CS523 - BDT
Big Data Technologies

NoSQL DB - Apache HBase

(Satisfying the need for real time read/write/update)

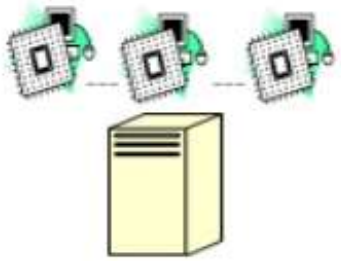
Need for a Database

- Data storage comes in different forms that have different features.
- Data storage as files has a feature of being fast to scan and is therefore well suited for batch processing.
- It is difficult, however, to find a particular record in large input files or to find a number of records that pertain to a single individual.
- Furthermore, updates to files can be difficult to coordinate.
- These actions are much better supported by some sort of database but at the significant cost in "scanning performance" relative to flat files.

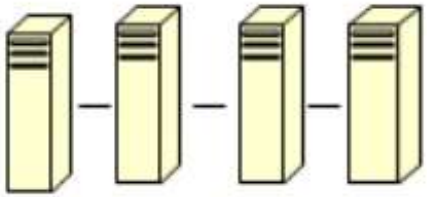
Need for NoSQL Databases

- In RDBMS, the structure of the data should be known beforehand. Changing structure and adding new data would leave lot of NULLs in your table which is not good.
- In RDBMS, a lot of resources are spent on referential integrity and normalization (not a disadvantage but still takes up lot of resources)
- RDBMS is not able to sustain and maintain the increasing volume of data with ACID properties.
- Typically RDBMS are built for single server, so they are good up to a few TBs of data but they cannot handle say 500TB of data.
- A new type of data is increasingly becoming large that doesn't necessarily 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 to storing and retrieving high volumes of data with horizontal scaling, simple design, high flexibility and high availability.

Non-RDBMS way of Thinking



Scale up



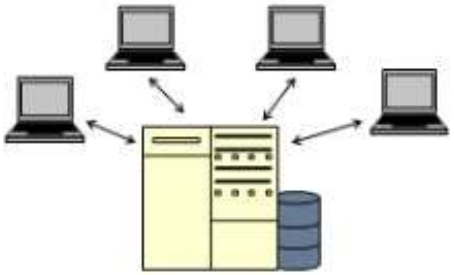
Scale out

Userid	Fname	Blog
User1	Chhavi	www.bigdata.impetus.com
User2	John	www.impetus.co.in
User3	Larry	www.impetus.com/webinar

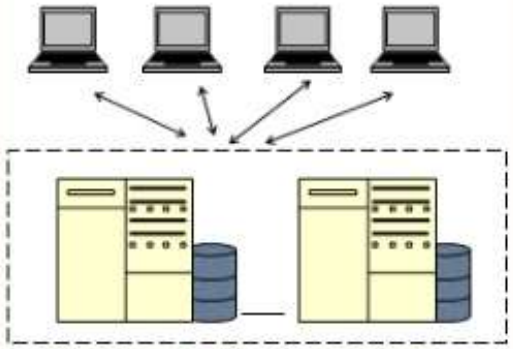
Static schema

User1	Chhavi		
User3	Larry	Pearson	www.impetus.co.in/webinar
User2	John	www.impetus.co.in	

Dynamic schema



Centralized



Decentralized

Comparison of NoSQL with RDBMS

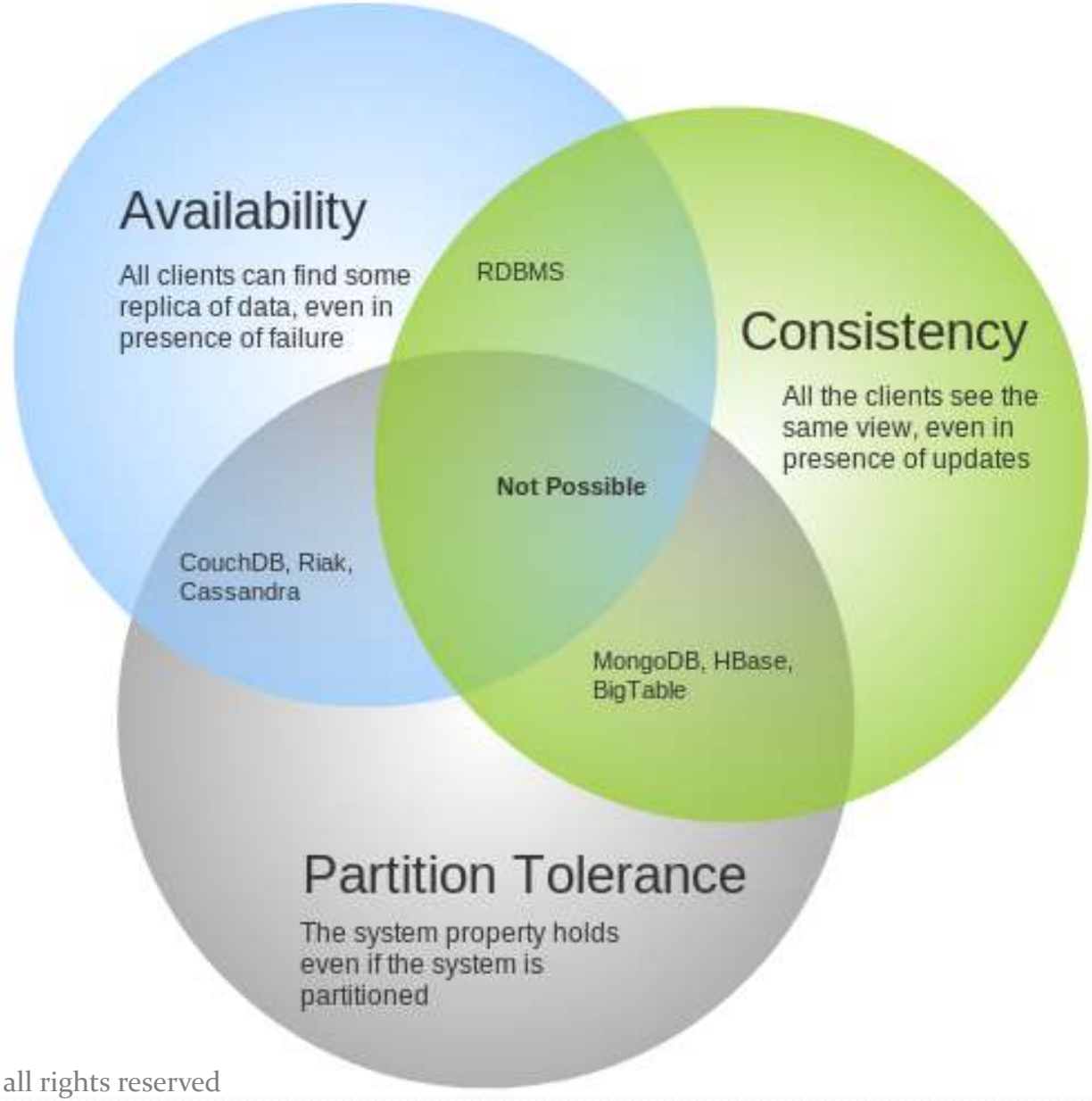
- NoSQL is a family of databases which do not conform to the SQL standard of data organization and is a revolutionary form of data storage in modern web applications.
- Whereas SQL databases often have very rigid, inflexible data models made up of columns and tables, linked together through foreign keys, 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 to the last.

Comparison of NoSQL with RDBMS

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

CAP Theorem

- The CAP theorem states that any networked shared-data system can have at most two of three desirable properties:
- **Consistency (C)** equivalent to having a single up-to-date copy of the data;
- **High Availability (A)** of that data (for updates); and
- **Partition Tolerance** to network partitions (P).



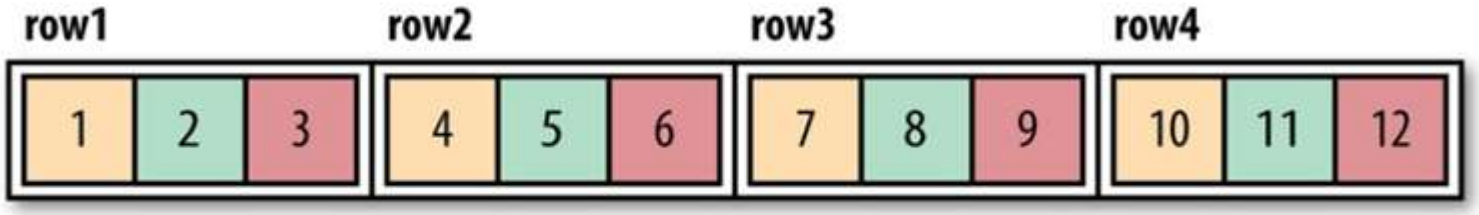
Types of NoSQL Databases

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

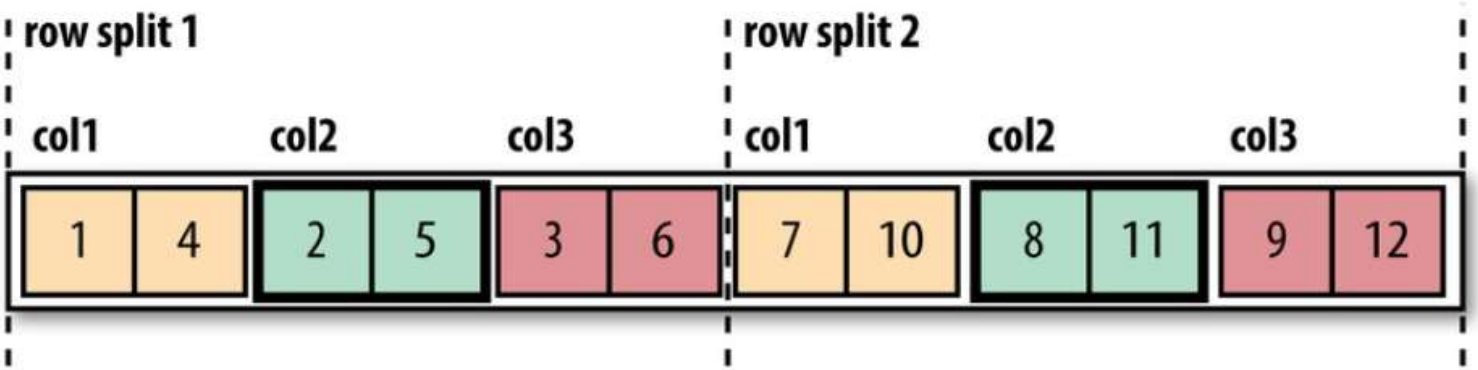
Logical table

	col1	col2	col3
row1	1	2	3
row2	4	5	6
row3	7	8	9
row4	10	11	12

Row-oriented layout

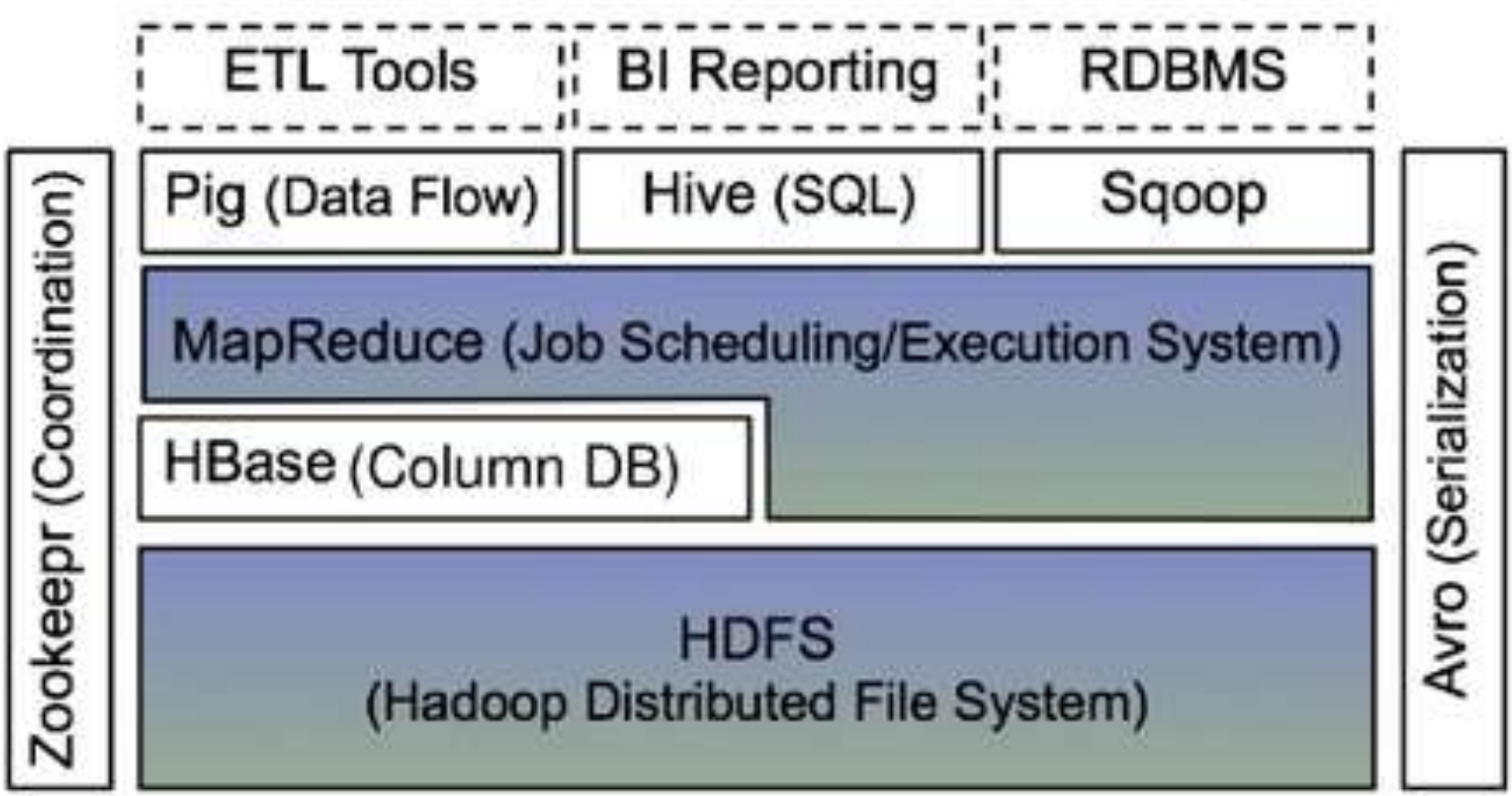


Column-oriented layout



- **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 is a distributed, NoSQL column-oriented database which runs on top of HDFS and is a good choice when real-time read/write random access to very large datasets is required.
- Based on Google's BigTable paper.
- 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 MR 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.

