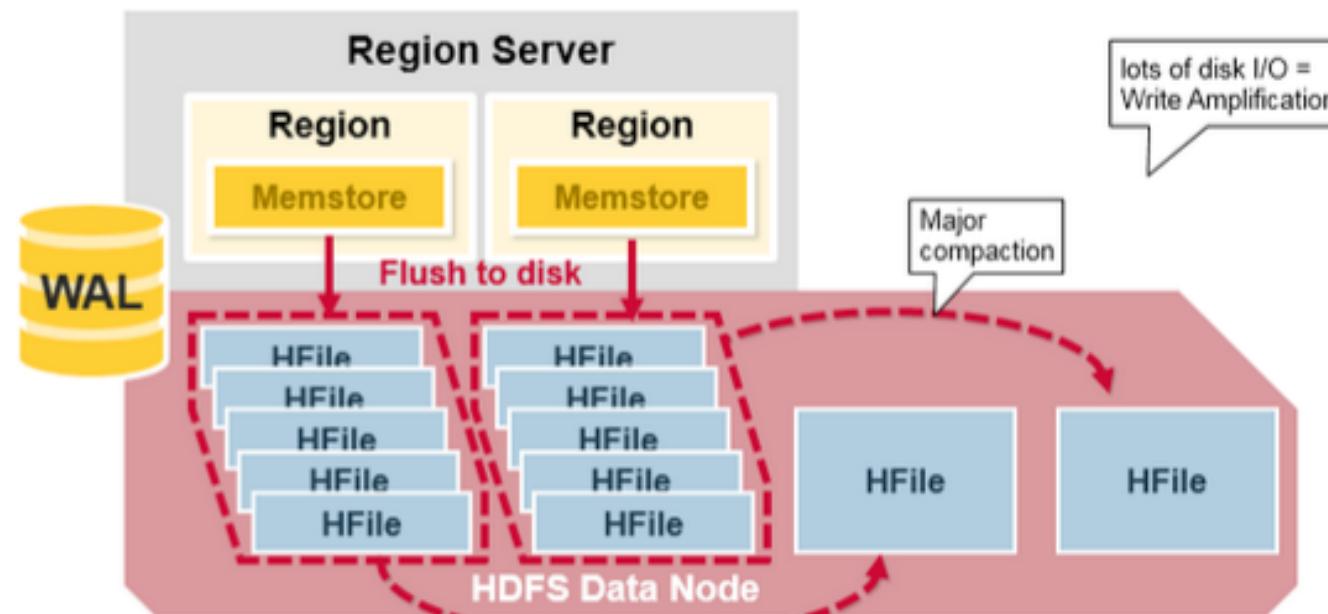
# Minor Compactions

- 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 store files by rewriting smaller files into fewer but larger ones, performing a merge sort.
- These are triggered each time a MemStore is flushed, and will merge the HFiles.
- **Minor compaction merges store files of a single store inside a single region.**