Backdrop

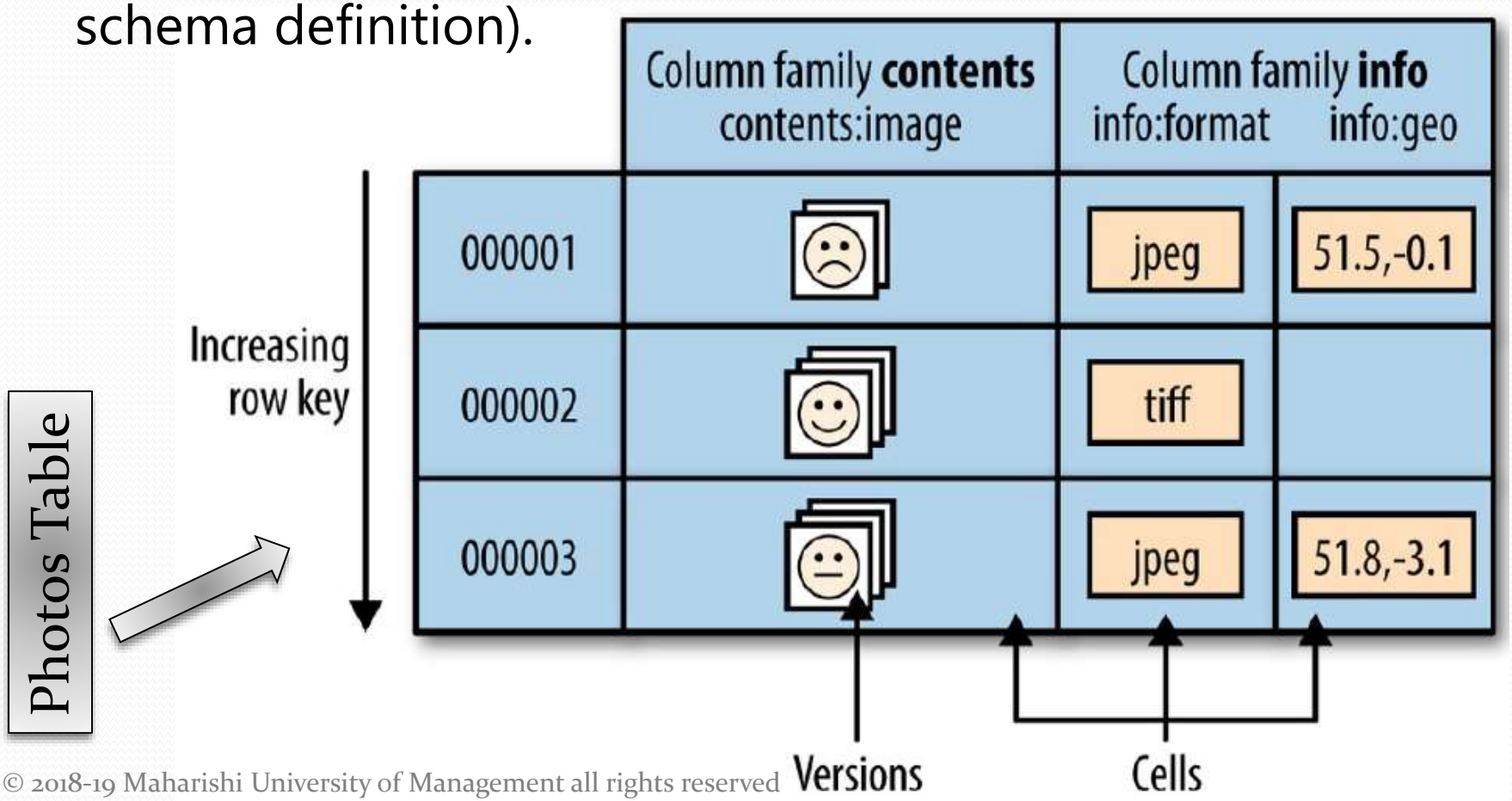
- The HBase project was started towards the end of 2006 by Chad Walters and Jim Kellerman at Powerset.
- It was modelled after Google's Bigtable, which had just been published.
- In February 2007, Mike Cafarella made a code drop of a mostly working system that Jim Kellerman then carried forward.
- The first HBase release was bundled as part of Hadoop 0.15.0 in October 2007.
- In May 2010, HBase graduated from a Hadoop subproject to become an Apache Top Level Project.
- Today, HBase is a mature technology used in production across a wide range of industries.

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 column 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.

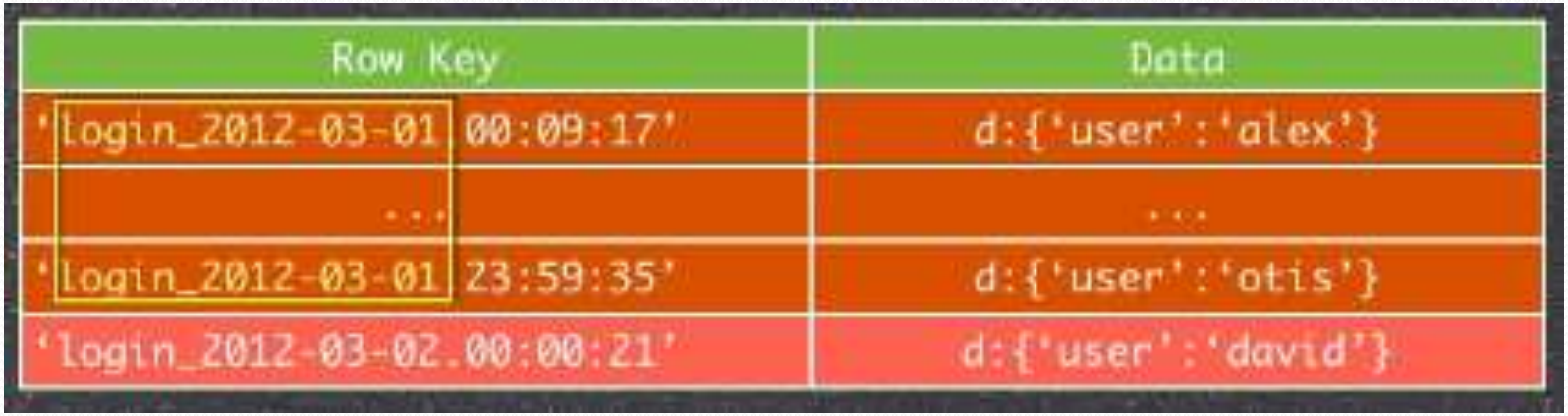
HBase Concepts

- 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).



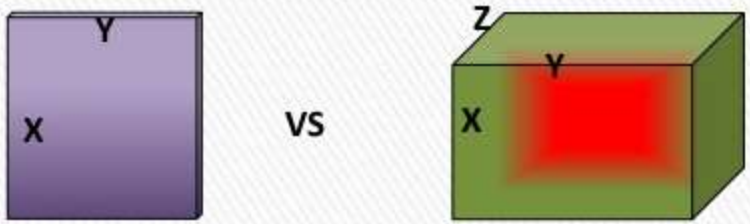
HBase Concepts: Row-Key

- 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 (PK). 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.



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.



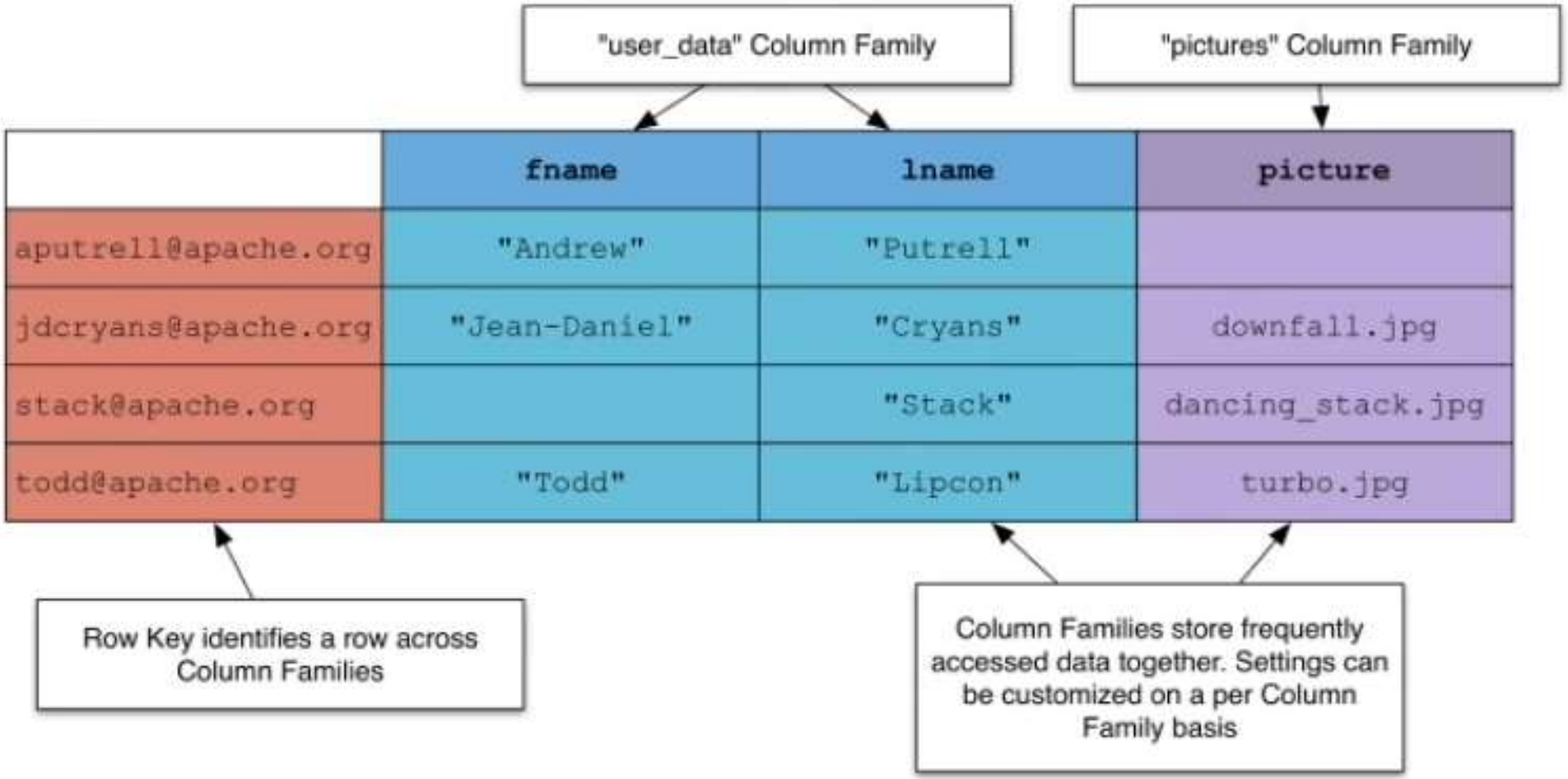
- Each cell in HBase is defined as a key/value pair, and each key to identify a cell is made up of row-key, CF name, column name and time-stamp.
  - So it is advised to keep all these names short
  - And serialize and store many values into single cell

Concepts: Column-Family

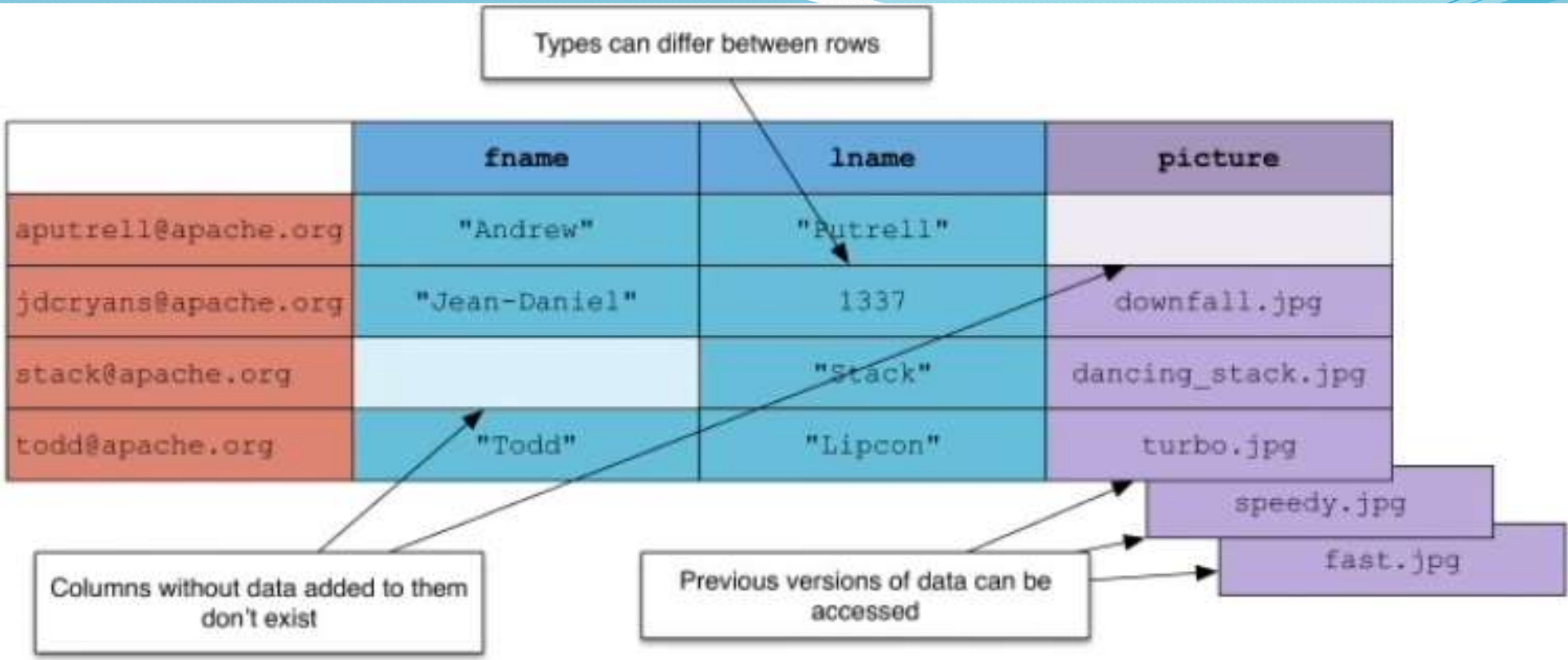
- Row columns are grouped into column families and column families are internally stored in HDFS as files.
- All column family members have a common prefix, so, for example, the columns *info*:*format* and *info*:*geo* are both members of the *info* column family, whereas *contents*:*image* belongs to 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 column family members have the same general access pattern and size characteristics.
  - For the photos table, the image data, which is large (megabytes), is stored in a separate column family from the metadata, which is much smaller in size (kilobytes).
- Separate column families are useful for:
  - Data that is not frequently accessed together
  - Data that uses different column family options
    - E.g. compression



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

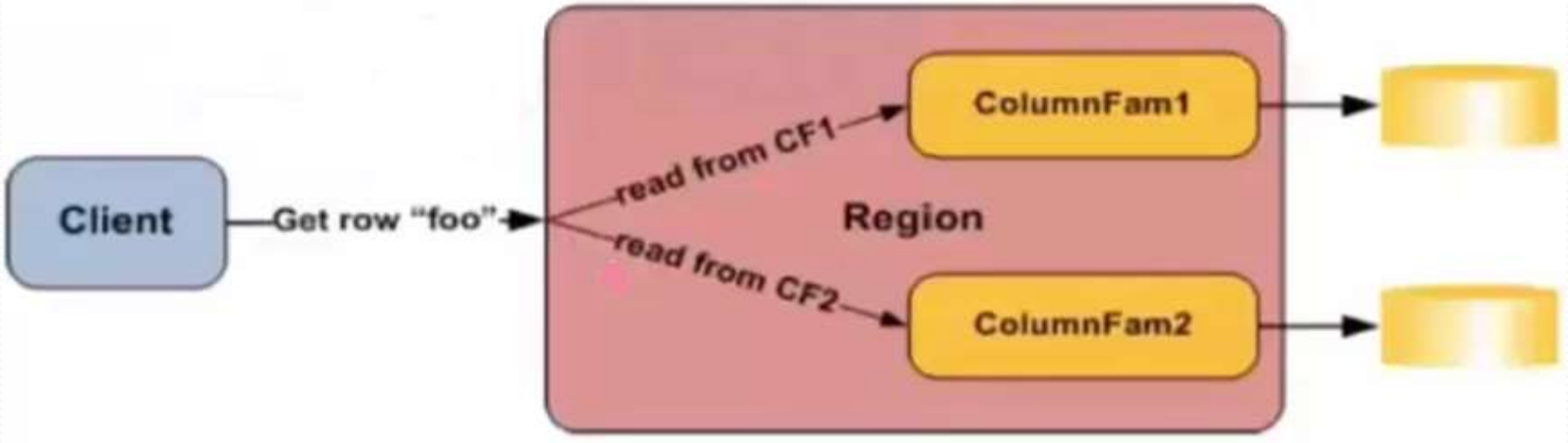


Things to remember:

- 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.