# Major Compactions

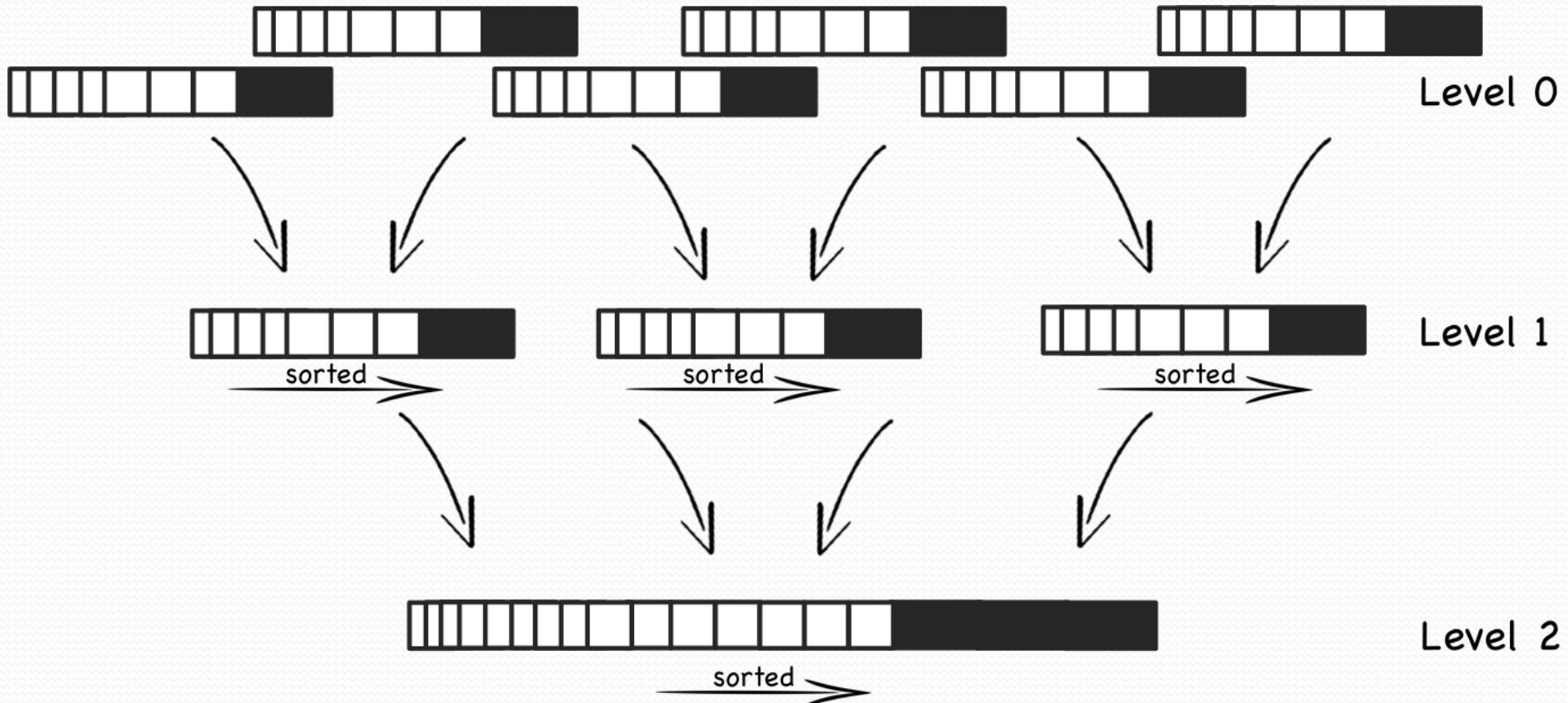
- Major compaction merges and rewrites all the HFiles in different regions to one HFile per CF, and in the process, drops deleted or expired cells.
- This improves read performance; however, since major compaction rewrites all of the files, lots of disk I/O and network traffic might occur during the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



# Major Compactions contd...

- Major compactions can be scheduled to run automatically. Due to write amplification, major compactions are usually scheduled for weekends or evenings.
- They can also be triggered manually, via the API or the shell.
- Major compactions process delete markers, max versions, etc., while minor compactions don't. This is because delete markers might also affect data in the non-merged files, so it is only possible to do this when merging all files.
- Major compaction also makes any data files that were remote, due to server failure or load balancing, local to the region server.

# Compactions



Compaction continues creating fewer, larger and larger files

# Main Point

Compaction chooses the HFiles from HRegions and combines them so that there'll be only one HFile per column family. Compaction reduces read amplification but Major compaction adds write amplification and so it should be done when the cluster is not in heavy use. Doing compactations regularly increases the efficiency of HBase.

**Science & Technology of Consciousness:** Nature functions in the most efficient way. Action in enlightenment expresses this law.

# Hbase Table Design Issues

- More complex than relational table design as table design is driven by data access patterns.
  - HBase tables should not be designed to mimic RDBMS tables.
- Tables should be structured to return data in an efficient manner.

## **Important considerations:**

- What values should compose the key?
- What column families are needed? (specified at table creation)
- What columns are associated with each column family?