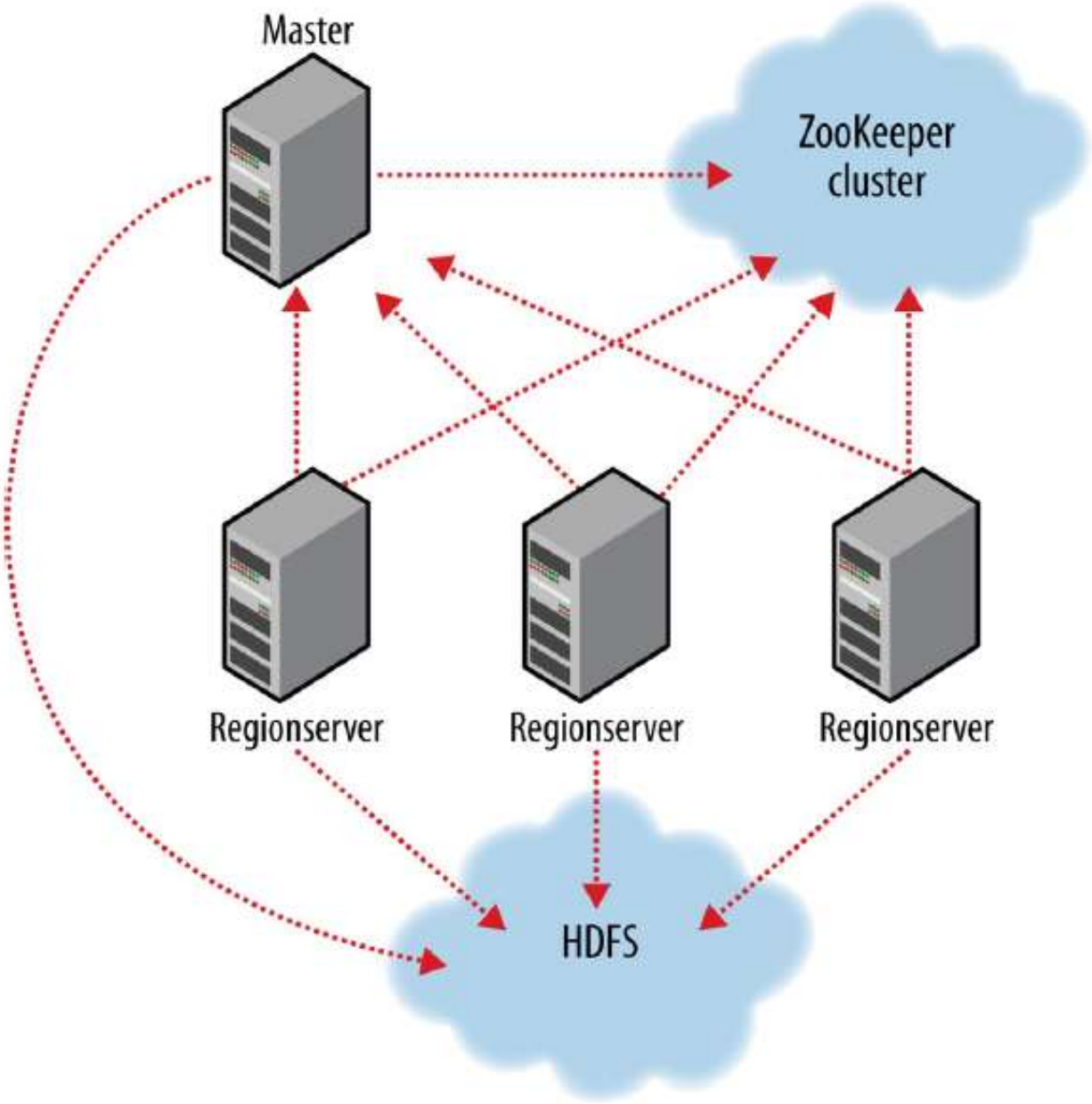
Designing Tables

- **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

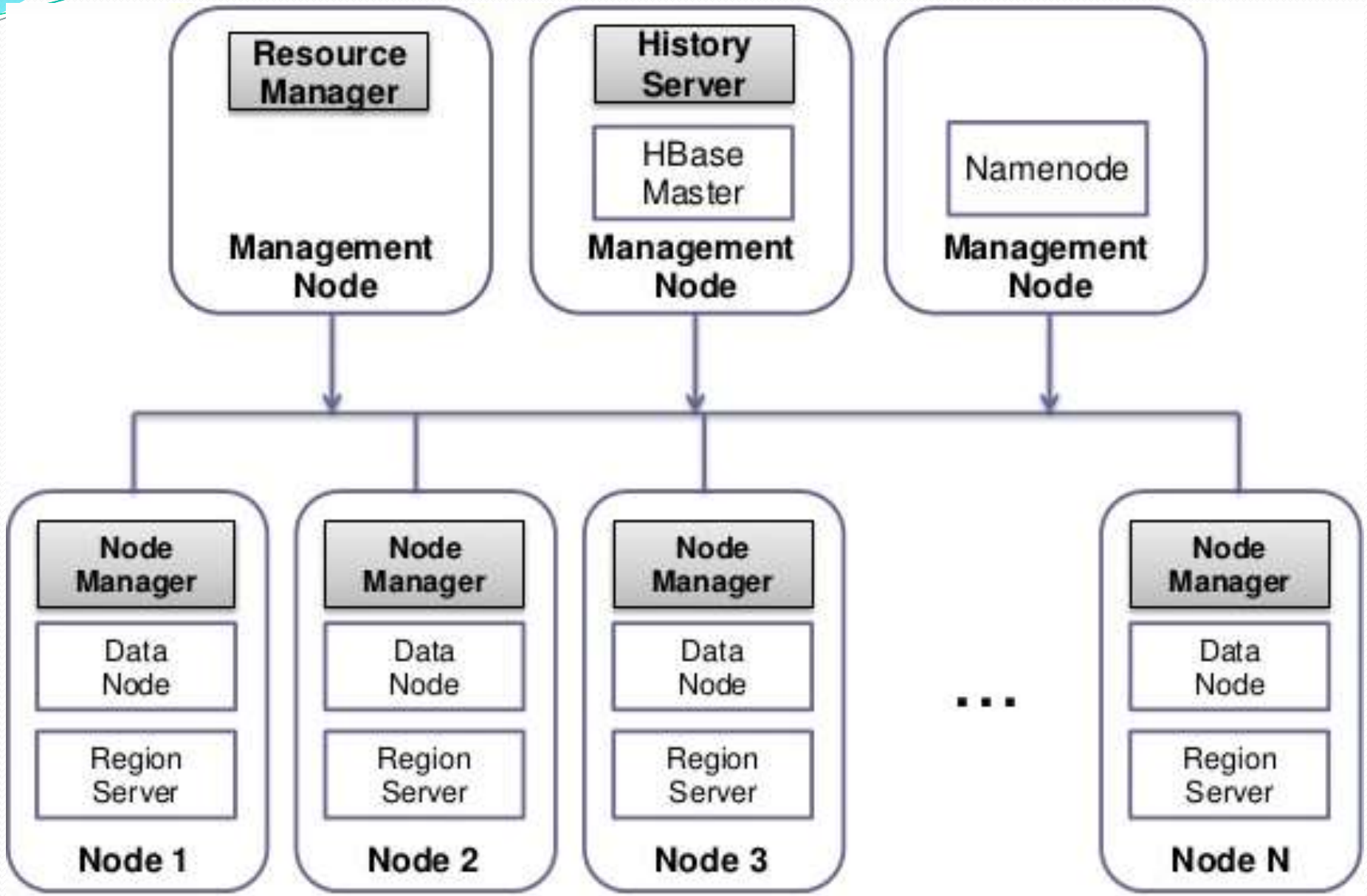


HBase Cluster

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



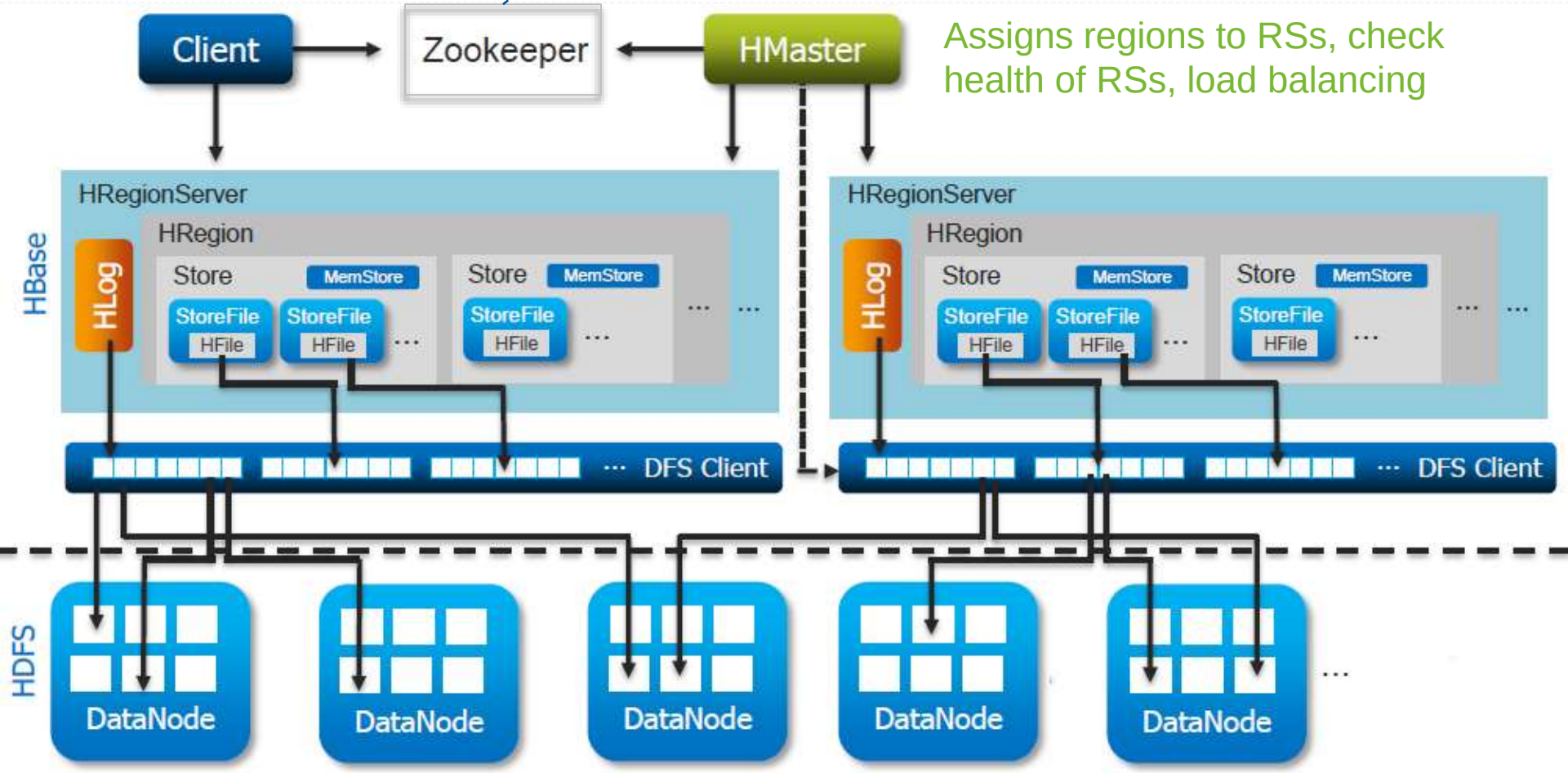
HDFS, YARN, HBase Daemons



HBase Architecture

Client reads/writes rows by accessing RS

Coordination & monitoring; Has HMaster and ROOT location



Major components of HBase

1. Zookeeper
2. HMaster
3. HRegionServer
4. HRegion
5. Catalog tables
   - ROOT
   - META

ZooKeeper

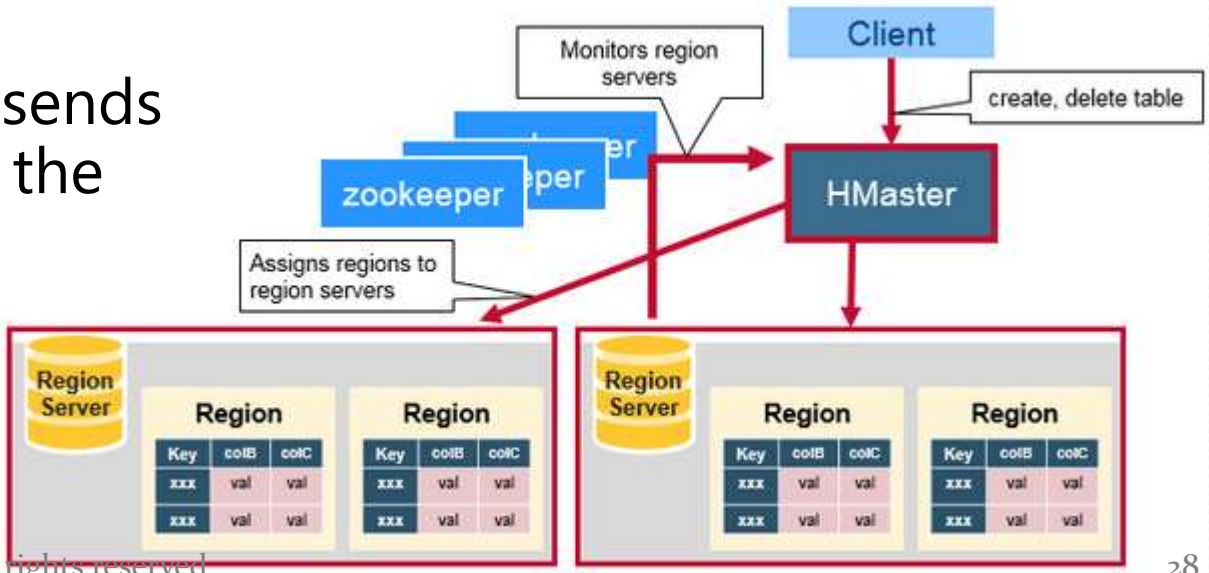
- Zookeeper is used to coordinate shared state information for members of distributed systems. Region servers and the active HMaster connect with a session to ZooKeeper.
- If client wants to communicate with regions, the client has to approach ZooKeeper first.

Services provided by ZooKeeper (centralized monitoring server)

- Maintains Configuration information
- Provides distributed synchronization
- Client Communication establishment with region servers
- To track 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.

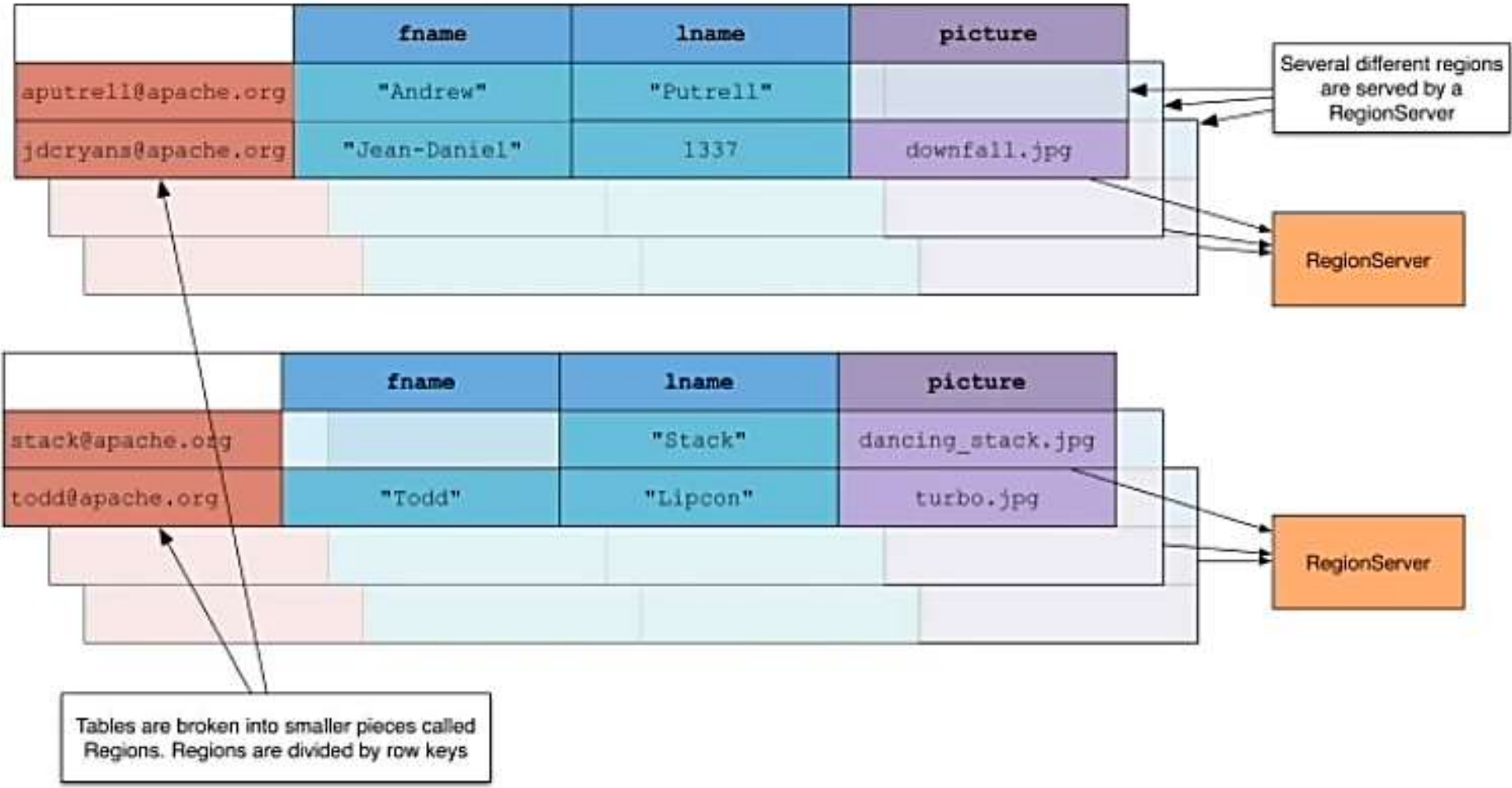
HMaster

- Assigns Regions to RegionServers
- Keeps pointer to –ROOT- table location on a region
- 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 ZK, & the inactive one waits.
  - Master server crash cannot cause data loss.



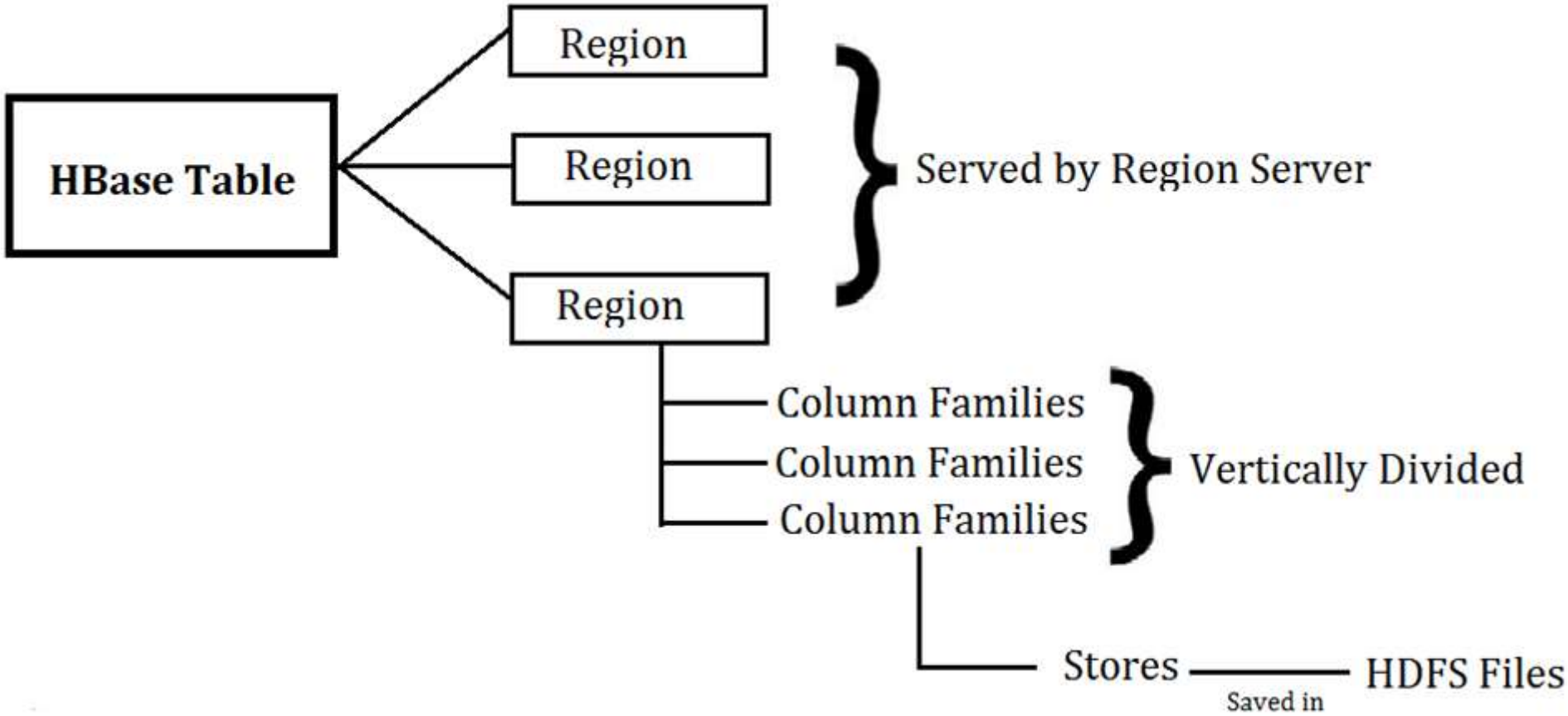
Regions (HRegions)

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



Regions (HRegions) Contd.

- Regions are the units that get distributed over an HBase cluster.
- 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.

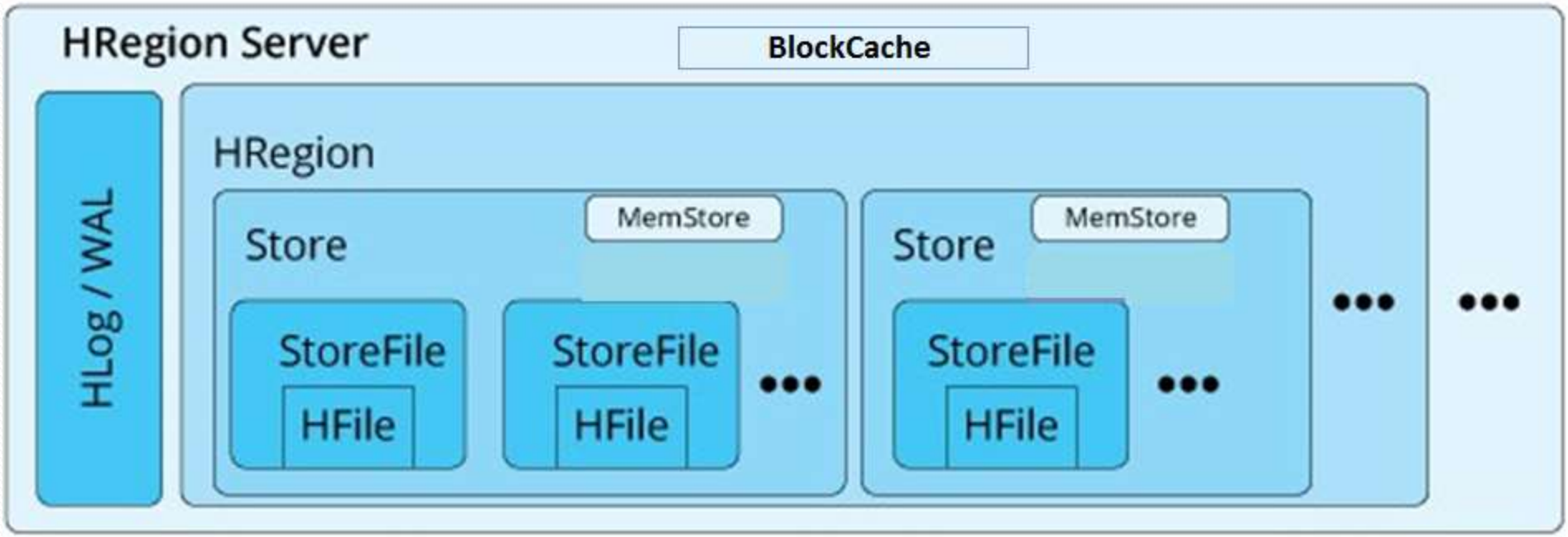


RegionServer

- 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**: 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.

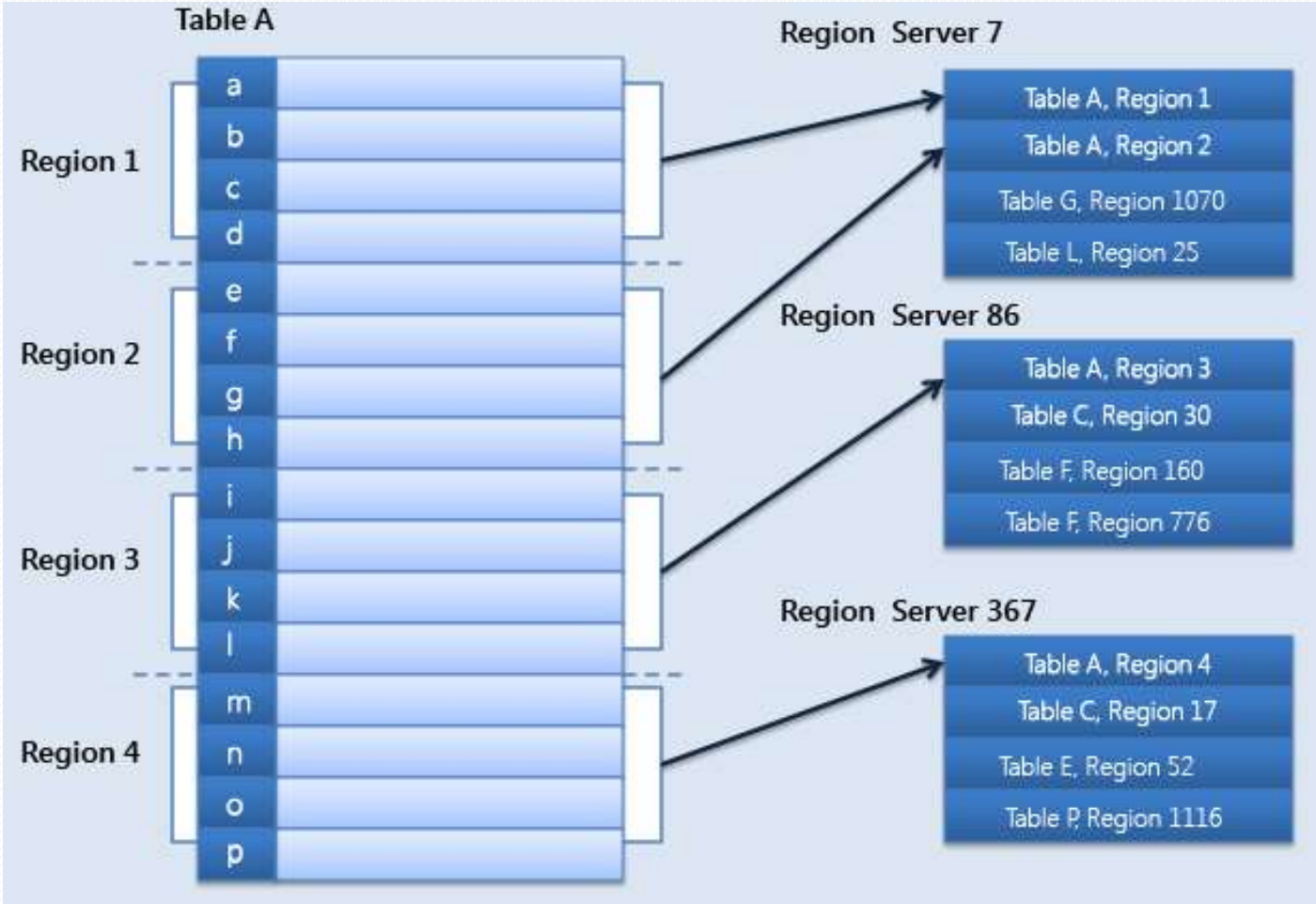
RegionServer Contd.

- 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.



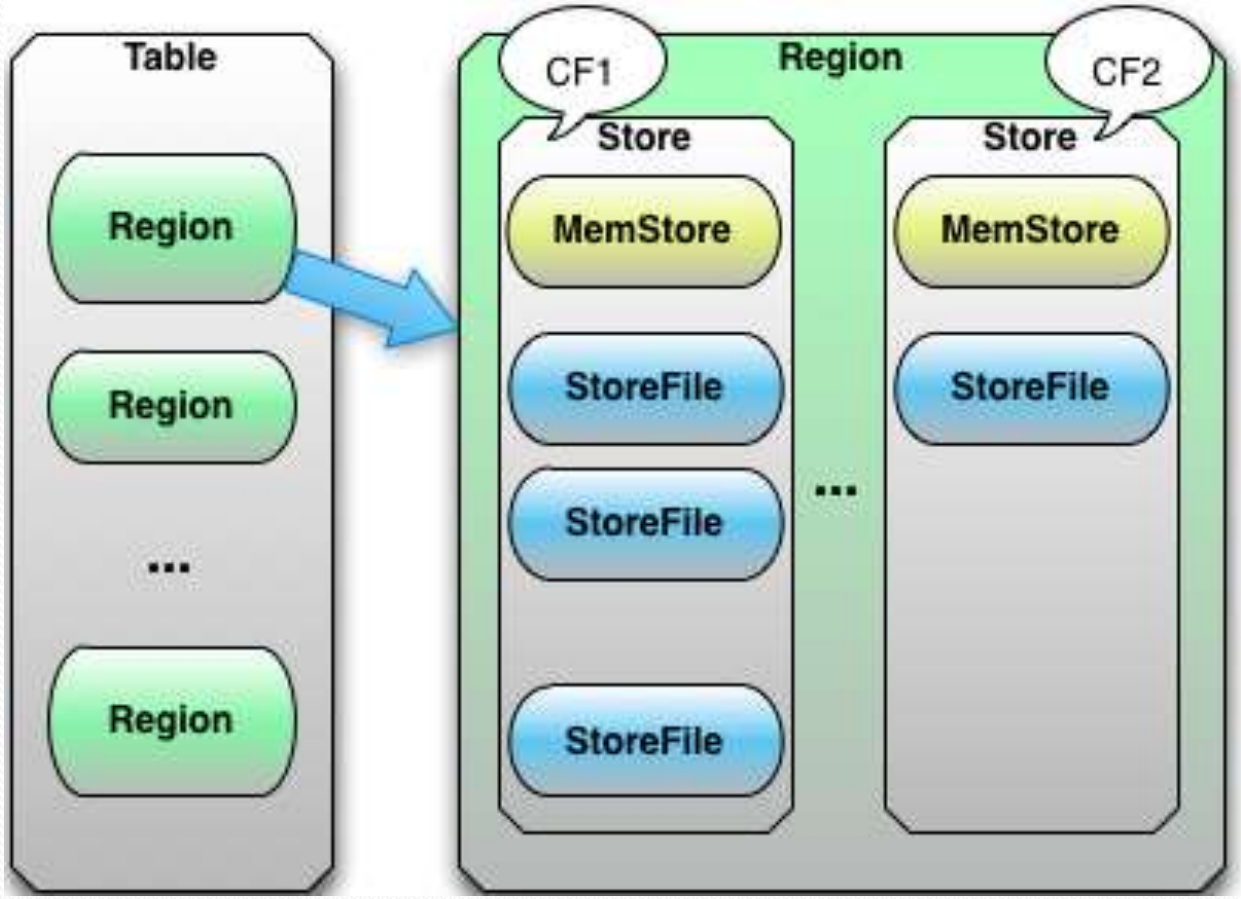
Regions

- Regions are the fundamental partitioning object in HBase.
- A table can be divided horizontally into one or more regions. A region contains a contiguous, sorted range of rows between a start key & an end key.
- Recommended max region size is 10-20Gb
- Default is 256Mb.



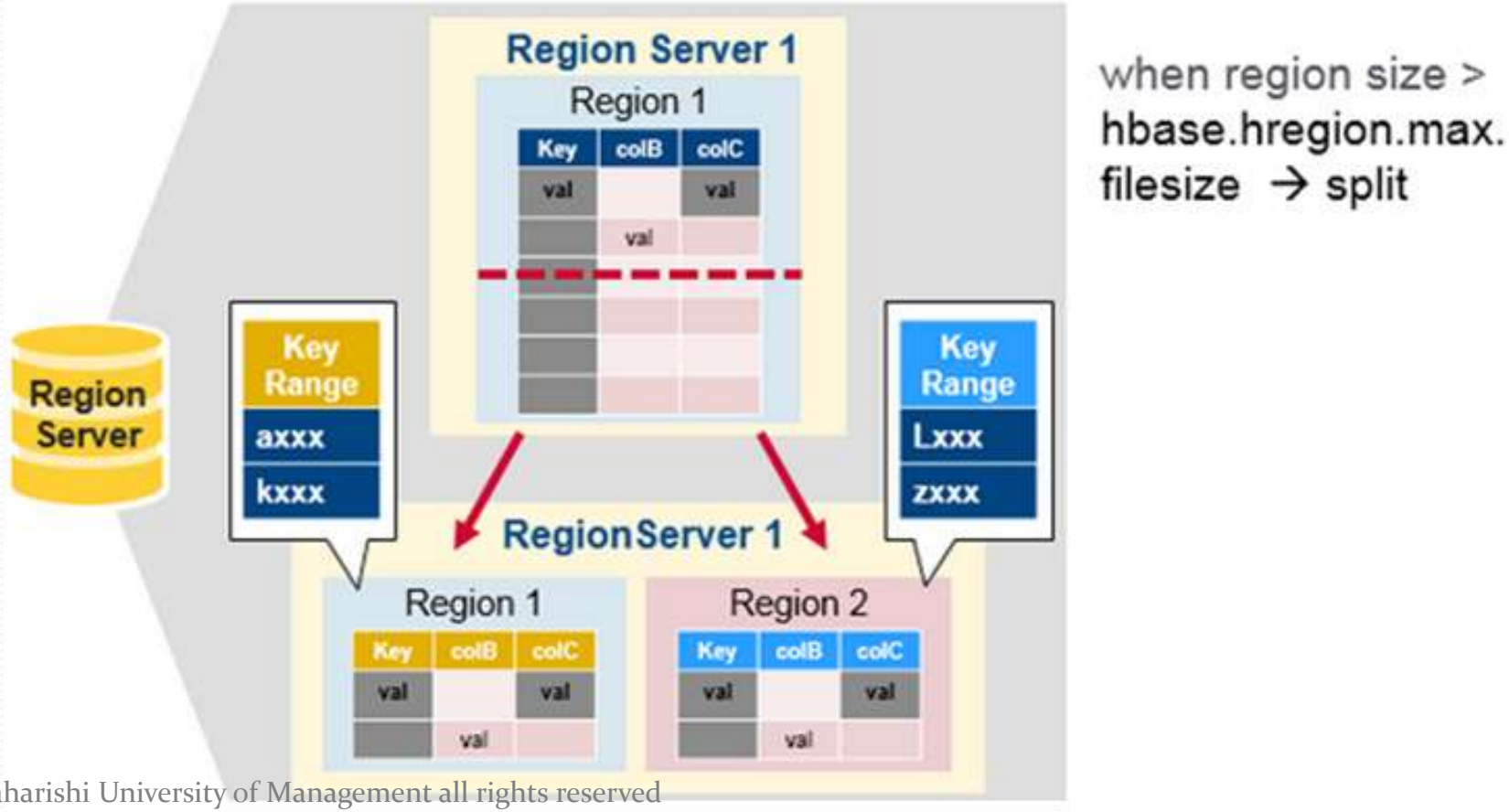
Regions Contd.

- 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)



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 HMaster.



Catalog Tables: -ROOT- & .META.

Since tables are partitioned and stored across the cluster, how can the client find which region is hosting a specific row key range?

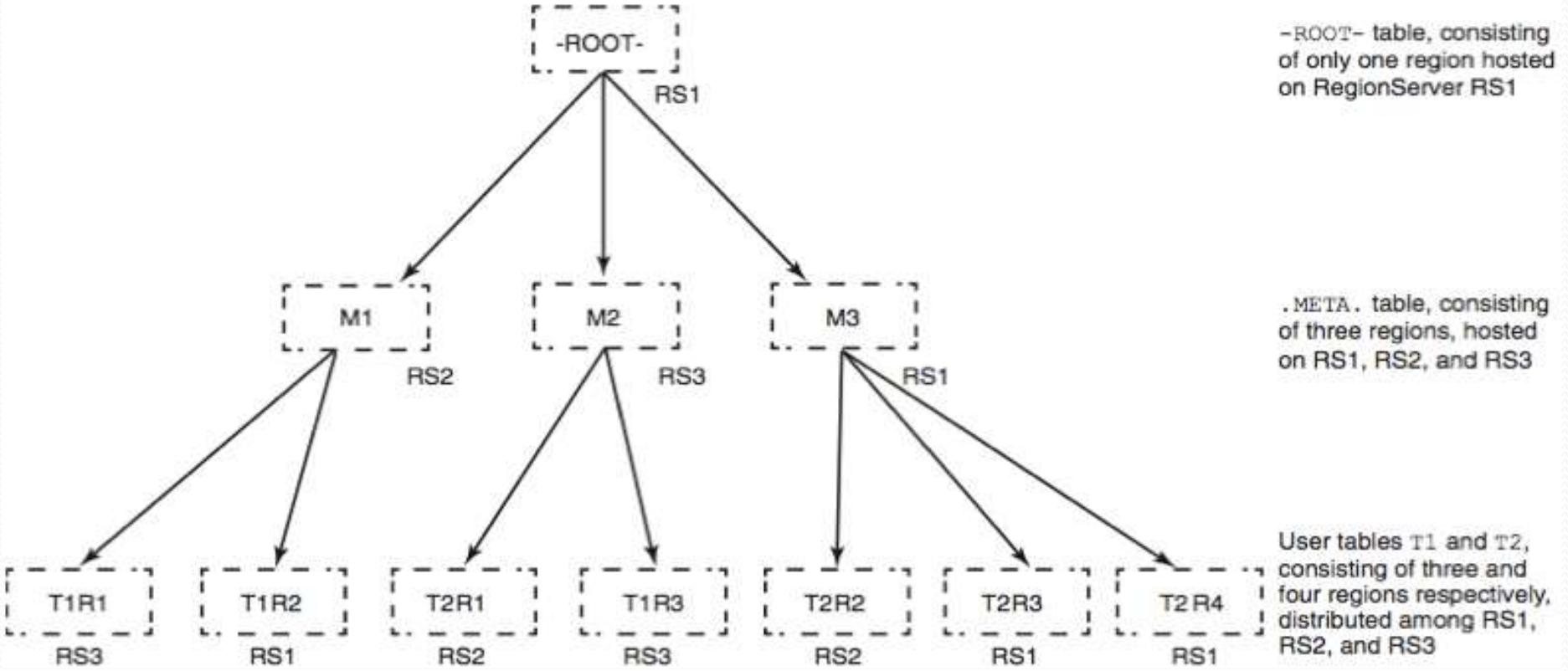
There are two special catalog tables, -ROOT- and .META. for this purpose.