# HBase Use Cases

- **2 well-known use cases**
  - Lots and lots of data (billions of rows and columns)
  - Large amount of clients/requests
- **Great for single random selects and range scans by key**
- **Great for variable schema - Storing data by row that doesn't conform well to a schema**
  - Rows may drastically differ
  - If your schema has many columns & most of them are null

# HBase Use Cases

- HBase is in use at thousands of organizations, including Adobe, Airbnb, Alibaba, Bloomberg, Flipkart, Huawei, HP, Hubspot, IBM, Microsoft, NetEase, Pinterest, Salesforce, Shopee, Tencent, Twitter, Xiaomi, and Yahoo! (now Verizon Media), among others.
- Some of the important use cases are:
  - Used by Mozilla: They generally store all crash data in HBase
  - Used by Facebook: Facebook uses a variant of HBase for real-time messaging.
  - Cloudera: Customer cluster data is in HBase, each customer node is a key with Avro value, easy to browse, quick to find issues on nodes, dump to HDFS and process with Pig
  - Navteq: Location based content serving, location makes a great key in HBase, preprocessing is done via MR jobs

# When not to use HBase?

- HBase is good, but not an RDBMS or HDFS replacement
- Not suitable for every problem
  - Compared to RDBMs has VERY simple and limited API
- When your data is very small (not in TBs)
- When you only append to your dataset and tend to read the whole thing
- HBase has intermittent but large IO access (memory and CPU intensive) - May affect response latency!
- Bad for traditional RDBMS retrieval
  - Transactional applications
  - Relational Analytics
    - 'group by', 'join', and 'where column like', etc....

# HBase Interaction

- We can interact with HBase in two ways:

## **1. HBase interactive shell mode**

- HBase shell is made up of JRuby  
(JRuby is Java implementation of Ruby)
- It is used to interact with HBase for table operations, table management, and data modelling.

## **2. Through HBase Java API**

- By using Java API model, we can perform all types of table and data operations in HBase

# HBase Shell Commands

- Starting HBase shell – “**hbase shell**”

- **LIST**

List all tables in HBase. Optional regular expression parameter could be used to filter the output.

```
hbase> list
```

```
hbase> list 'abc.*'
```

- **DESCRIBE**

Describe the named table.

```
hbase> describe 't1'
```

# Hbase Shell Commands

- **CREATE**

create table; pass table name, a dictionary of specifications per column family, and optionally a dictionary of table configuration.

```
hbase> create 't1', {NAME => 'f1', VERSIONS => 5}
```

```
hbase> create 't1', {NAME => 'f1'}, {NAME => 'f2'}
```

# The above in shorthand would be similar to:

```
hbase> create 't1', 'f1', 'f2', 'f3'
```

```
hbase> create 't1', {NAME => 'f1', VERSIONS => 1, TTL => 2592000, BLOCKCACHE => true}
```

```
hbase> create 't1', {NAME => 'f1', CONFIGURATION => {'hbase.hstore.blockingStoreFiles' => '10'}}}
```

- Table configuration options can be put at the end.

# Hbase Shell Commands

- **ALTER**

```
hbase> alter 't1', NAME => 'f1', VERSIONS => 5
```

```
hbase> alter 't1', NAME => 'f1', MIN_VERSIONS => 2
```

```
hbase> alter 't1', NAME => 'f1', TTL => 15
```

- **PUT**

```
hbase> put 't1', 'row1', 'f1:a', 'value1'
```

```
hbase> put 't1', 'row2', 'f2:b', 'value2'
```

```
hbase> put 't1', 'row3', 'f3:c', 'value3'
```

- **SCAN**

```
hbase> scan 't1'
```

# Hbase Shell Commands

- **DISABLE**

Start disable of named table.

```
hbase> disable 't1'
```

- **DROP**

Drop the named table. Table must first be disabled.

```
hbase> drop 't1'
```

- **ENABLE**

Start enable of named table.

```
hbase> enable 't1'
```