- **-ROOT- table**: hosts the .META. table info.
  - There is only one Region Server which stores the -ROOT- table. And the Root region never split into more than one region.
- **.META. table**: hosts the region location info for a specific row key range.
  - The table is stored on Region Servers, which can be split into as many regions as required.

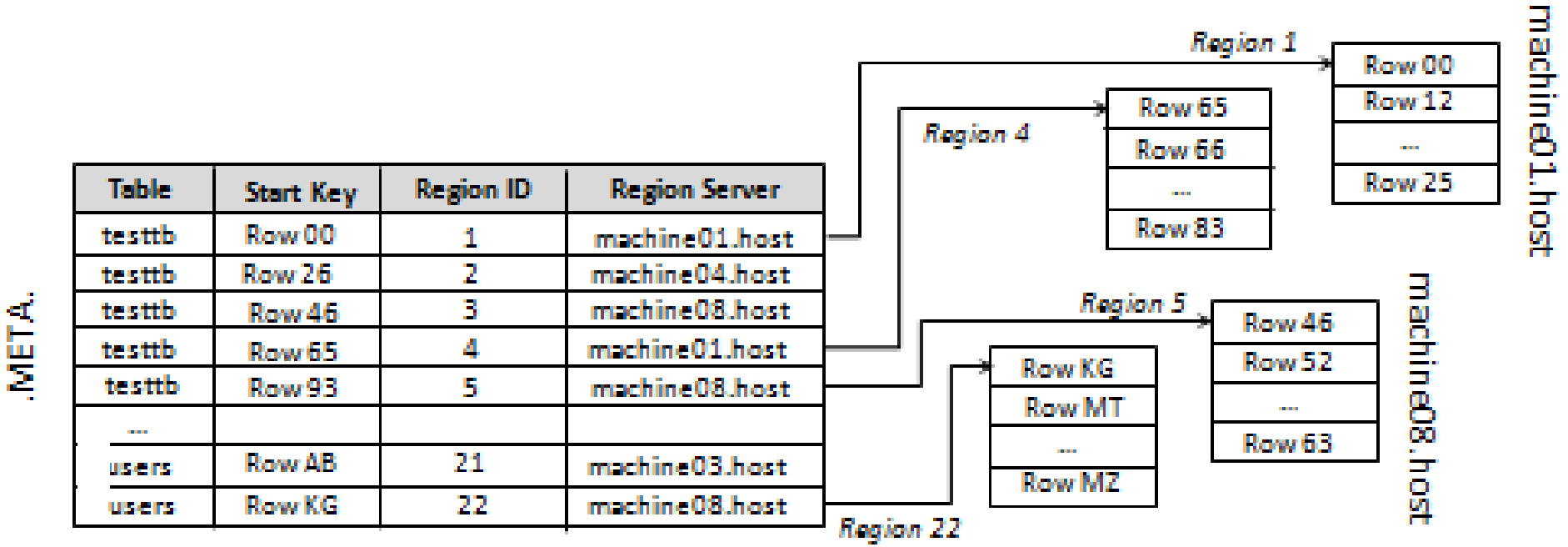
The -ROOT- and .META. table structure logically looks like a B+ tree which is best suited for range queries.

Catalog Tables: -ROOT- & .META.

- The RegionServer RS1 hosts the -ROOT- table
- The .META. table is split into 3 regions: M1, M2, M3, hosted on RS2, RS3, RS1.
- Table T1 contains three regions, T2 contains four regions. For example, T1R1 is hosted on RS3, the meta info is hosted on M1.



.META. Table structure

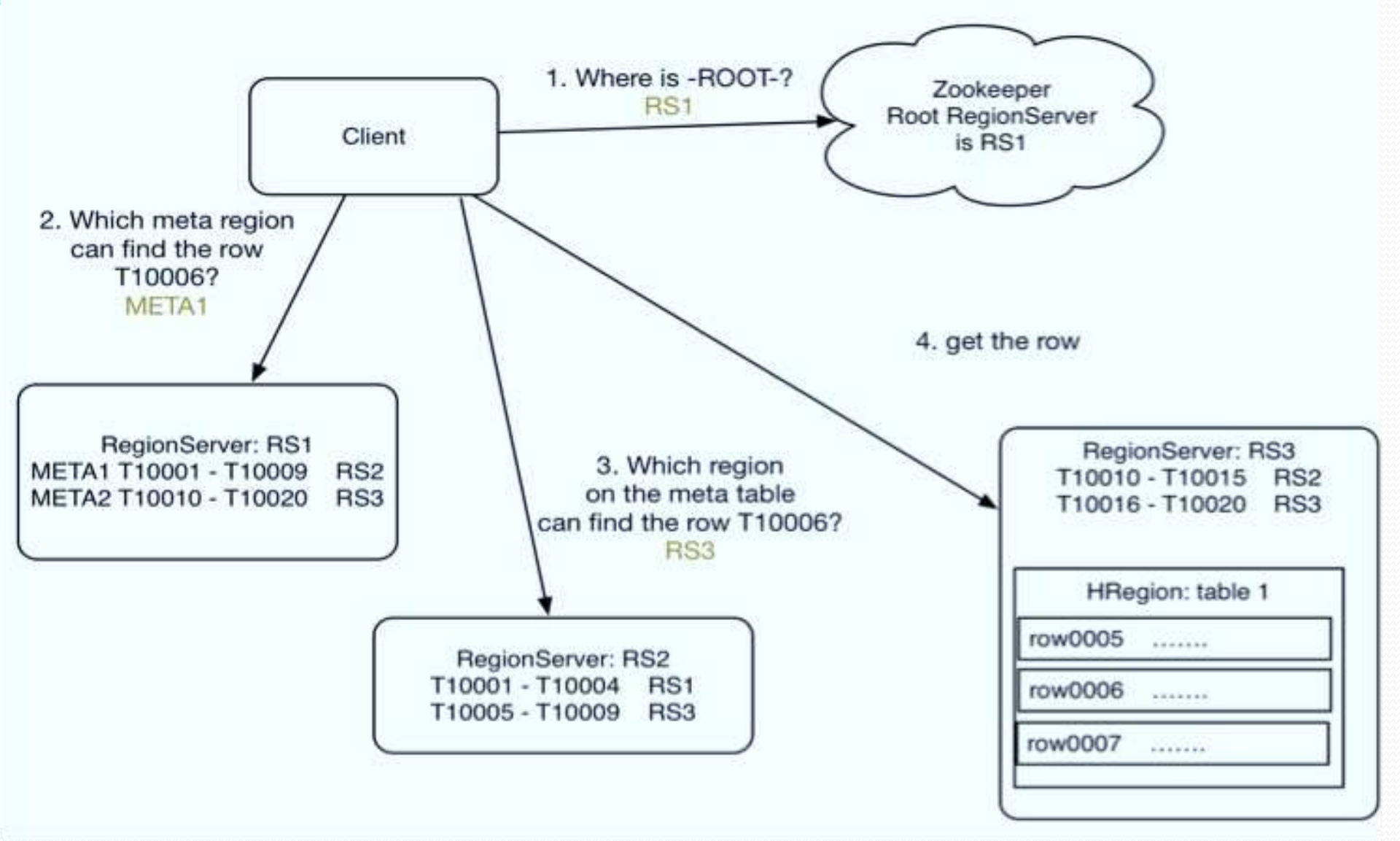


Region Lookup – Read Path

- Whenever a client wants to read data related to a particular row key, it's a 3 step process.

1. How to find the region where the target row is?
   - This information is with the .META. Table.
2. How to find where the .META. Table is?
   - This information is with the -ROOT- table.
3. How to find where the -ROOT- table is?
   - This information can be obtained from ZooKeeper.

Read Path



Region Lookup – Read Path

1. Client queries Zookeeper: where is the –ROOT-?
   On RS1.
2. Client request RS1: Which meta region contains row T10006?
   META1 on RS2
3. Client request RS2: Which region can find the row T10006?
   Region on RS3
4. Client gets the row from the region on RS3
5. Client caches the region info.

- 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 again, unless there is a miss because a region has moved; then it will re-query and update the cache.

Use of BlockCache in Read Path

- BlockCache is a LRU priority cache for data reading.
- 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.
- This 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**.
- HBase data is organized similarly to a sorted map, with the sorted key space partitioned into different shards or 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.

Write Path contd.

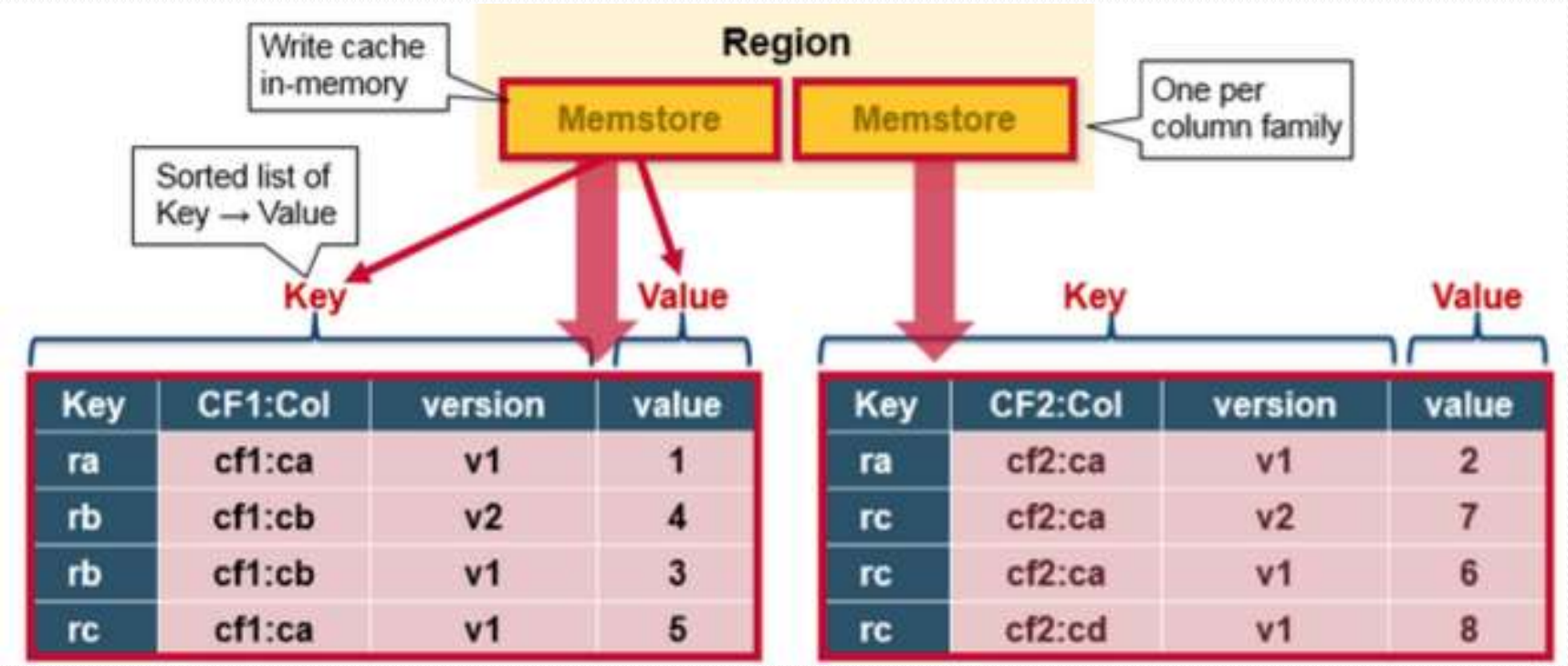
- A change request is for a specific row. Each row key belongs to a specific region which is served by a region server.
- So based on the put or delete's key, an HBase client can locate a proper region server using the 3 step process discussed earlier.
- The region location is cached to avoid this expensive 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. This allows searching for random rows efficiently 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 files 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' 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.

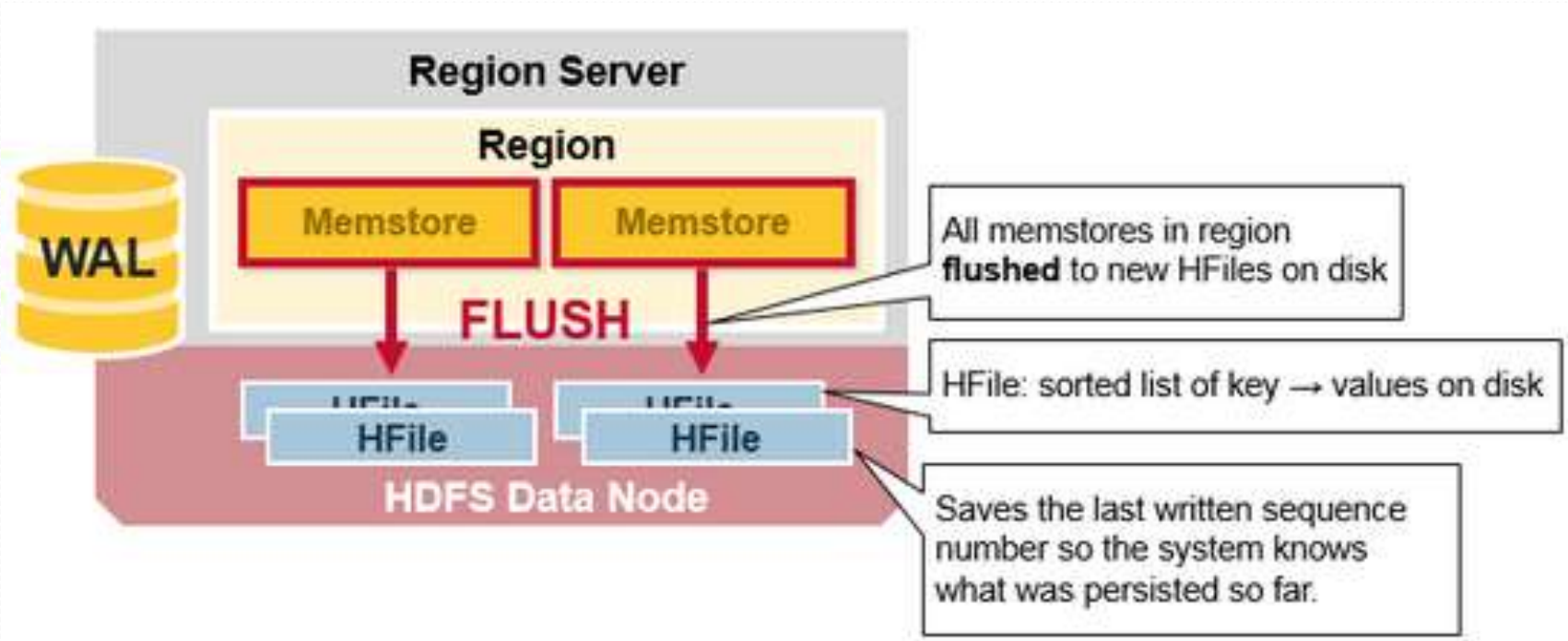


Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

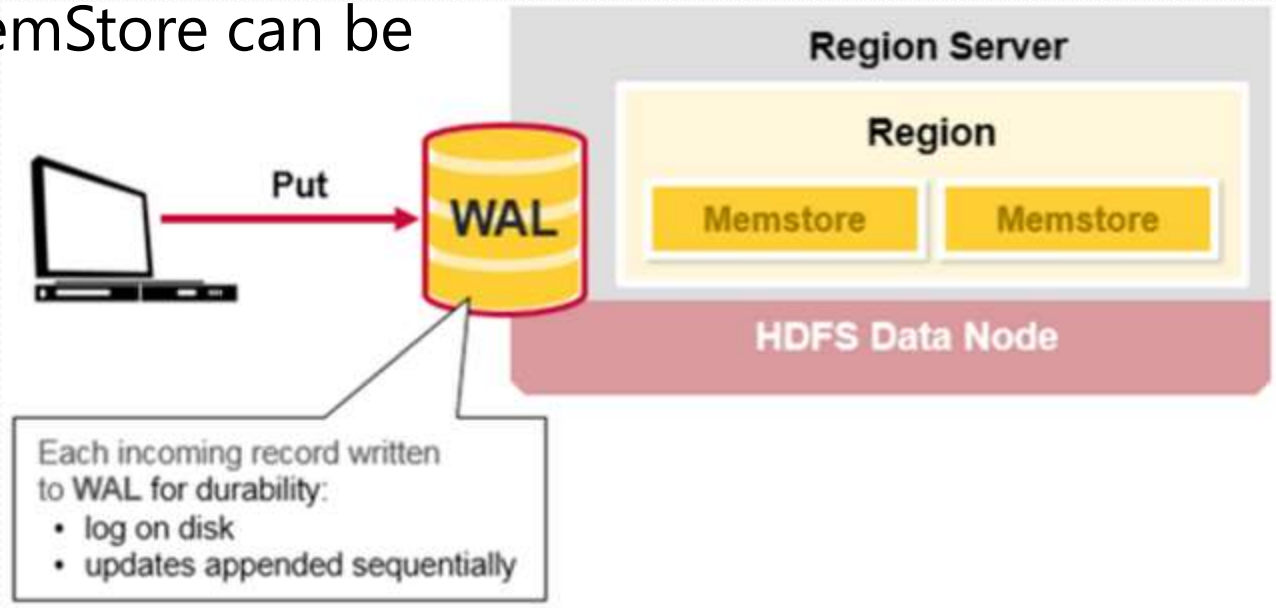
MemStore Flush

- 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 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.



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).

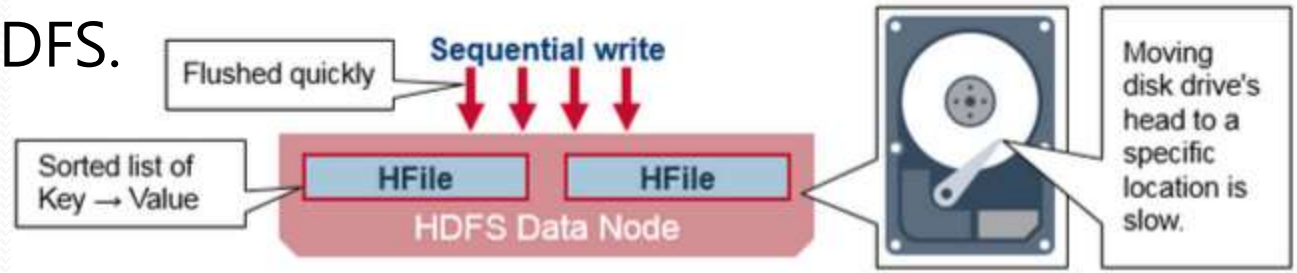


Use of WAL(HLog) in Write Path contd.

- 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.
- HFiles store the rows as sorted Key-Values on disk and according to this order, data is stored and split across the nodes.
- HFile is allocated to 1 region and Row key is the primary identifier. HBase uses multiple HFiles per CF, containing the actual cells, or KeyValue instances.
- 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.

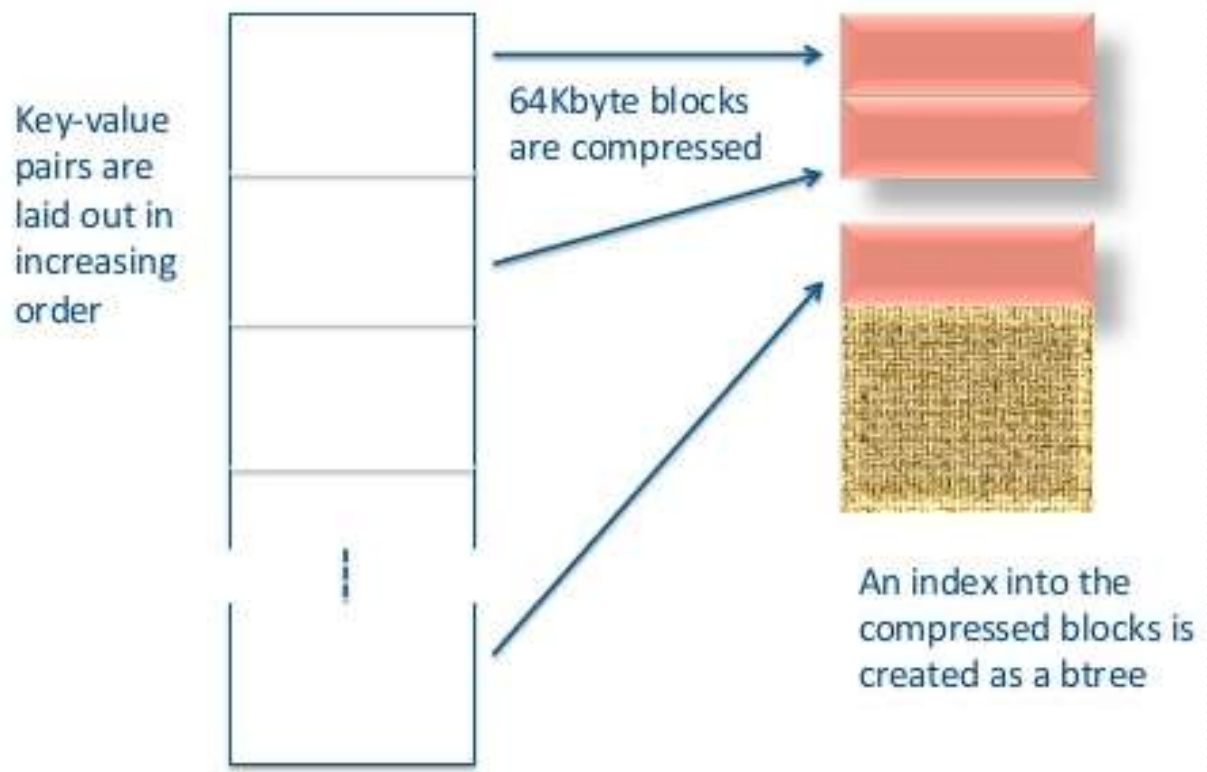


Key	CF1:Col	version	value
ra	cf1:ca	v1	1
rb	cf1:cb	v2	4
rb	cf1:cb	v1	3
rc	cf1:ca	v1	5

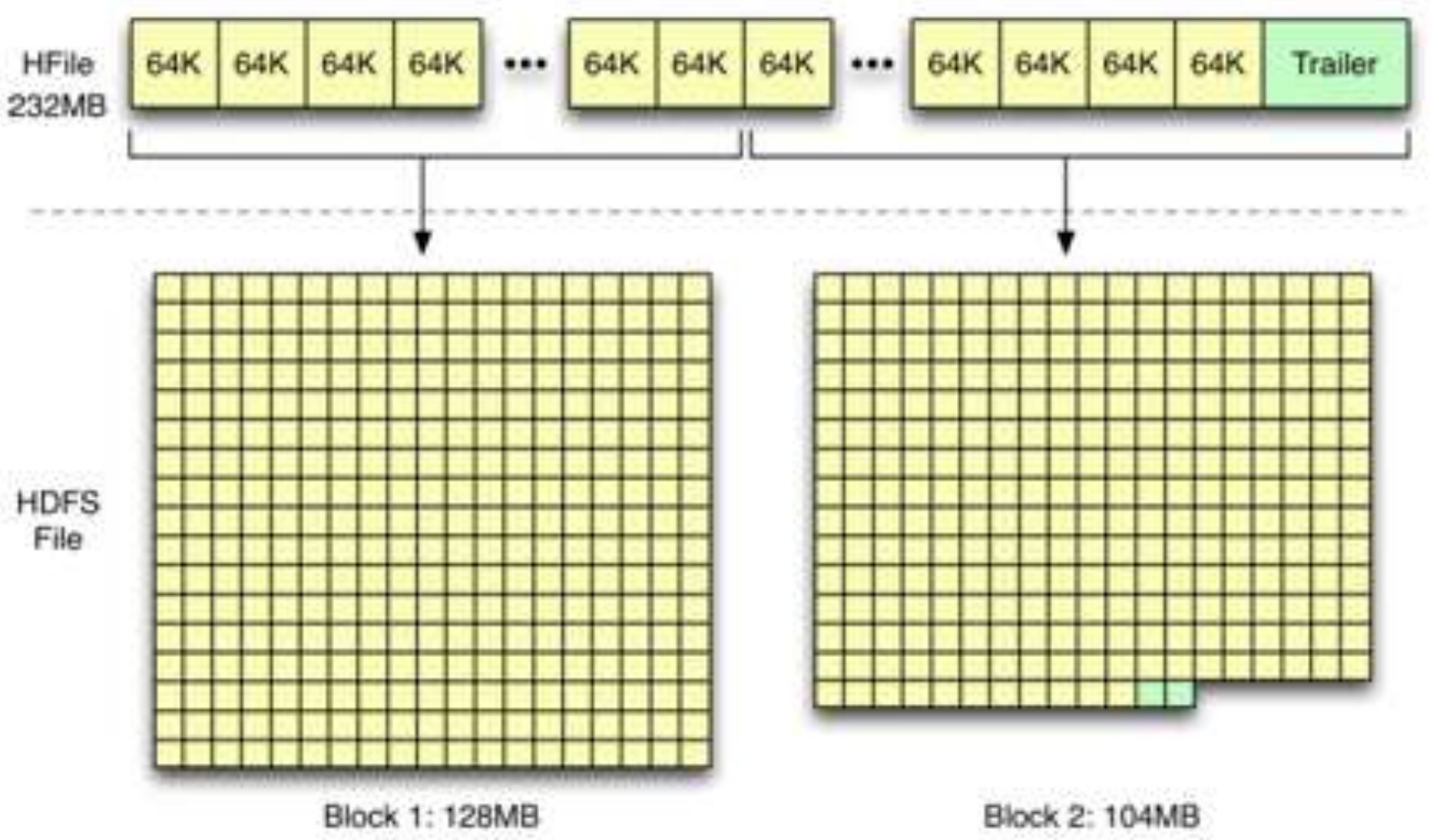
Key	CF2:Col	version	value
ra	cf2:ca	v1	2
rc	cf2:ca	v2	7
rc	cf2:ca	v1	6
rc	cf2:cd	v1	8

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
- A row repeats the key for each column



HFile



HFile Contd.

- 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.
- HDFS replicates the WAL and HFile blocks which happens automatically.
- One HFile can contain only one column family.
- What if data grows in a particular HFile?
  - Different HFile is created with same column family and writing data is continued.
- No two column families can exist in single HFile.

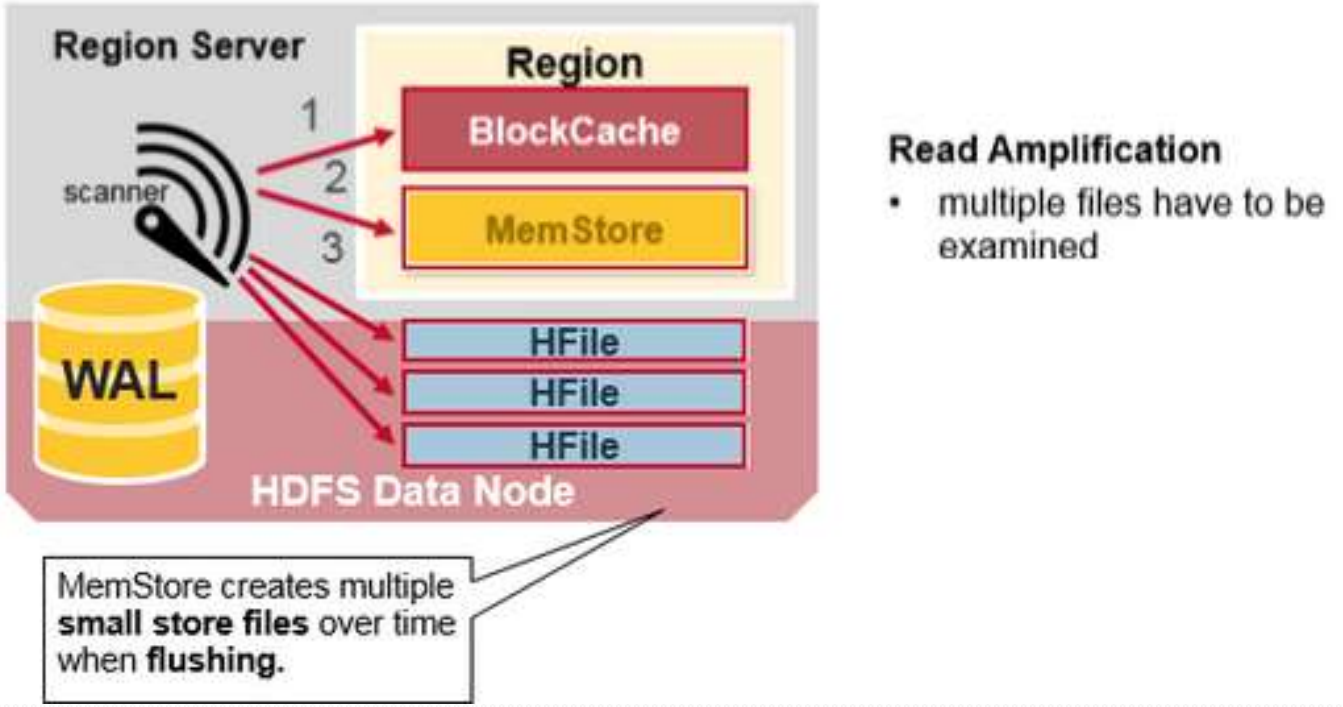
HBase Read Merge

- We have seen that the 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 the MemStore, and recently read cells are in the BlockCache.
- So when you read a row, how does the system get the corresponding cells to return?
- A Read merges Key Values from the BlockCache, MemStore, and HFiles in the following steps:
  - First, the scanner looks for the Row cells in the Block cache - the read cache. Recently Read Key Values are cached here, and Least Recently Used are evicted when memory is needed.
  - Next, the scanner looks in the MemStore, the write cache in memory containing the most recent writes.
  - 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.

HBase Read Merge contd…

- Updating tables means replacing the previous value with the new one. But in HBase if we try to rewrite the column values, it does not overwrite the existing value but rather stores different values per row by time (and qualifier).
- There may be many HFiles per MemStore, which means for a read, multiple files may need to be examined, which can affect the performance. This is called read amplification.



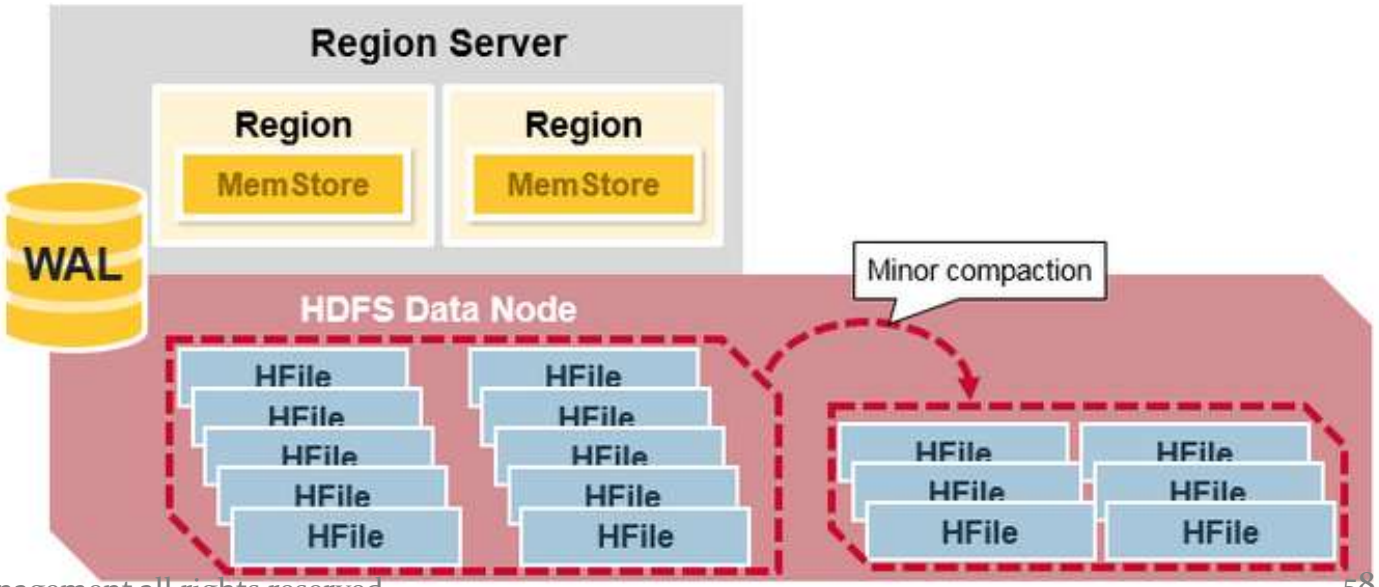
Compactions

- The store files (HFiles) created on disk are immutable. Sometimes the store files are merged together by a process called *compaction* in which excess versions are removed.

- **Two types of compactions:**
  - **Minor compactions**
  - **Major compactions**

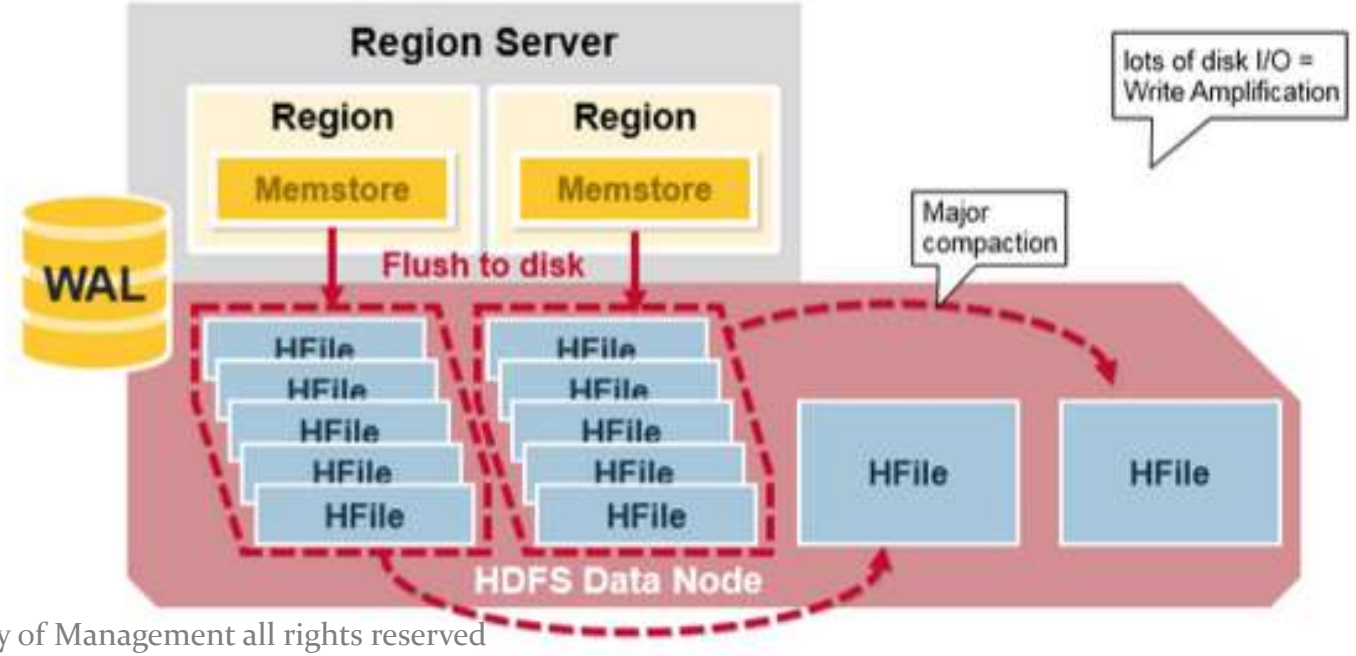
Minor Compactions

- 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 file of a single store inside a single region.



Major Compactions

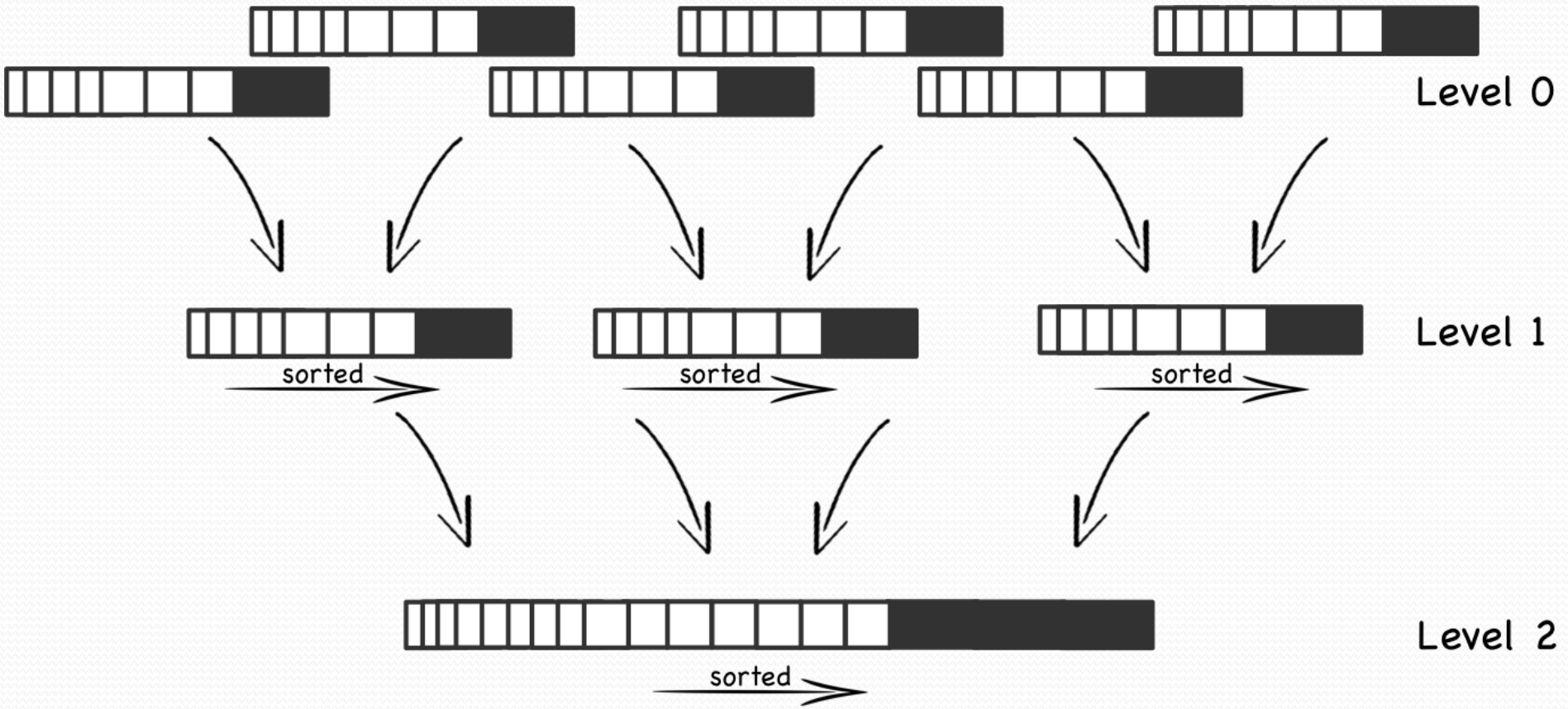
- Major compaction merges and rewrites all the HFiles in a region to one HFile per CF, and in the process, drops deleted or expired cells.
- This improves read performance; however, since major compaction rewrites all of the files, lots of disk I/O and network traffic might occur during the process.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

When to use HBase?

- Good for large amounts of data
  - 100s of millions or billions of rows
  - If data is too small all the records will end up on a single node leaving the rest of the cluster idle
- Carefully evaluate HBase for mixed work loads
  - Client Request vs. Batch processing (MapReduce)
- If you find that your data is stored in collections, for example some meta data, message data or binary data that is all keyed on the same value, then you should consider HBase.
- If you need key based access to data when storing or retrieving, then you should consider HBase.

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 and most of them are null
- There are a lot of real-life implementations of HBase. Some of the important use cases are:
  - Used by Mozilla: They generally store all crash data in HBase
  - Used by Facebook: Facebook uses HBase for real-time messaging.

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
  - May affect response latency!!!
- Bad for traditional RDBMS retrieval
  - Transactional applications
  - Relational Analytics
    - 'group by', 'join', and 'where column like', etc....

When not to use HBase

- Have to have enough hardware!!
  - At the minimum 5 nodes as there are multiple management daemon processes: Namenode, HBaseMaster, Zookeeper, etc....
  - HDFS won't do well on anything under 5 nodes anyway; particularly with a block replication of 3
  - HBase is memory and CPU intensive

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 the following:
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’
  ```